CSCI 331:
Introduction to Computer Security

Lecture 7: Password Cracking, part 2

Instructor: Dan Barowy

Williams

Topics

Address Sanitizer

Keyed Encryption Functions

Password Salts

Precomputed Hash Chains

Your to-dos

1. Read Oeschlin for **Thursday 10/5** and take notes.
2. Lab 3, part 1 **due Sunday 10/8**.

Address Sanitizer

Add
`-fsanitize=address -static-libasan`
to `gcc`'s flags.

Keyed encryption functions

Oeschlin uses terminology like

$$C_0 = S_k(P_0)$$

where S_k is a **cipher** from family S with index k

where P_0 is the **first** (of many) plaintexts in space P

and where C_0 is the **first** (of many) ciphertexts in space C

Keyed encryption functions

Example cipher family

ROT-0

a	b	c	…	x	y	z
↓	↓	↓		↓	↓	↓
a	b	c	…	x	y	z

I	B	M
↓	↓	↓
I	B	M

Keyed encryption functions

Example cipher family

ROT-1

a	b	c	…	x	y	z
↓	↓	↓		↓	↓	↓
b	c	d	…	y	z	a

I	B	M
↓	↓	↓
J	C	N

Keyed encryption functions

Example cipher family

ROT-25

a	b	c	…	x	y	z
↓	↓	↓		↓	↓	↓
z	a	b	…	w	x	y

I	B	M
↓	↓	↓
H	A	L

Keyed encryption functions

Example cipher family

Keyed encryption functions

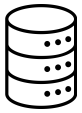
```
def rot(i, p):
  alpha = "abcdefghijklmnopqrstuvwxyz"
  j = i % len(alpha)
  out = ""
  for char in p:
    idx = alpha.index(char)
    idx2 = (idx + j) % len(alpha)
    out += alpha[idx2]
  return out

print(rot(25, "ibm"))
```

Password salts

User types: `password`

`login` prepends salt: `pxZ6j!`



Database stores: `pxZ6j!password`

Interesting fact about salts: usually stored in plaintext!

When you change your password, the */bin/passwd* program selects a salt based on the time of day. The salt is converted into a two-character string and is stored in the */etc/passwd* file along with the encrypted “password.” In this manner, when you type your password at login time, the same salt is used again. Unix stores the salt as the first two characters of the encrypted password.

— Practical UNIX and Internet Security, 3rd Edition by Simson Garfinkel, Gene Spafford, Alan Schwartz

Precomputed Hash Chains

Precomputed Hash Chains

Motivation: dictionaries are **too big** to distribute

Recall:

About 29 terabytes!

Want: something smaller

Ordinary dictionary

plaintext ciphertext

Ordinary dictionary

hash( )

plaintext ciphertext

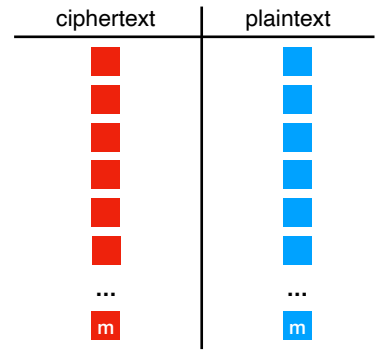
Ordinary dictionary

plaintext ciphertext

Ordinary dictionary

m = # of possible passwords

So storing a table takes roughly O(**m**) space.



plaintext ciphertext

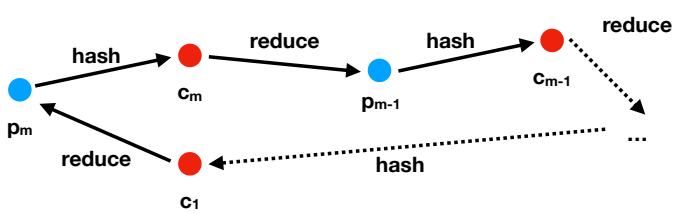
We can enumerate all plaintexts*

plaintexts

ciphertexts

Suppose f(p_i) = c_i

Suppose $r(c_i)$ = p_{i-1} if i>1 otherwise p_m



*in principle, assuming no collisions

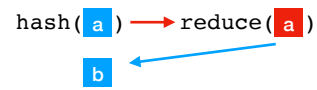
Hash chain

plaintext ciphertext

Hash chain

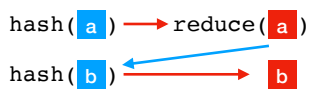
plaintext ciphertext

Hash chain



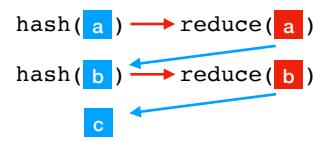
plaintext ciphertext

Hash chain



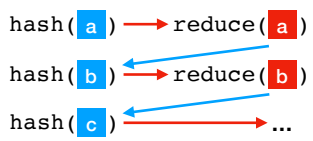
plaintext ciphertext

Hash chain



plaintext ciphertext

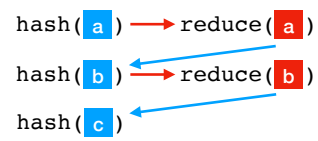
Hash chain



plaintext ciphertext

Hash chain of length 2

Chain length: # of calls to `reduce()`



plaintext ciphertext

Hash chain of length 2

Discard everything but **start** and **end** points and store them in a table

start	end
a	c

plaintext ciphertext

Hash chain of length 2

But store the **end** as the **`key`** and **start** as the **`data`**.

(e.g., using the `hsearch(3)` implementation we explored in lab)

end	start
c	a

plaintext ciphertext

Hash chain of length 2

Each chain contains roughly **k** plaintexts

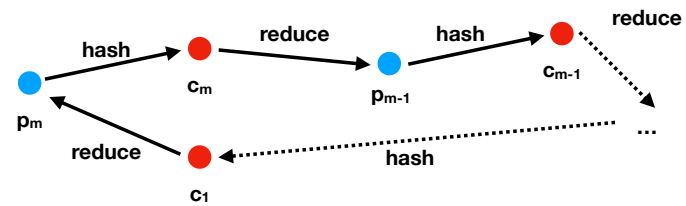
So storing a table now takes roughly O(**m/k**) space.

end	start
c	a
f	d
i	g
l	j
o	m
...	...

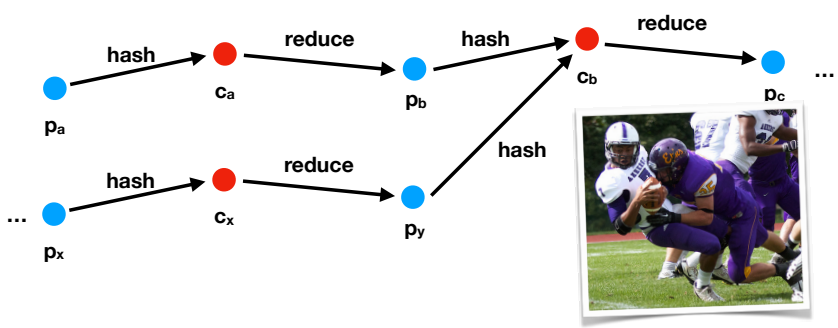
plaintext ciphertext

Thought experiment: drawbacks

- Saving just the first password **buys us nothing**. On average, we have to compute **O(m/2)** hash-reductions to find a password.
- It is **probably not possible** to find a reducer that lets you explore the entire password space.
- **Hash functions collide!**



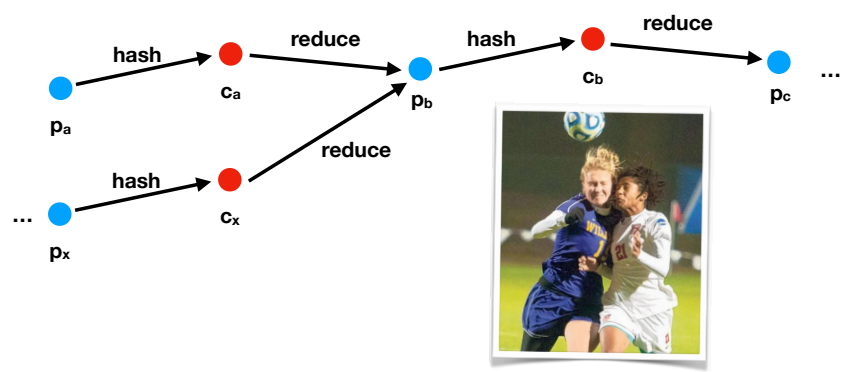
Collisions from hash functions



After the **collision**, the chain "**loops**."

Collisions prevent us from enumerating the **entire space**!

Collisions from reducers

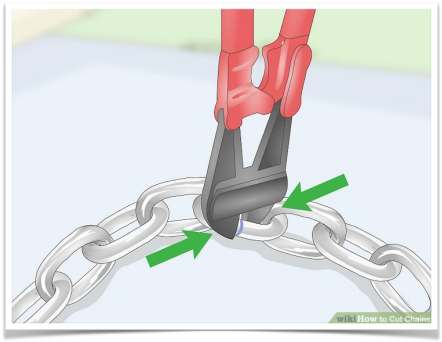


Reducers can produce collisions too!

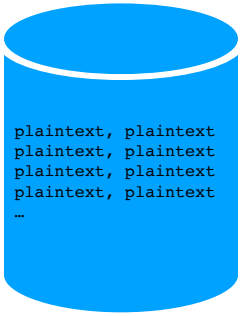
This is what we mean by an **imperfect reducer**.

Hash chain of length k

We are going to chop up our long chain into **smaller chains** of length **k**.



Precomputed hash chain



Ahead of time:

```
while i < NUM_PT:
    // gen ith possible plaintext
    p = genPassword(i)
    start = p
    for j from 0 to k-1:
      // create ciphertext
      c = hash(p)
      // reduce
      p = reduce(c)
    // save chain in table
    table[p] = start
    i++
```

Suppose you are given c_4

end,	start
p_{m-3} ,	p_m
…	
p_3 ,	p_5
p_1 ,	p_3

$$c_4 \xrightarrow{\text{reduce}} p_3$$

Is p_3 an **end point**? **yes**

Hash and **reduce** from **start point**.

password! original ciphertext

$$p_5 \xrightarrow{\text{hash}} c_5 \xrightarrow{\text{reduce}} p_4 \xrightarrow{\text{hash}} c_4$$

Hash chain lookup pseudocode

chain table

ciphertext

chain length

```
def lookup(c, db, len):
  // look for endpoint
  p = reduce(c)
  i = 0
  while p not end in db && i < len:
    c2 = hash(p)
    p = reduce(c2)
    i++
  if p not end: FAIL
  // we found chain; lookup start pt
  s = db[p]
  c2 = hash(s)
  // decrypt
  i = 0
  while c2 != c && i < len:
    s = reduce(c2)
    c2 = hash(s)
    i++
  if i == len: FAIL
  return s
```

end,start

♥♥★♥,♥♥♥♥

♥★★★,♥♥♥★

★♥★★,★★♥★

★★★★,♥★♥★

chain table

Stores **plaintexts**!

Hash chain lookup pseudocode

chain table

ciphertext

chain length

```
def lookup(c, db, len):
  // look for endpoint
  p = reduce(c)
  i = 0
  while p not end in db && i < len:
    c2 = hash(p)
    p = reduce(c2)
    i++
  if p not end: FAIL
  // we found chain; lookup start pt
  s = db[p]
  c2 = hash(s)
  // decrypt
  i = 0
  while c2 != c && i < len:
    s = reduce(c2)
    c2 = hash(s)
    i++
  if i == len: FAIL
  return s
```

end,start

♥♥★♥,♥♥♥♥

♥★★★,♥♥♥★

★♥★★,★★♥★

★★★★,♥★♥★

chain table

Stores **plaintexts**!

Class Activity

Decrypt the hash

7F975A56C761DB6506ECA0B37CE6EC87

Answer:

★♥★★

Recap & Next Class

Today we learned:

PCHC algorithm

Next class:

Rainbow algorithm