CSCI 331:
Introduction to Computer Security

Lecture 14: Stack Smashing

Instructor: Dan Barowy

Williams

Topics

Buffer overflow exploit: overview

Using GDB to analyze a program

Crafting inputs

Your to-dos

1. Read *Undefined Behavior* (Wang) **for Thu 11/2**.
2. Lab 5, **due Sunday 11/5**.

Recall from last class

```
void foo() {}

int main() {
  foo();
}
```

The compiled program

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

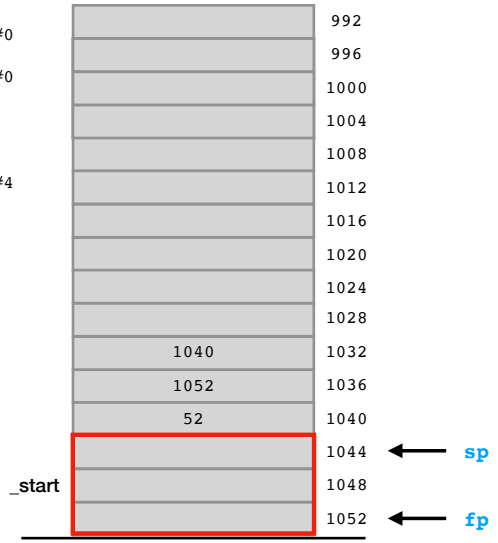
Class Activity

Everything is **back to where it started** except `pc`, which is advanced to `52`.

```
foo:
0  push  {fp}
4  add   fp, sp, #0
8  nop
12 add   sp, fp, #0
16 pop   {r11}
20 bx lr
main:
24 push  {fp, lr}
28 add   fp, sp, #4
32 bl foo
36 mov   r3, #0
40 mov   r0, r3
44 pop   {fp, pc}
_start:
…
48 bl    main
52 …
```

pc →

r0 = 0
r3 = 0
fp =1052
sp =1044
lr = 36
pc = 52



Observations

- After a function is "torn down," **everything (that matters) is back where it was** before the call, **except** that `pc` is **advanced**.
- Notice that the `pc` saved on the stack is the **next instruction to run** after a return. All instructions except `b/bl/bx` (and a `pop` special case) advance `pc`.
- (You can't `push pc`!)
- Values are left on the stack. **Nobody cleans up!**
- Automatic variables: **only sort-of reclaimed**.
- Sometimes `gcc` **adds** `NOP` instructions. In general, these are added to align branches to 16-byte boundaries.

What are the parts of this program?

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

What are the parts of this program?

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

func. prologue: **callee** sets up stack for **itself**.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: callee sets up stack for itself.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: callee sets up stack for itself.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: callee sets up stack for itself.

func. epilogue: callee restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: callee sets up stack for itself.

func. epilogue: callee restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: **callee** sets up stack for **itself**.

func. epilogue: **callee** restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: **callee** sets up stack for **itself**.

transfer of control: **caller** gives control to **callee**.

func. epilogue: **callee** restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. prologue: **callee** sets up stack for **itself**.

transfer of control: **caller** gives control to **callee**.

func. epilogue: **callee** restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
    push  {r11}
    add fp, sp, #0
    nop
    add sp, fp, #0
    pop {r11}
    bx  lr
main:
    push  {fp, lr}
    add fp, sp, #4
    bl  foo
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
```

func. body: where **callee** does work (nothing here).

func. prologue: **callee** sets up stack for **itself**.

transfer of control: **caller** gives control to **callee**.

func. epilogue: **callee** restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

func. body: where **callee** does work (nothing here).

func. prologue: **callee** sets up stack for **itself**.

transfer of control: **caller** gives control to **callee**.

func. epilogue: **callee** restores stack & returns.

What are the parts of this program?

func. labels: where a function **starts**.

```
foo:
  push  {r11}
  add fp, sp, #0
  nop
  add sp, fp, #0
  pop {r11}
  bx  lr
main:
  push  {fp, lr}
  add fp, sp, #4
  bl  foo
  mov r3, #0
  mov r0, r3
  pop {fp, pc}
```

func. body: where **callee** does work (nothing here).

func. prologue: **callee** sets up stack for **itself**.

transfer of control: **caller** gives control to **callee**.

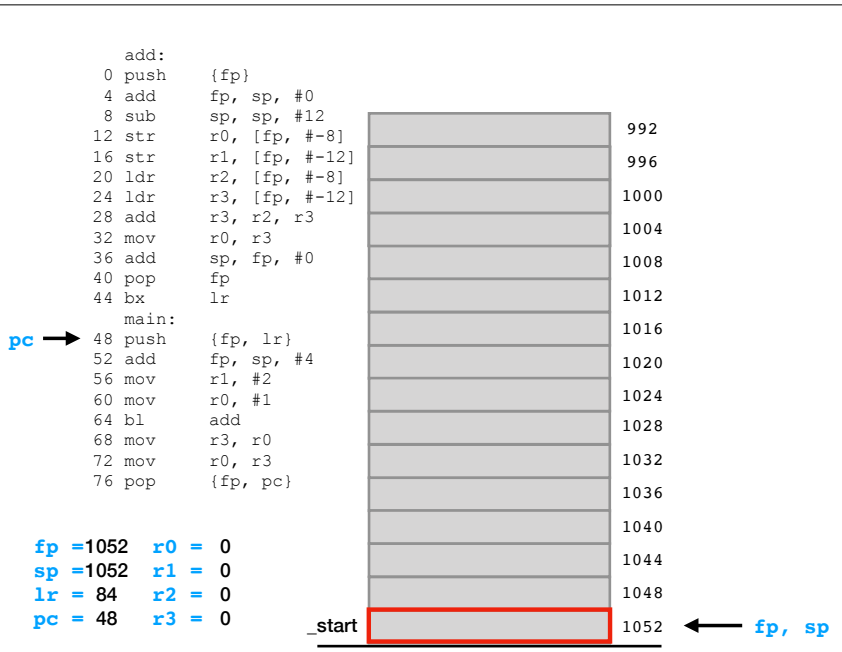
return value: **callee** prepares return value for **caller**.

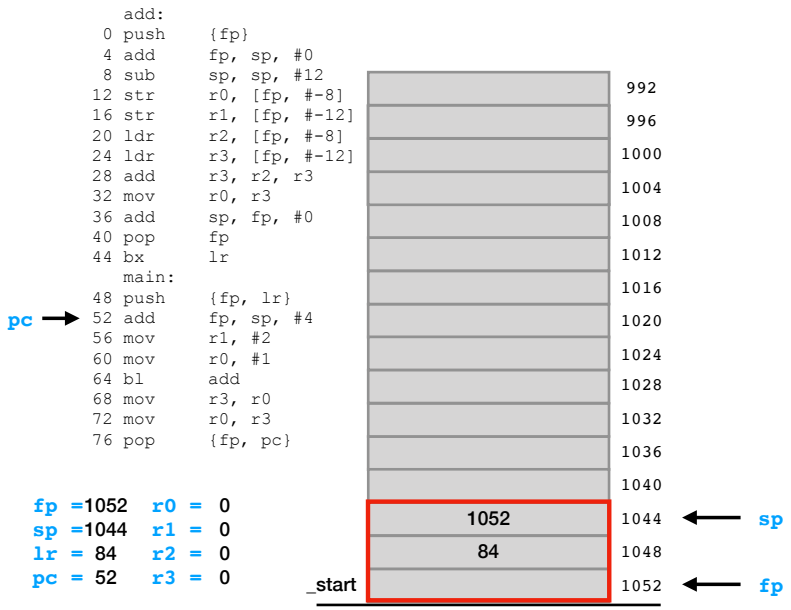
func. epilogue: **callee** restores stack & returns.

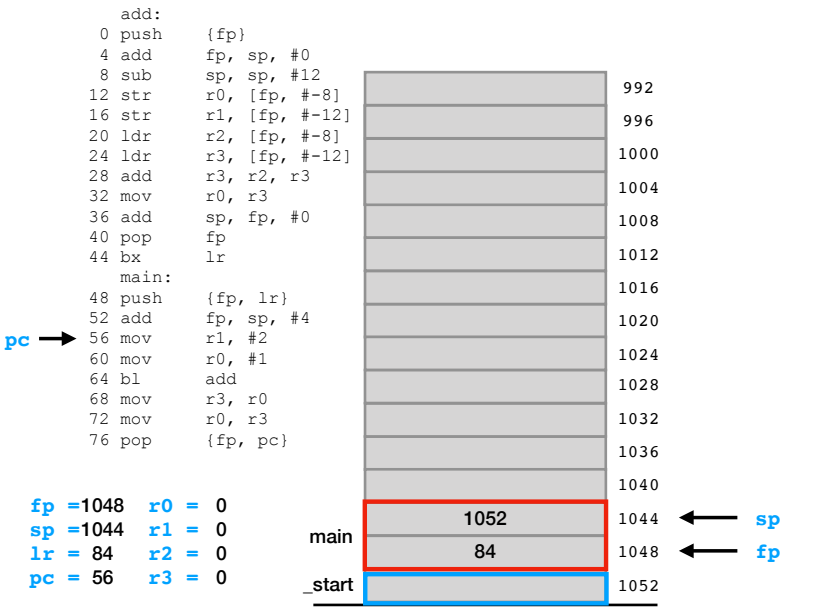
Arguments

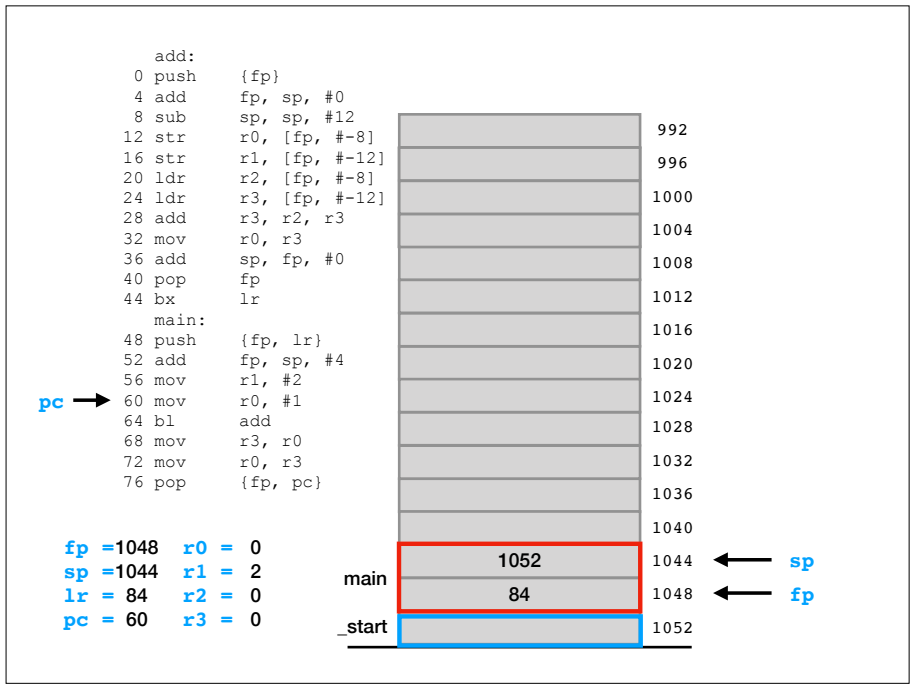
```c
int add(int a, int b) {
  return a + b;
}

int main() {
  return add(1, 2);
}
```

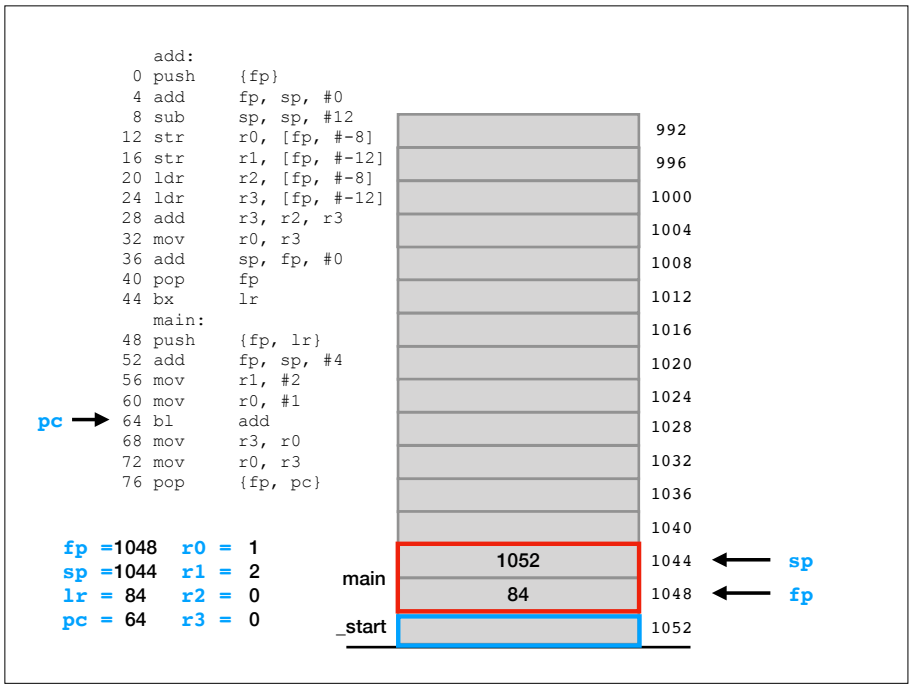
```
   add:
 0         push    {fp}
 4         add     fp, sp, #0
 8         sub     sp, sp, #12
12         str     r0, [fp, #-8]
16         str     r1, [fp, #-12]
20         ldr     r2, [fp, #-8]
24         ldr     r3, [fp, #-12]
28         add     r3, r2, r3
32         mov     r0, r3
36         add     sp, fp, #0
40         pop     fp
44         bx      lr
   main:
48         push    {fp, lr}
52         add     fp, sp, #4
56         mov     r1, #2
60         mov     r0, #1
64         bl      add
68         mov     r3, r0
72         mov     r0, r3
76         pop     {fp, pc}
```

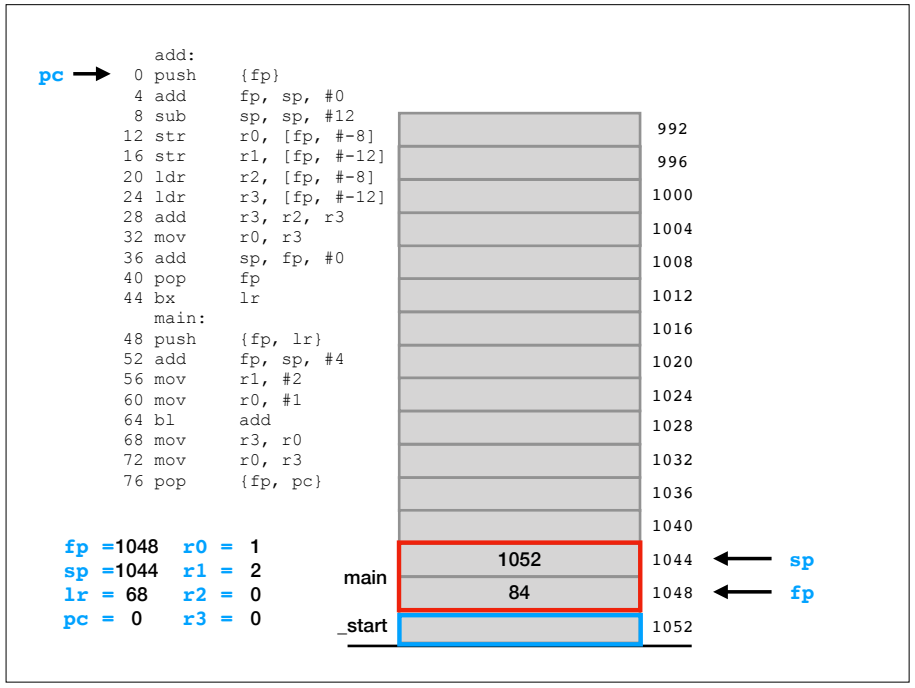


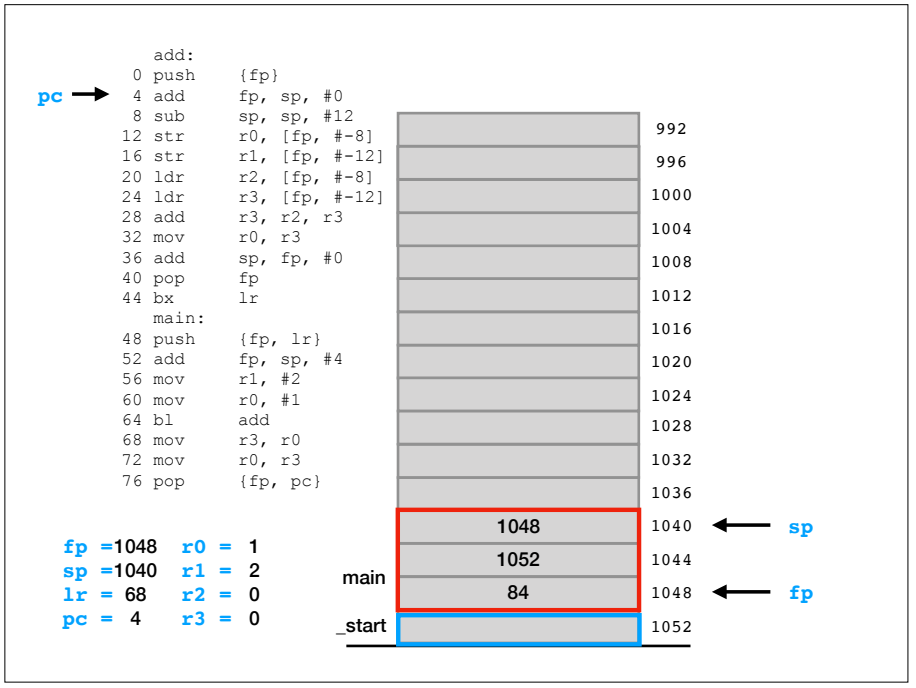


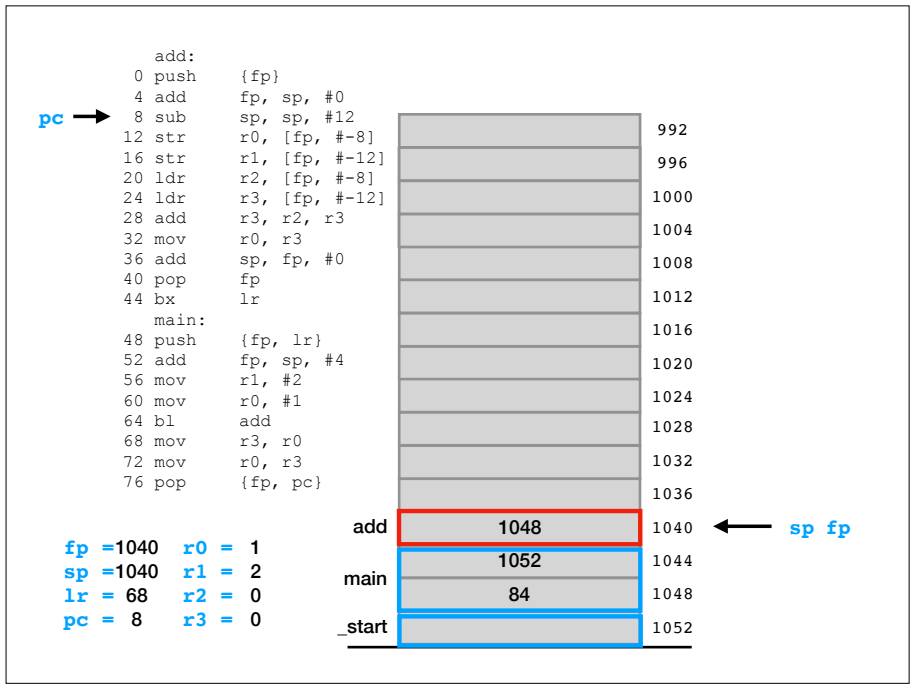


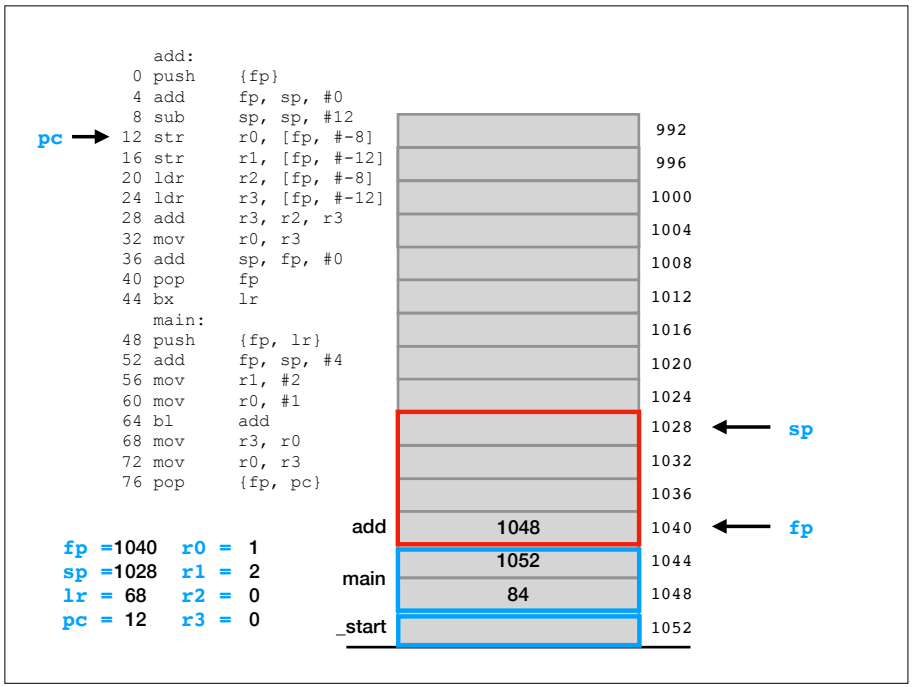

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1048 r0 = 0
sp =1044 r1 = 2
lr = 84 r2 = 0
pc = 60 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
1052
84
main
_start
sp
fp

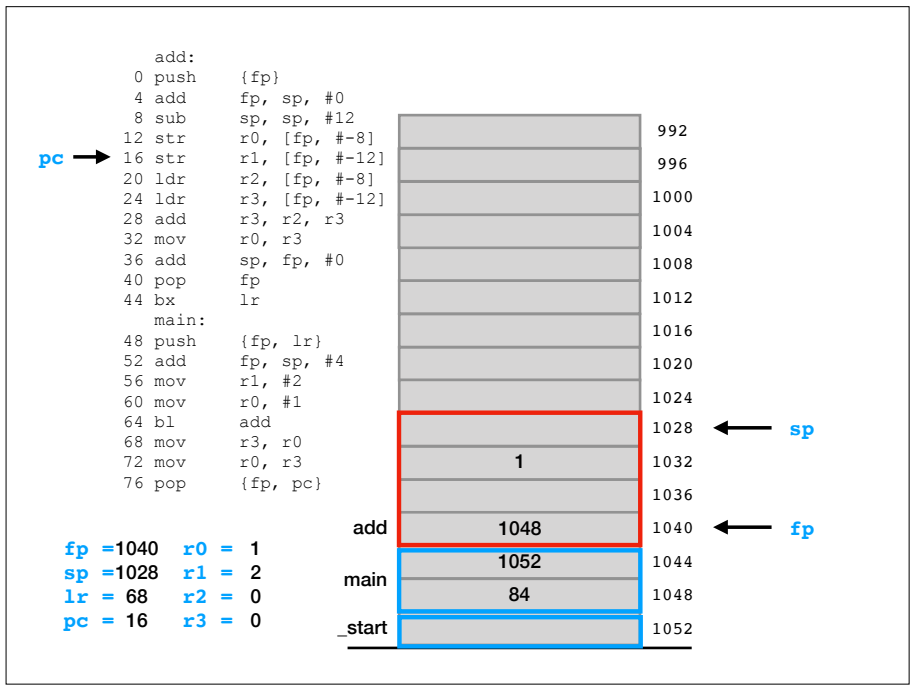

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1048 r0 = 1
sp =1044 r1 = 2
lr = 84 r2 = 0
pc = 64 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
1052
84
main
_start
sp
fp

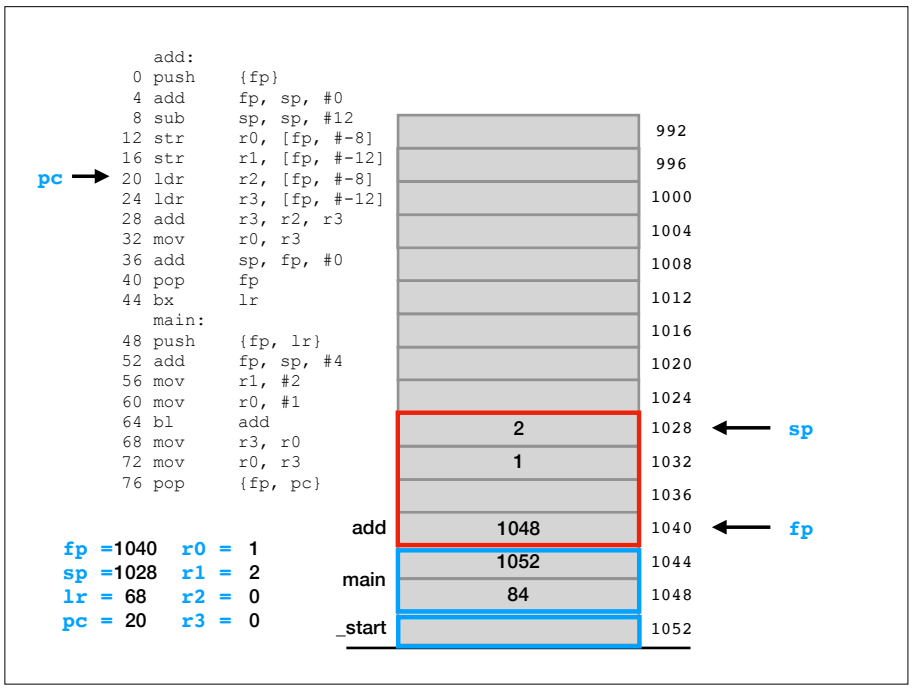

add:
pc
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
fp =1048 r0 = 1
sp =1044 r1 = 2
lr = 68 r2 = 0
pc = 0 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
1052
84
main
_start
sp
fp

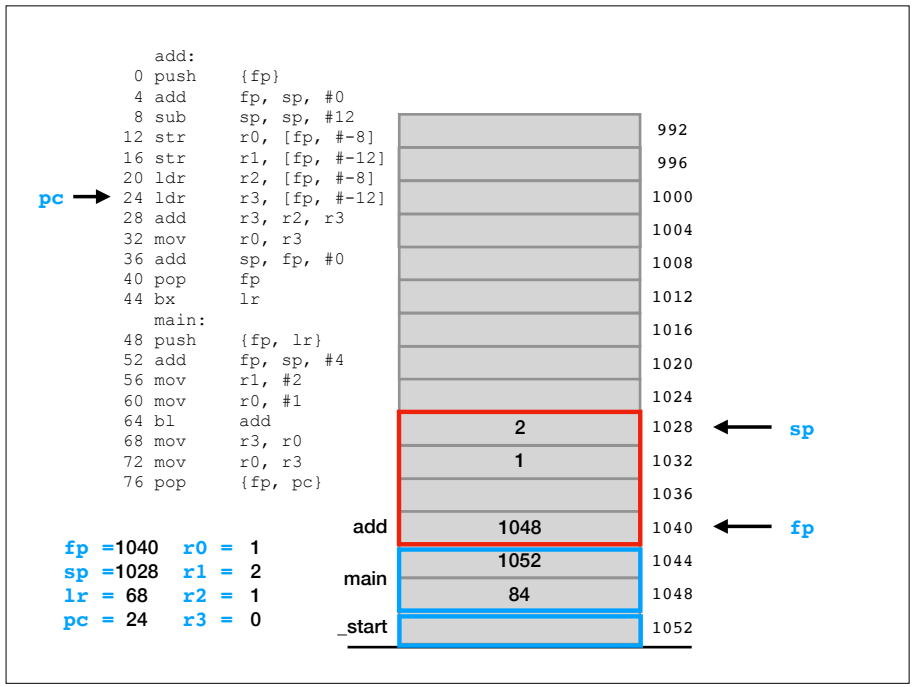

add:
pc
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
fp =1048 r0 = 1
sp =1040 r1 = 2
lr = 68 r2 = 0
pc = 4 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
1048
1052
84
main
_start
sp
fp

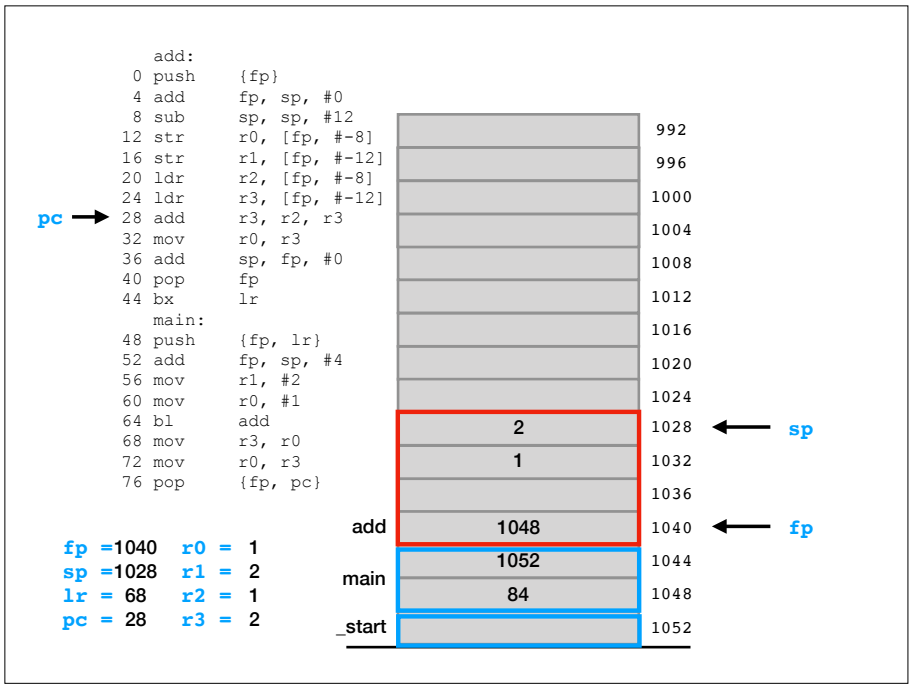

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1040 r1 = 2
lr = 68 r2 = 0
pc = 8 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
add
1048
1052
main
84
_start
sp fp

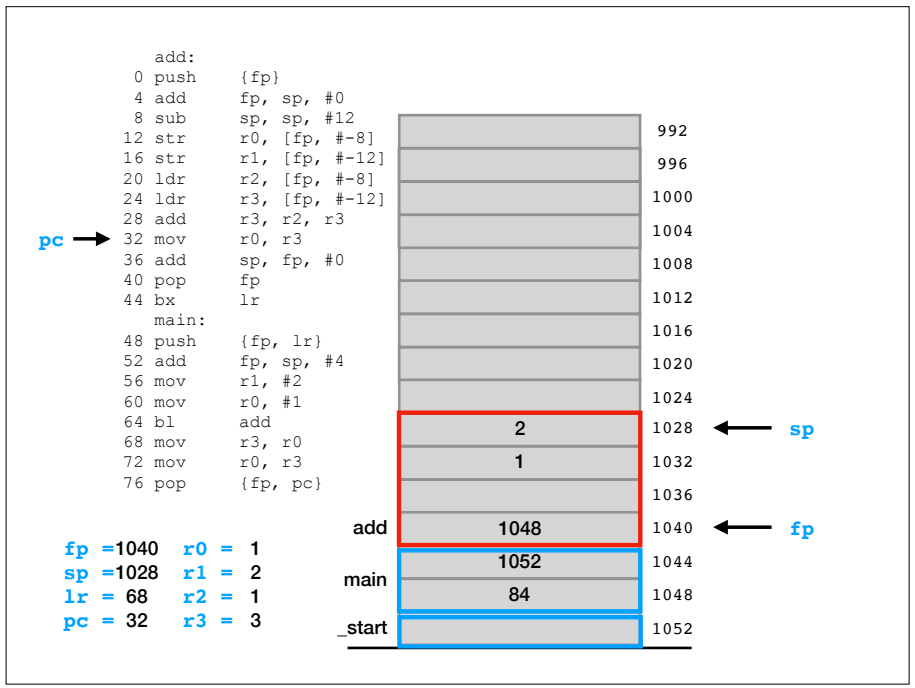

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 0
pc = 12 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
add
1048
1052
main
84
_start
sp
fp

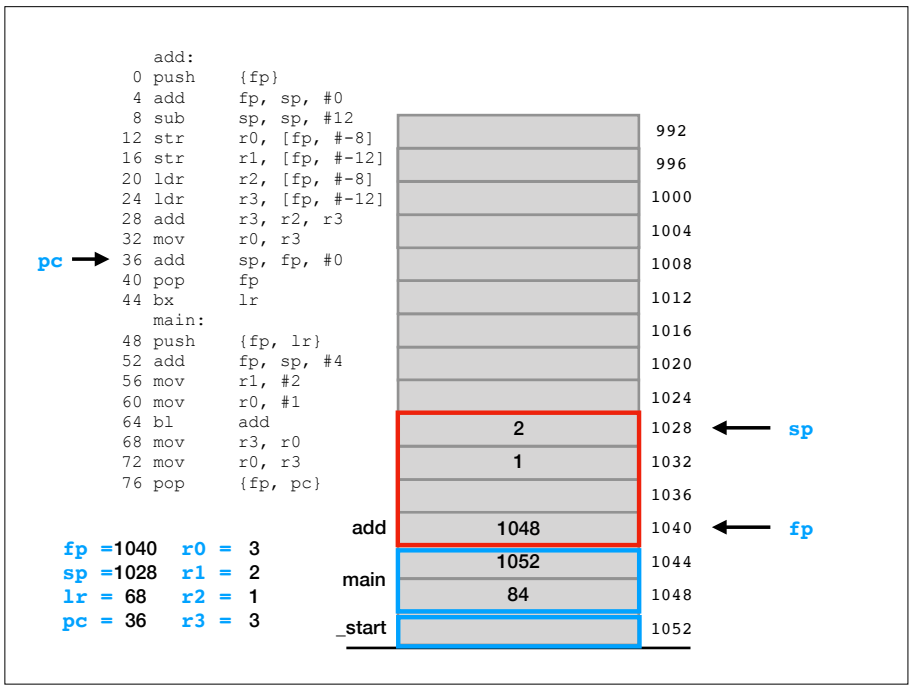

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 0
pc = 16 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
1
add
1048
1052
main
84
_start
sp
fp

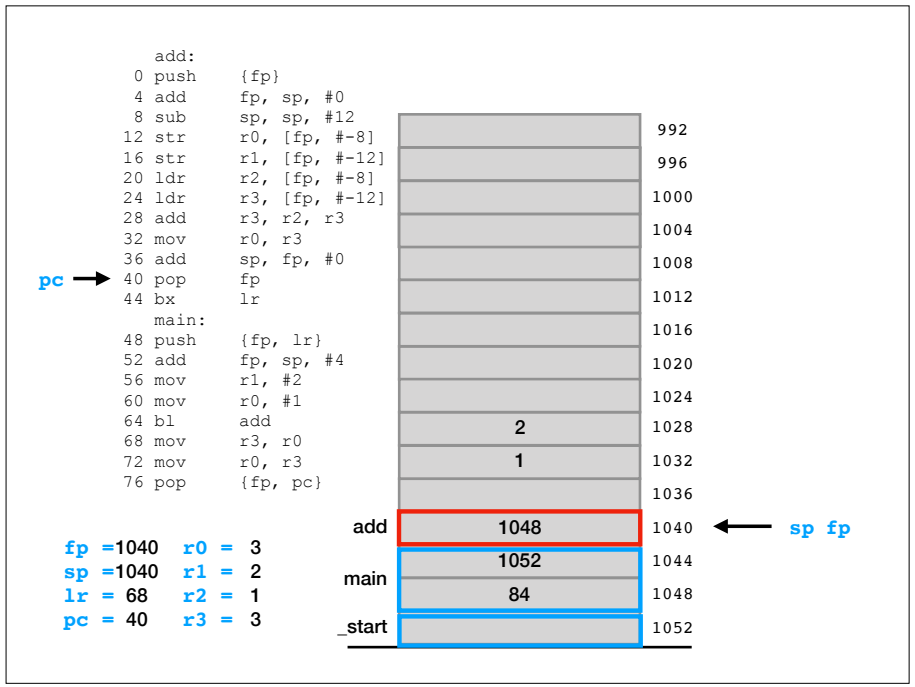

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 0
pc = 20 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
2
1
add
1048
1052
main
84
_start
sp
fp

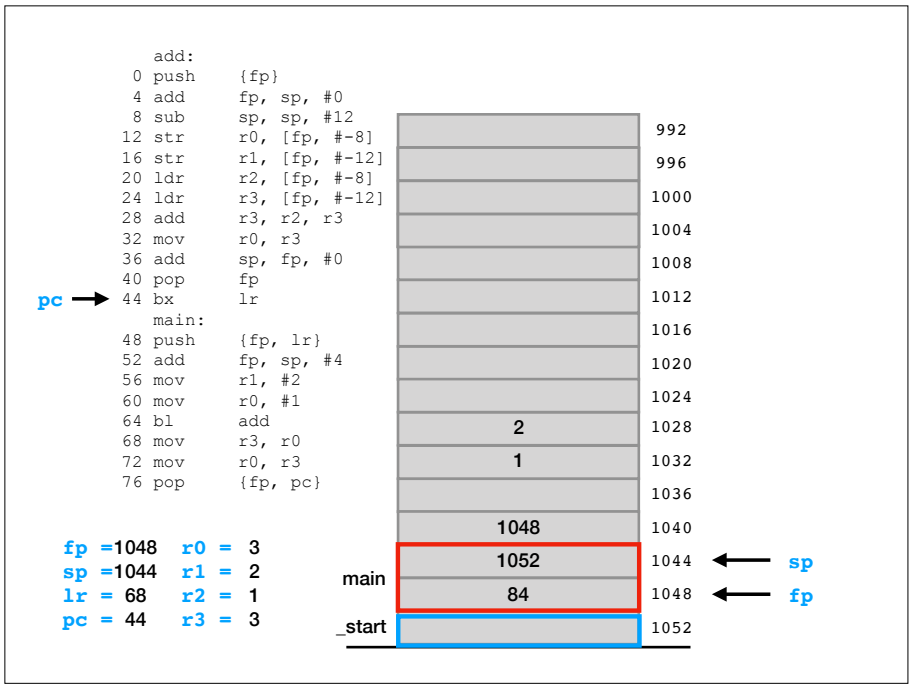

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 1
pc = 24 r3 = 0
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
2
1
add
1048
main
1052
84
_start
sp
fp

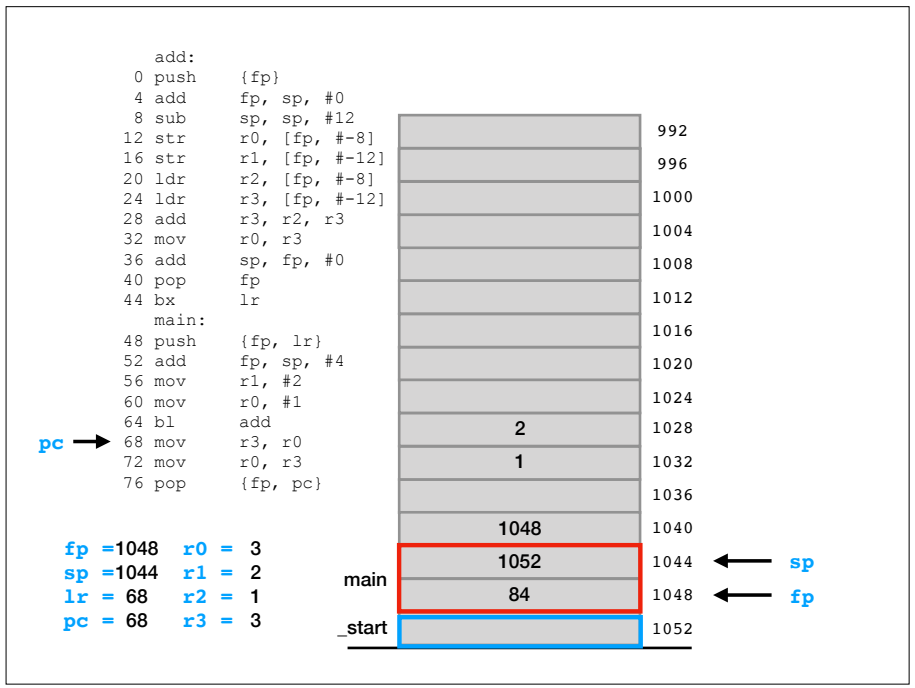

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 1
pc = 28 r3 = 2
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
2
1
add
1048
main
1052
84
_start
sp
fp

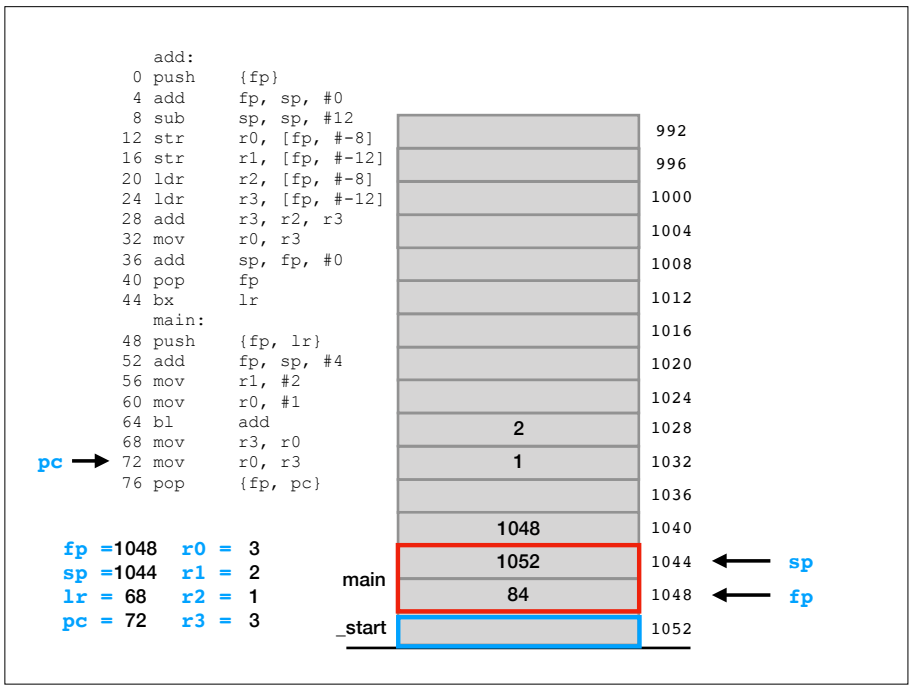

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 1
sp =1028 r1 = 2
lr = 68 r2 = 1
pc = 32 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
2
1
add
1048
main
1052
84
_start
sp
fp

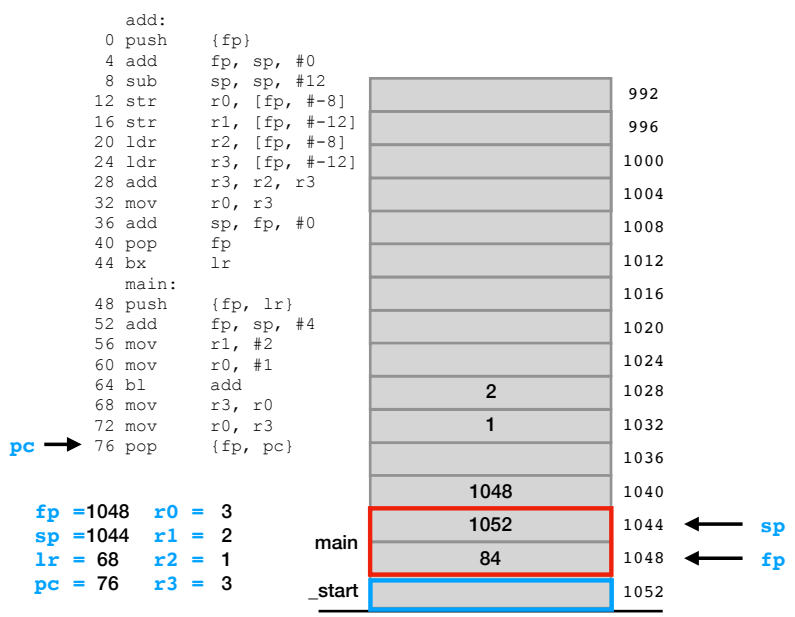

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 3
sp =1028 r1 = 2
lr = 68 r2 = 1
pc = 36 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044
1048
1052
2
1
add
1048
main
1052
84
_start
sp
fp

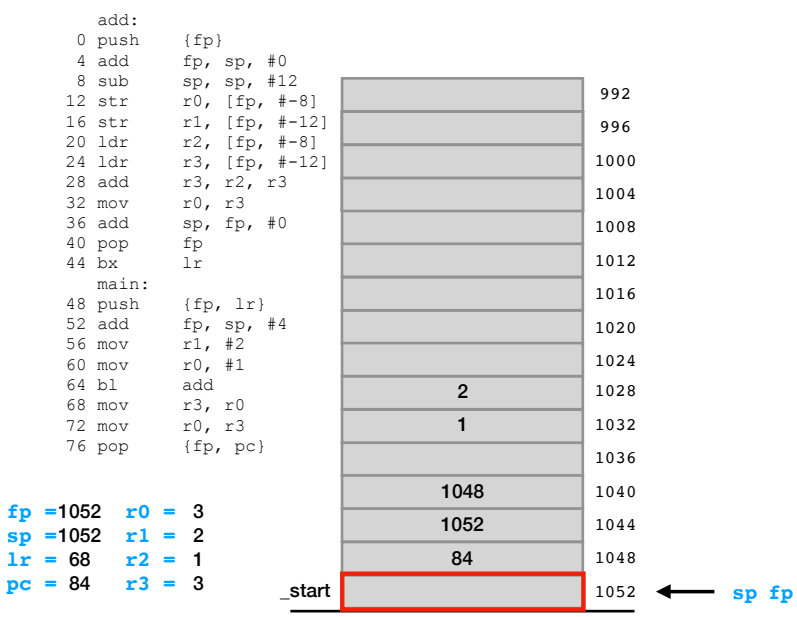

add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1040 r0 = 3
sp =1040 r1 = 2
lr = 68 r2 = 1
pc = 40 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
2
1
1028
1032
1036
add
1048
1040
sp fp
main
1052
84
1044
1048
_start
1052


add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1048 r0 = 3
sp =1044 r1 = 2
lr = 68 r2 = 1
pc = 44 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
2
1
1028
1032
1036
1048
1040
main
1052
sp
84
1044
1048
fp
_start
1052


add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1048 r0 = 3
sp =1044 r1 = 2
lr = 68 r2 = 1
pc = 68 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
2
1
1028
1032
1036
1048
1040
main
1052
sp
84
1044
1048
fp
_start
1052


add:
0 push {fp}
4 add fp, sp, #0
8 sub sp, sp, #12
12 str r0, [fp, #-8]
16 str r1, [fp, #-12]
20 ldr r2, [fp, #-8]
24 ldr r3, [fp, #-12]
28 add r3, r2, r3
32 mov r0, r3
36 add sp, fp, #0
40 pop fp
44 bx lr
main:
48 push {fp, lr}
52 add fp, sp, #4
56 mov r1, #2
60 mov r0, #1
64 bl add
68 mov r3, r0
72 mov r0, r3
76 pop {fp, pc}
pc
fp =1048 r0 = 3
sp =1044 r1 = 2
lr = 68 r2 = 1
pc = 72 r3 = 3
992
996
1000
1004
1008
1012
1016
1020
1024
2
1
1028
1032
1036
1048
1040
main
1052
sp
84
1044
1048
fp
_start
1052



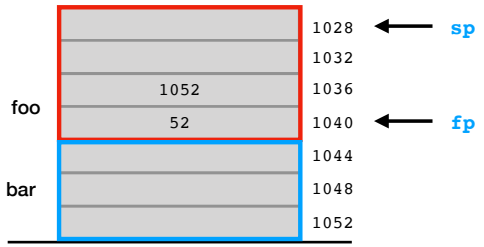


Buffer overflow exploits

Recall: preamble stores retaddr

- The callee's preamble stores the **`retaddr`** to the stack.
- E.g., **`foo`** stores the **`retaddr`** for **`bar`** at the location pointed to by **`fp`**.
- **`foo`** does this so that if it calls another function (e.g., **`printf`**), which would overwrite the **`retaddr`** in **`lr`**, it can just restore it from the stack.
- The epilogue restores the **`retaddr`** from the stack and then jumps to that address.
- E.g.,

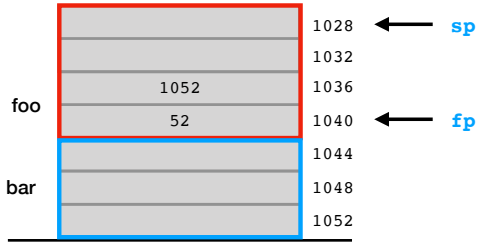
```
sub sp, fp, #4
pop {fp, pc}
```



Recall: preamble stores retaddr

- The callee's preamble stores the **`retaddr`** to the stack.
- E.g., **`foo`** stores the **`retaddr`** for **`bar`** at the location pointed to by **`fp`**.
- **`foo`** does this so that if it calls another function (e.g., **`printf`**), which would overwrite the **`retaddr`** in **`lr`**, it can just restore it from the stack.
- The epilogue restores the **`retaddr`** from the stack and then jumps to that address.
- E.g.,

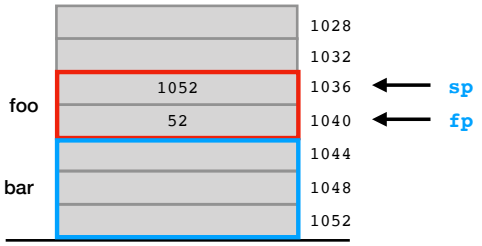
pc → **`sub sp, fp, #4`**
`pop {fp, pc}`



Recall: preamble stores retaddr

- The callee's preamble stores the **`retaddr`** to the stack.
- E.g., **`foo`** stores the **`retaddr`** for **`bar`** at the location pointed to by **`fp`**.
- **`foo`** does this so that if it calls another function (e.g., **`printf`**), which would overwrite the **`retaddr`** in **`lr`**, it can just restore it from the stack.
- The epilogue restores the **`retaddr`** from the stack and then jumps to that address.
- E.g.,

`sub sp, fp, #4`
pc → **`pop {fp, pc}`**

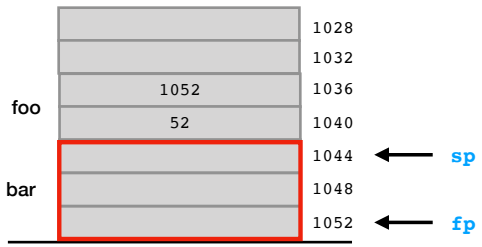


Recall: preamble stores retaddr

- The callee's preamble stores the **`retaddr`** to the stack.
- E.g., **`foo`** stores the **`retaddr`** for **`bar`** at the location pointed to by **`fp`**.
- **`foo`** does this so that if it calls another function (e.g., **`printf`**), which would overwrite the **`retaddr`** in **`lr`**, it can just restore it from the stack.
- The epilogue restores the **`retaddr`** from the stack and then jumps to that address.
- E.g.,

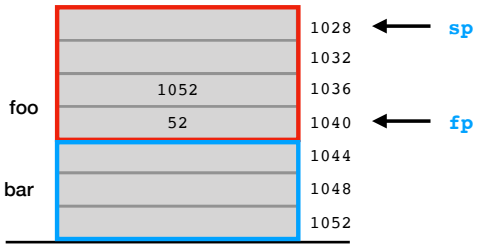
`sub sp, fp, #4`
`pop {fp, pc}`

pc = 52



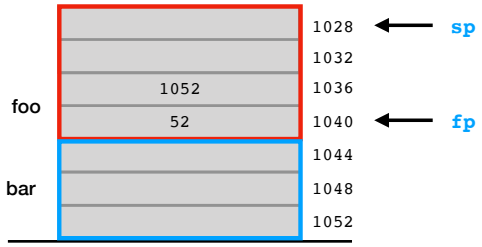
Buffer overflow exploit

- The goal of a buffer overflow exploit is to overflow a buffer such that we also corrupt the return address.
- Suppose an 8-byte buffer starts at 1028.
- If we write >8 bytes, values overflow into the parts of the stack that store control values.
- E.g., suppose we want to jump to a completely different function that happens to be at address 192.



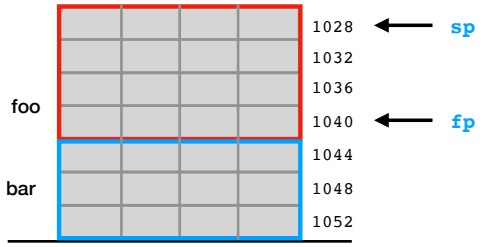
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- `1052 = 0x 00 00 04 1c`
- `52 = 0x 00 00 00 34`
- Remember that ARM is little-endian, so the little end is stored first.



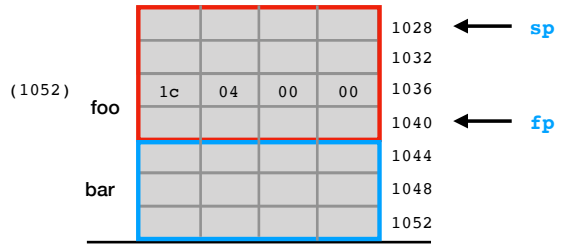
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- `1052 = 0x 00 00 04 1c`
- `52 = 0x 00 00 00 34`
- Remember that ARM is little-endian, so the little end is stored first.



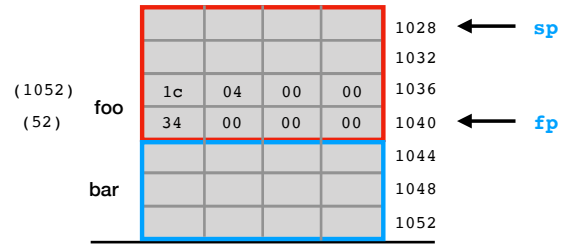
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- `1052 = 0x 00 00 04 1c`
- `52 = 0x 00 00 00 34`
- Remember that ARM is little-endian, so the little end is stored first.



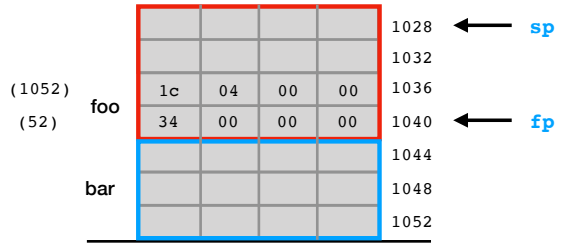
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- `1052 = 0x 00 00 04 1c`
- `52 = 0x 00 00 00 34`
- Remember that ARM is little-endian, so the little end is stored first.



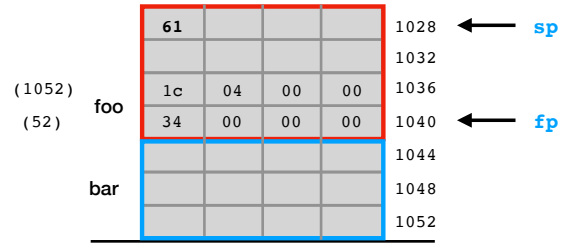
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



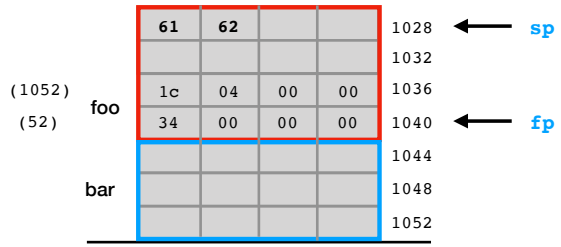
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



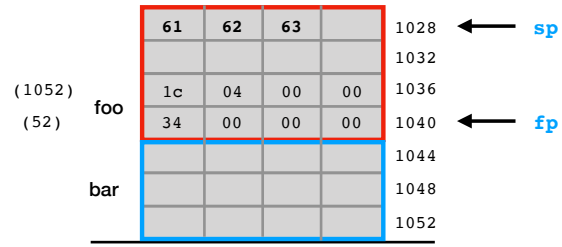
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



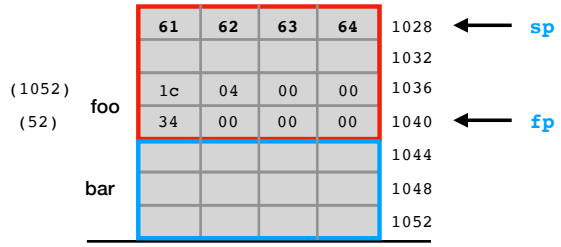
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



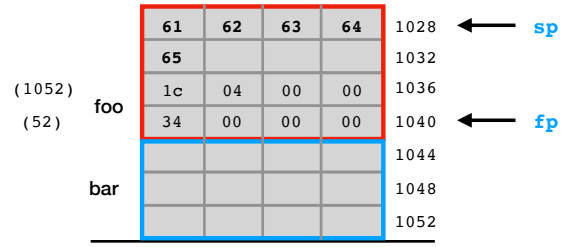
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?

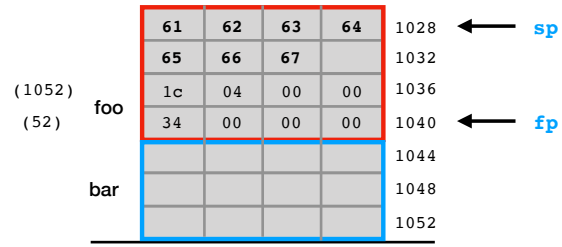


Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?

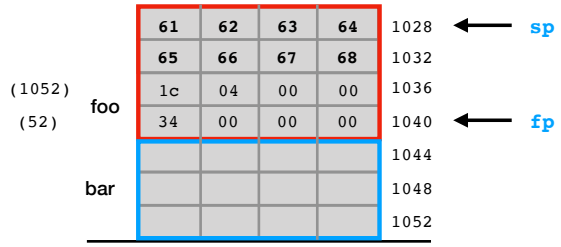
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



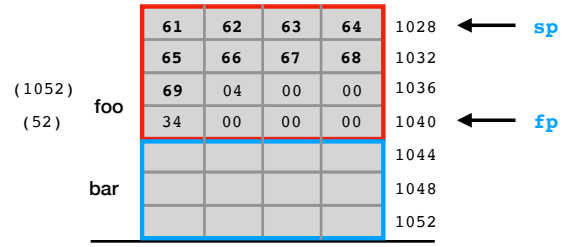
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



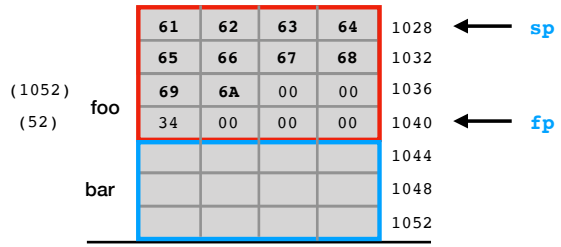
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



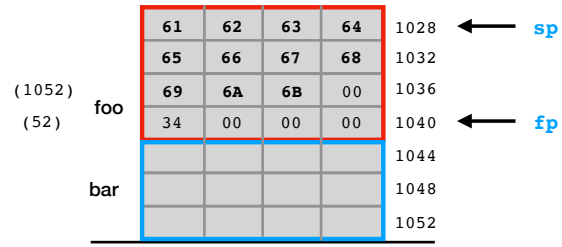
Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?

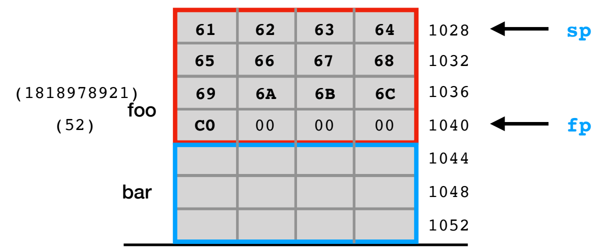


Buffer overflow exploit

- **192 = 0x 00 00 00 c0**
- We just need to write 16 bytes, ending with **0x000000c0**.
- How about **abcdefghijkl\xc0\x00\x00\x00**?

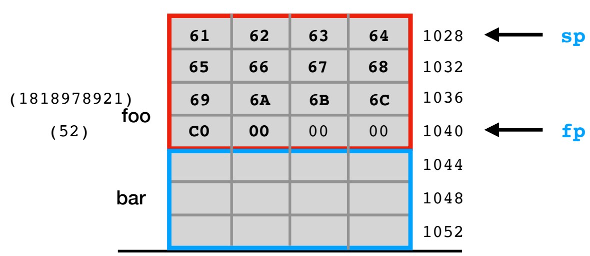
Buffer overflow exploit

- **192 = 0x 00 00 00 c0**
- We just need to write 16 bytes, ending with **0x000000c0**.
- How about **abcdefghijkl\xc0\x00\x00\x00**?



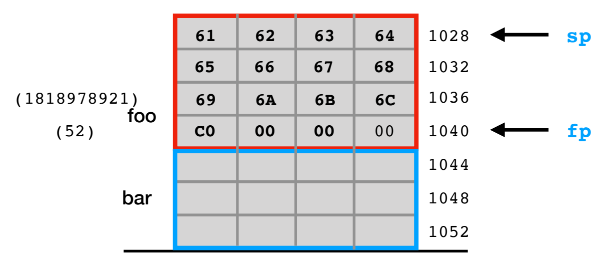
Buffer overflow exploit

- **192 = 0x 00 00 00 c0**
- We just need to write 16 bytes, ending with **0x000000c0**.
- How about **abcdefghijkl\xc0\x00\x00\x00**?



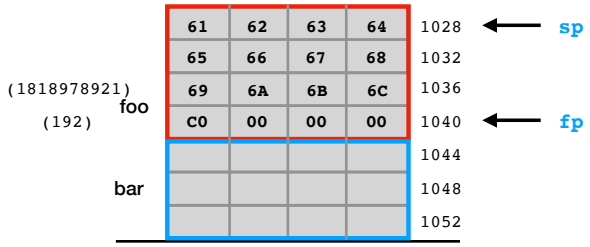
Buffer overflow exploit

- **192 = 0x 00 00 00 c0**
- We just need to write 16 bytes, ending with **0x000000c0**.
- How about **abcdefghijkl\xc0\x00\x00\x00**?



Buffer overflow exploit

- `192 = 0x 00 00 00 c0`
- We just need to write 16 bytes, ending with `0x000000c0`.
- How about `abcdefghijkl\xc0\x00\x00\x00`?



Buffer overflow exploit

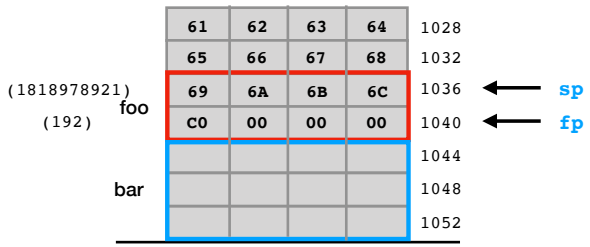
- Now when foo returns, it returns to the wrong place.

```
pc → sub sp, fp, #4
     pop   {fp, pc}
```

Buffer overflow exploit

- Now when foo returns, it returns to the wrong place.

```
     sub sp, fp, #4
pc → pop   {fp, pc}
```

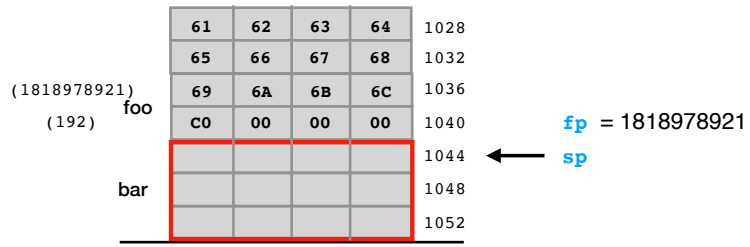


Buffer overflow exploit

- Now when foo returns, it returns to the wrong place.

```
     sub sp, fp, #4
     pop   {fp, pc}
```

pc = 192



Crafting inputs

globalthermonuclearwar.c

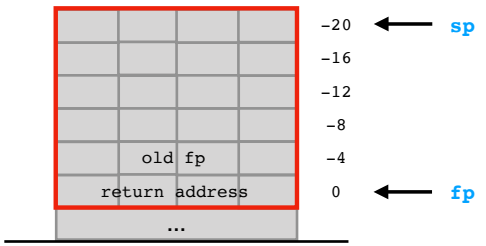
Remember this program?

globalthermonuclearwar.c

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

void    launch_missiles(int n) {
  printf("Launching %d missiles\n", n);
  // TODO: implement this function
}

void    authenticate_and_launch(void) {
  int n_missiles = 2;
  bool allowaccess = false;
  char response[8];

  printf("Secret: ");
  gets(response);

  if (strcmp(response, "Joshua") == 0)
    allowaccess = true;

  if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
  }

  if (!allowaccess)
    puts("Access denied");
}

int main(void) {
  puts("WarGames MissileLauncher v0.1");
  authenticate_and_launch();
  puts("Operation complete");
}
```

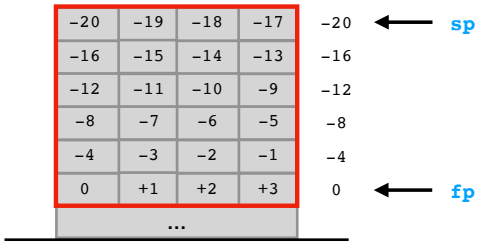
globalthermonuclearwar.c

authenticate_and_launch function



globalthermonuclearwar.c

authenticate_and_launch function



It is helpful to think about where the **target buffer** is relative to the **buffer you intend to overflow**.

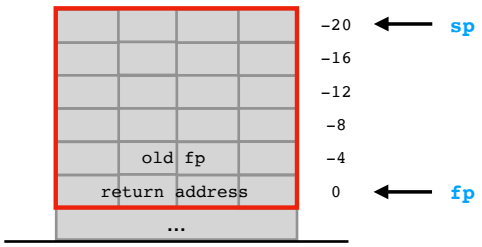
Using GDB to find locations of locals

globalthermonuclearwar.c

```
1  void    authenticate_and_launch(void) {
2    int n_missiles = 2;
3    bool allowaccess = false;
4    char response[8];
5
6    printf("Secret: ");
7    gets(response);
8
9    if (strcmp(response, "Joshua") == 0)
10     allowaccess = true;
11
12   if (allowaccess) {
13     puts("Access granted");
14     launch_missiles(n_missiles);
15   }
16
17   if (!allowaccess)
18     puts("Access denied");
19 }
```

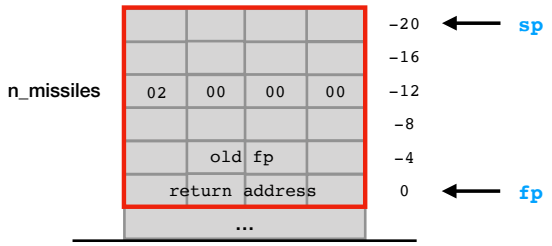
globalthermonuclearwar.c

authenticate_and_launch function



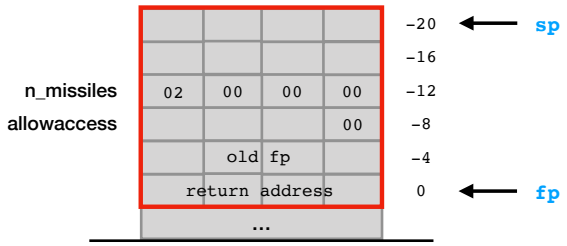
globalthermonuclearwar.c

authenticate_and_launch function



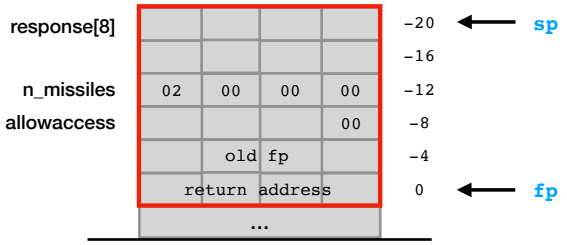
globalthermonuclearwar.c

authenticate_and_launch function



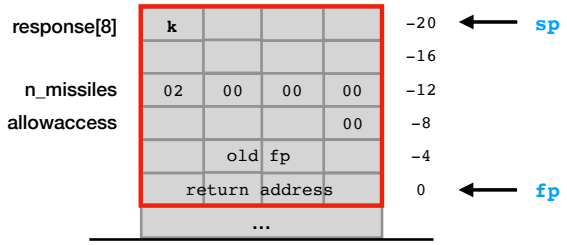
globalthermonuclearwar.c

authenticate_and_launch function



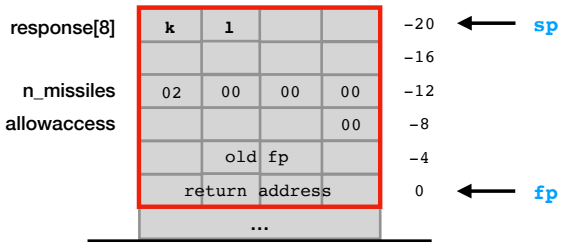
globalthermonuclearwar.c

authenticate_and_launch function



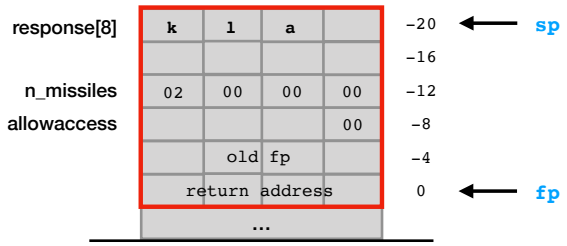
globalthermonuclearwar.c

authenticate_and_launch function



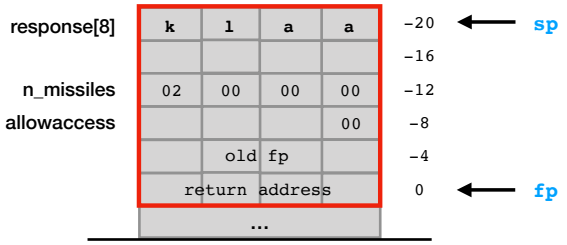
globalthermonuclearwar.c

authenticate_and_launch function



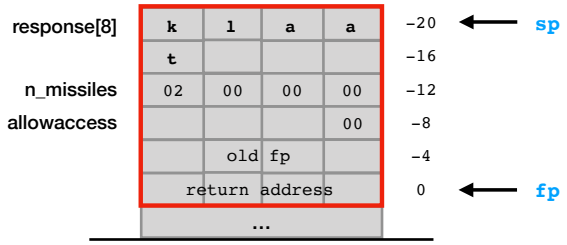
globalthermonuclearwar.c

authenticate_and_launch function



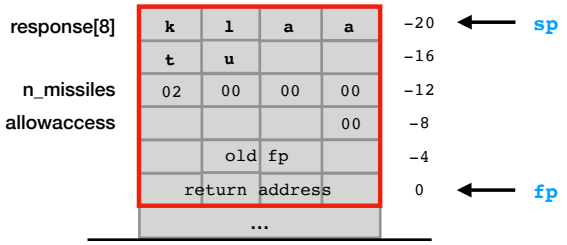
globalthermonuclearwar.c

authenticate_and_launch function



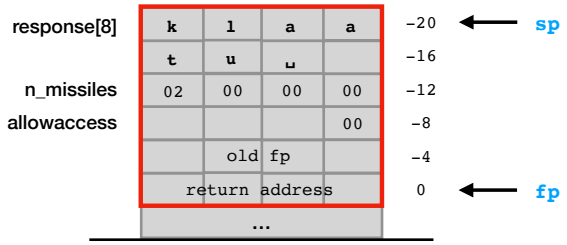
globalthermonuclearwar.c

authenticate_and_launch function



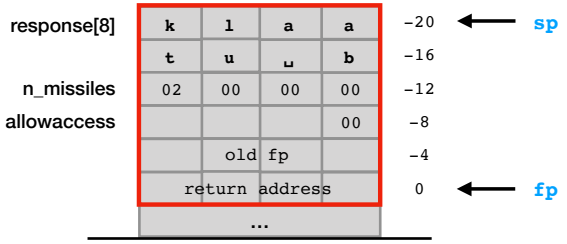
globalthermonuclearwar.c

authenticate_and_launch function



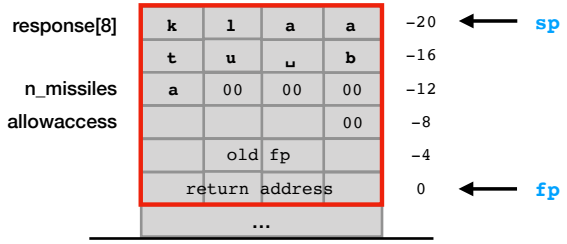
globalthermonuclearwar.c

authenticate_and_launch function



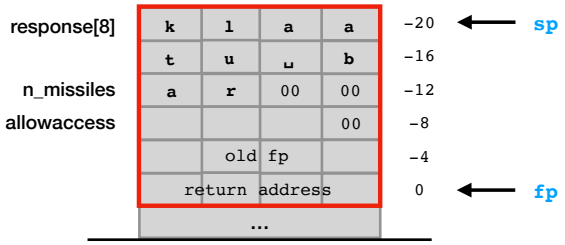
globalthermonuclearwar.c

authenticate_and_launch function



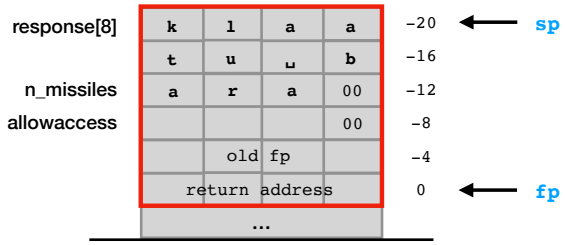
globalthermonuclearwar.c

authenticate_and_launch function



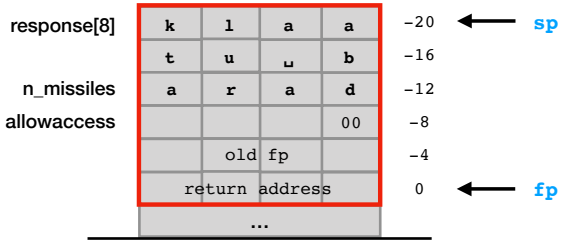
globalthermonuclearwar.c

authenticate_and_launch function



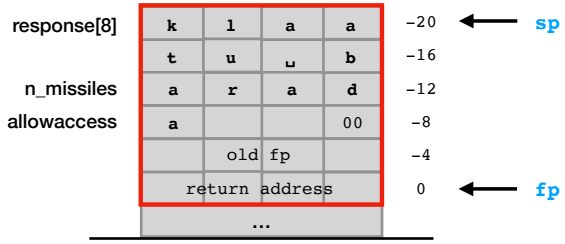
globalthermonuclearwar.c

authenticate_and_launch function



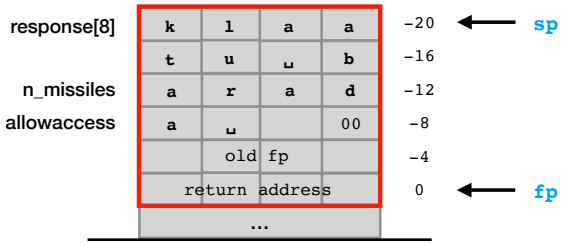
globalthermonuclearwar.c

authenticate_and_launch function



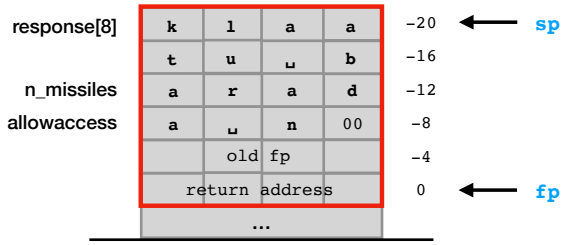
globalthermonuclearwar.c

authenticate_and_launch function



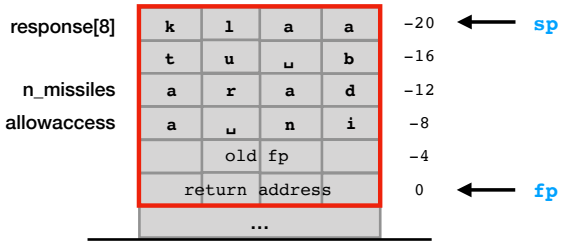
globalthermonuclearwar.c

authenticate_and_launch function



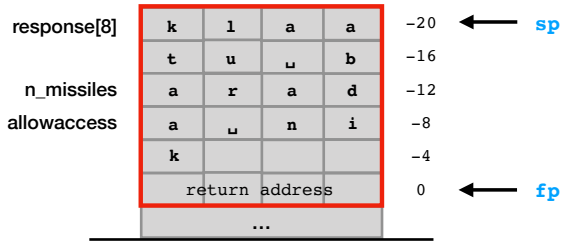
globalthermonuclearwar.c

authenticate_and_launch function



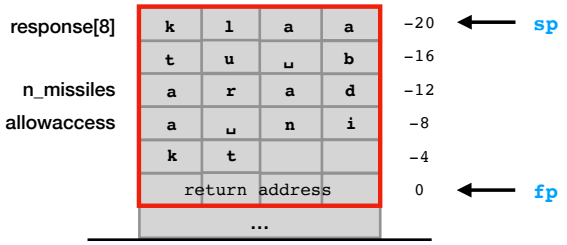
globalthermonuclearwar.c

authenticate_and_launch function



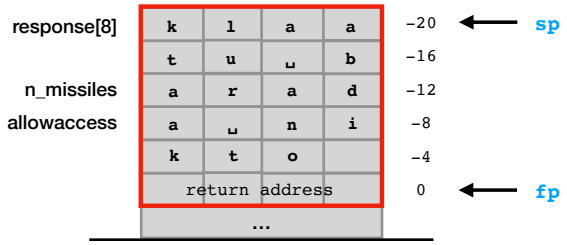
globalthermonuclearwar.c

authenticate_and_launch function



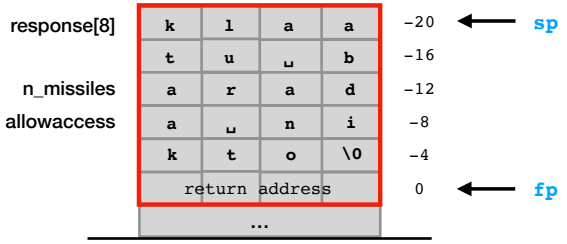
globalthermonuclearwar.c

authenticate_and_launch function



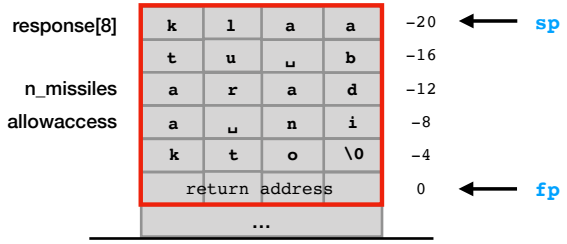
globalthermonuclearwar.c

authenticate_and_launch function



globalthermonuclearwar.c

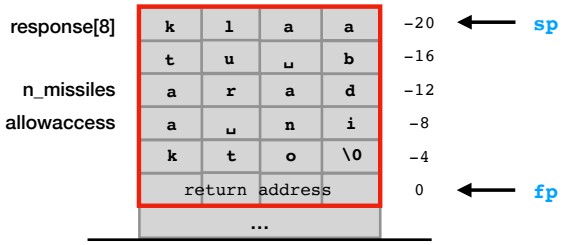
authenticate_and_launch function



What **value** is `allowaccess`? `0x69 >0 → true`

globalthermonuclearwar.c

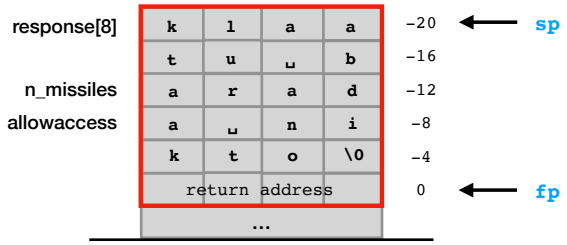
authenticate_and_launch function



What **number** is `n_missiles`?

globalthermonuclearwar.c

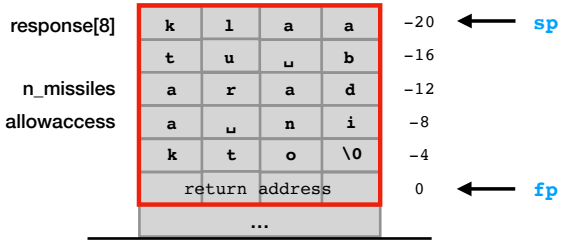
authenticate_and_launch function



What **number** is `n_missiles`? `0x64617261 = 1684107873`

globalthermonuclearwar.c

If I wanted the program to jump to `launch_missiles` by overwriting the return address, what kind of input would I need to give it?



Assume the address of `launch_missiles` is 0x10498.

Recap & Next Class

Today we learned:

Crafting inputs

Next class:

Shellcode