EECS 388

Introduction to Computer Security

Lecture 16:

Control Hijacking (Part 2)

October 24, 2024

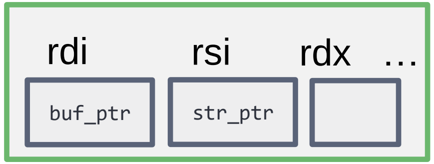
Aidan Delwiche

champ gives the craziest side eye…

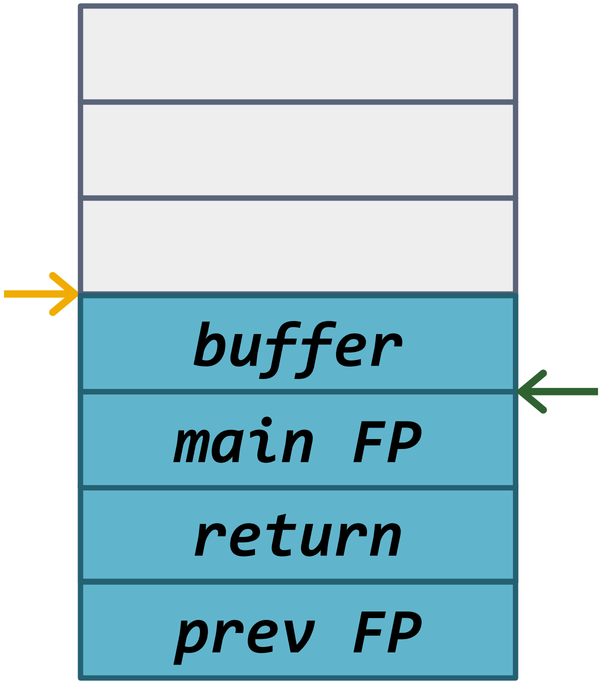
example.s (x64)

```
void foo(char *str) {
  char buffer[4];
  strcpy(buffer, str);
}
```

```
foo:
   push   rbp
   mov    rbp, rsp
   sub    rsp, 4
   mov    rsi, rdi
   lea    rdi,[rbp-4]
   call   strcpy
   leave
   ret
```



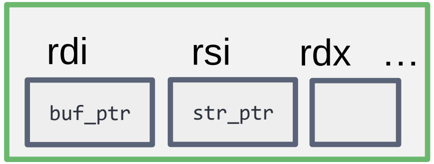
str_ptr: “AAAABBBBBBBB1234567”



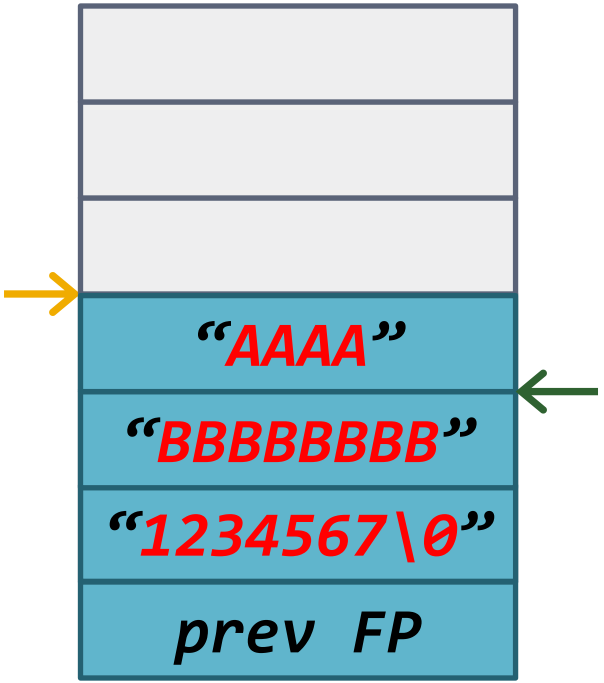
example.s (x64)

```
void foo(char *str) {
  char buffer[4];
  strcpy(buffer, str);
}
```

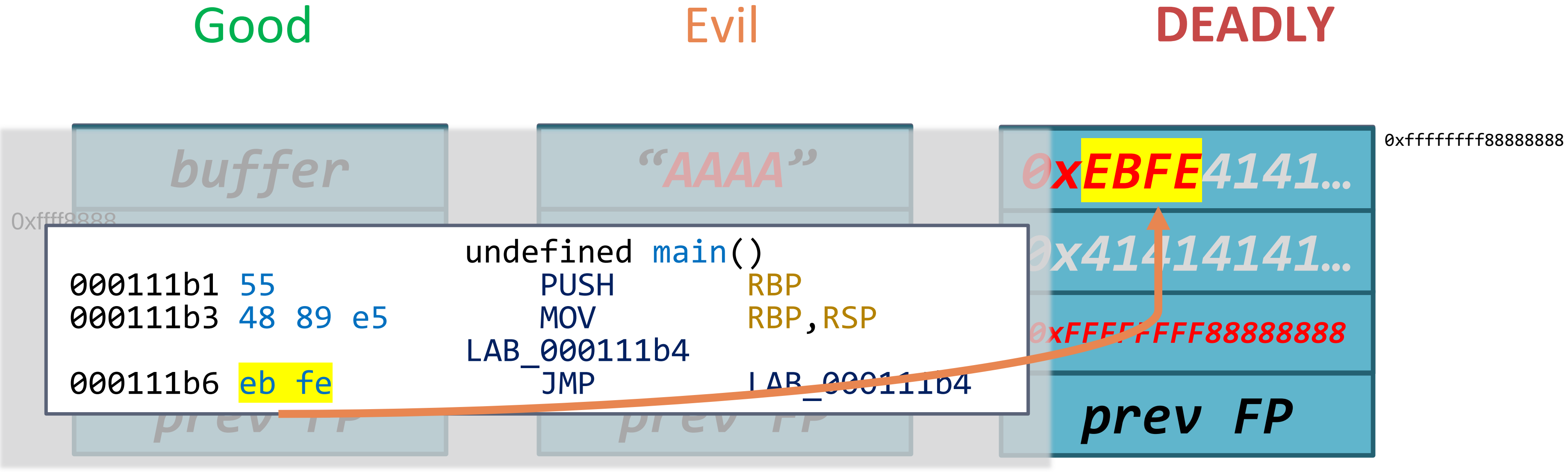
```
foo:
   push   rbp
   mov    rbp, rsp
   sub    rsp, 4
   mov    rsi, rdi
   lea    rdi,[rbp-4]
   call   strcpy
   leave
   ret
```



str_ptr: “AAAABBBBBBBB1234567”



Stack Shellcode



DEP

Write XOR Execute

- Write (heap/stack)
- Execute (code segments)
- Never both!

Data-only attacks

```
int delete_account(char* username,
  int length, VOID* creds) {
      int admin;
      char name[100];
      admin = check_admin(creds);
      strncpy(name, username, length);
      canonicalize_username(name);
      if (admin) {delete_user(name);}
      return (admin > 0);
}
```

Cat-and-Mouse Exploitation

Buffer Overflow

Stack Shellcode

DEP

Data-only attacks

Return-to-libc

Return to libc

Defender:

DEP prevents executing injected shellcode

Attacker:

Reuse code that already exists

Already marked as executable!

Recall… Calling Convention

Caller:

Push arguments to registers **`RDI`**, **`RSI`**, **`RDX`**, **`RCX`**, **`R8`**, **`R9`**

Push **`rip`**

Jump

Callee:

Push **`rbp`**

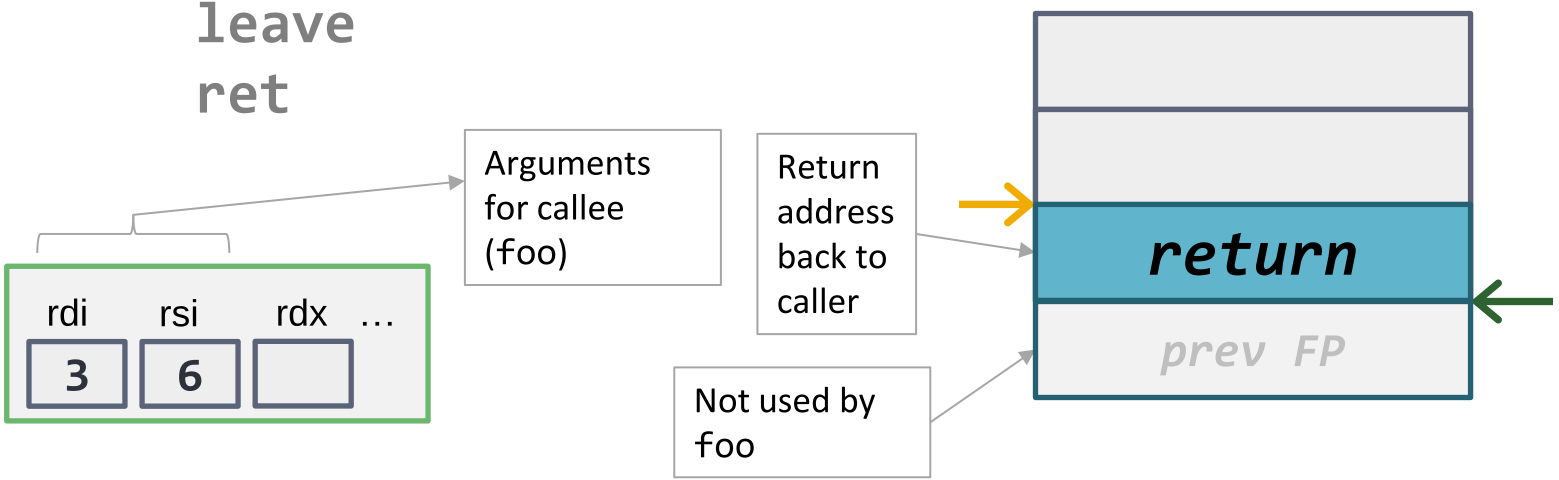
Move **rbp** to **`rsp`**

Stack after entering a new function

```
foo:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 0x10
    leave
    ret
```

Stack after entering a new function

```
foo:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 0x10
    leave
    ret
```



Stack after entering a new function

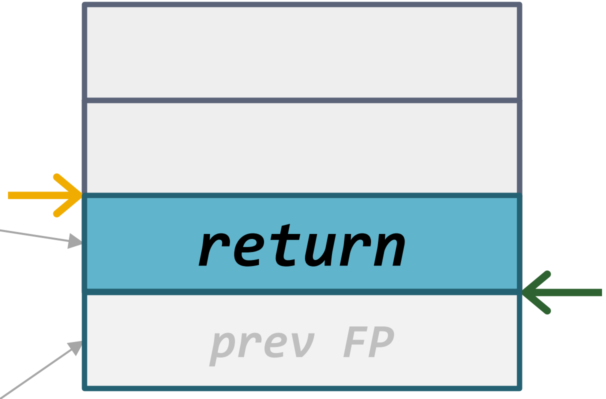
(some function that already exists)

```
ATTACK:
    push  rbp
    mov   rbp, rsp
    sub   rsp, 0x10
    leave
    ret
```

Arguments for callee (ATTACK)

Return address back to caller

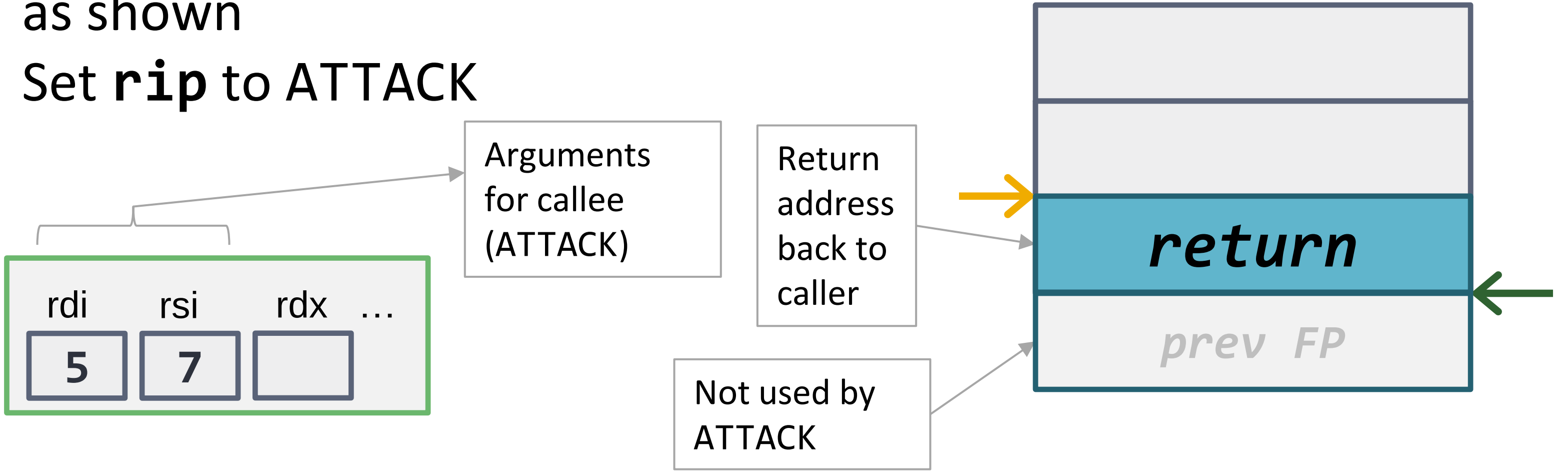
Not used by ATTACK



Stack after entering a new function

Our goal:
Execute ATTACK using a `ret` instead of a `call`
"Return-to-ATTACK"

How?
1. Set stack, **`rsp`**, and argument registers as shown
2. Set **`rip`** to ATTACK



Stack after entering a new function

Our goal:

Execute ATTACK using a `ret` instead of a `call`

"Return-to-ATTACK"

How?

0. Enter vulnerable function
1. Set stack, **`rsp`**, and argument registers as shown
2. Set **`rip`** to ATTACK

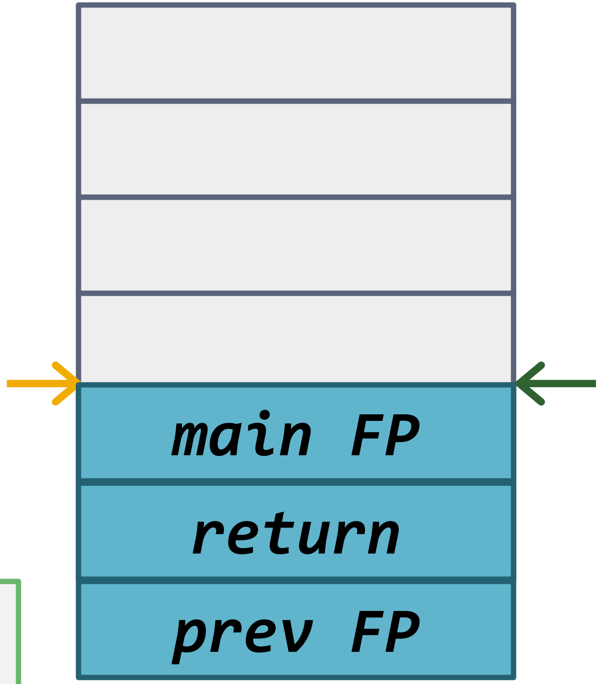
```
void foo(int a, int b) {
    char buf1[16];
}

void vulnerable() {
  int i = 3;
  int j = 6;
  char buffer[8];
  gets(buffer);
  foo(i, j);
}
```

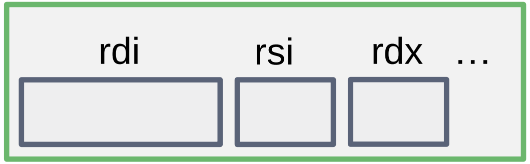
Attack surface

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp – 16]
  call  gets
  mov   rsi, [rbp – 8]
  mov   rdi, [rbp – 4]
  call  foo
  leave
  ret
```

Attack surface

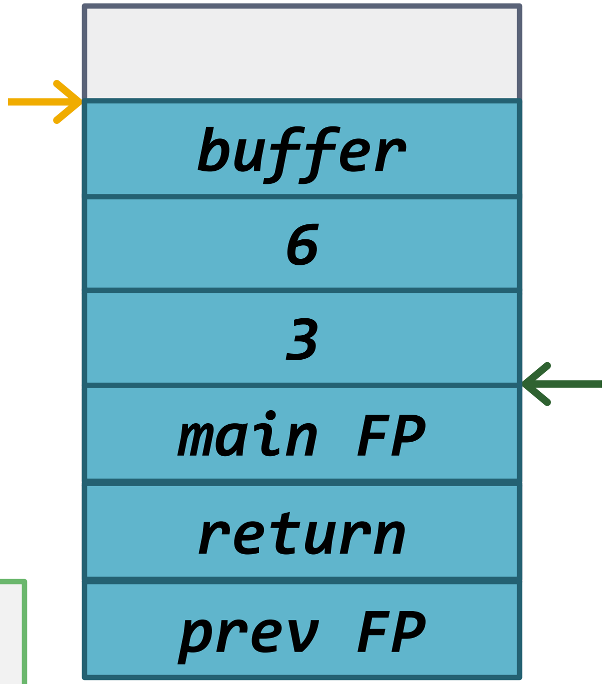
```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp - 16]
  call  gets
  mov   rsi, [rbp - 8]
  mov   rdi, [rbp - 4]
  call  foo
  leave
  ret
```





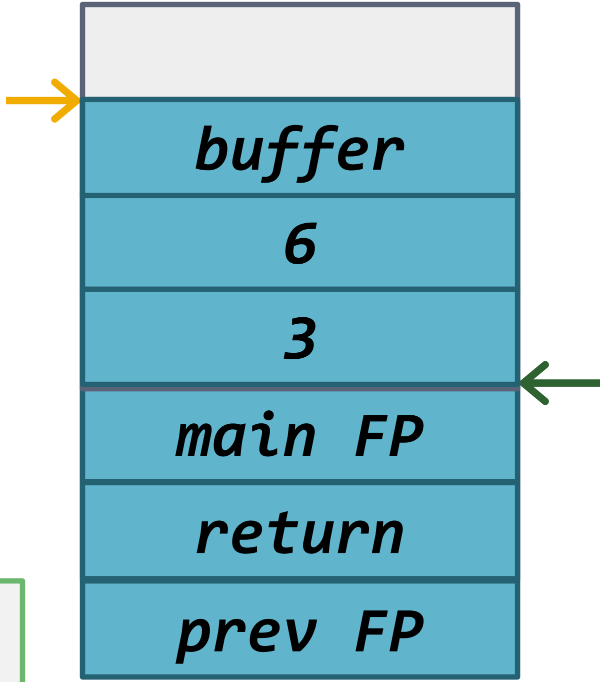
Attack surface

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp – 16]
  call  gets
  mov   rsi, [rbp – 8]
  mov   rdi, [rbp – 4]
  call  foo
  leave
  ret
```



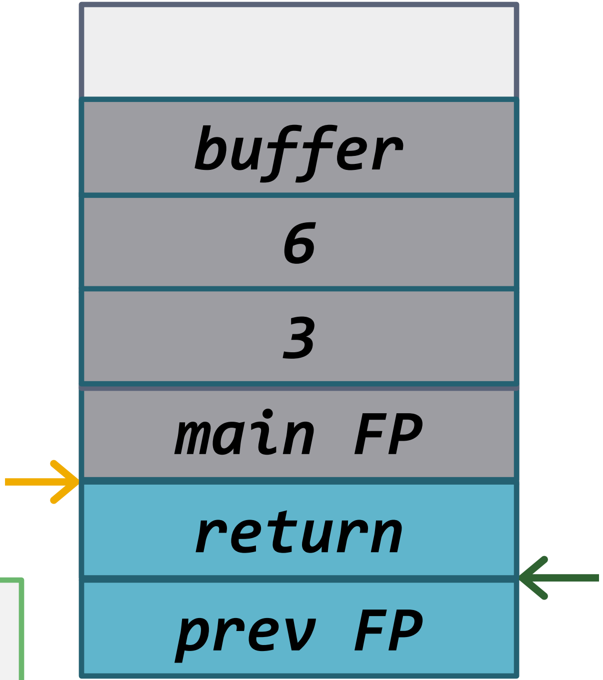
Attack surface

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp - 16]
  call  gets
  mov   rsi, [rbp - 8]
  mov   rdi, [rbp - 4]
  call  foo
  leave
  ret
```



Attack surface

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp – 16]
  call  gets
  mov   rsi, [rbp – 8]
  mov   rdi, [rbp – 4]
  call  foo
  leave
  ret
```

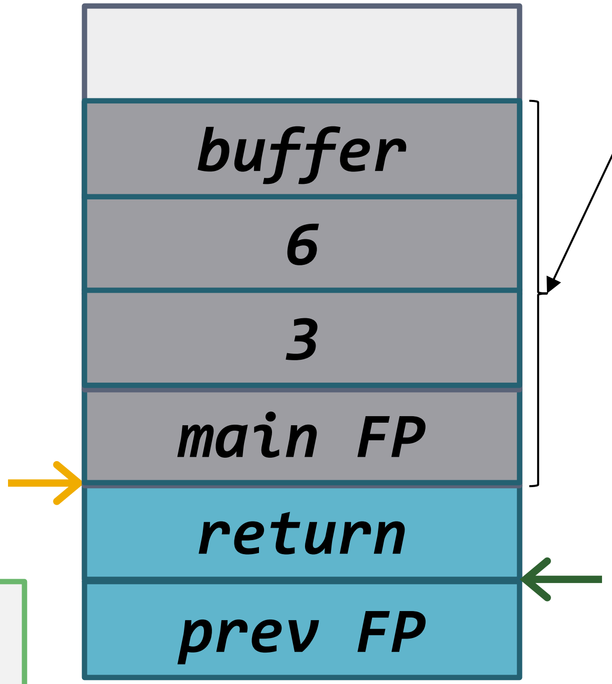


Attack surface

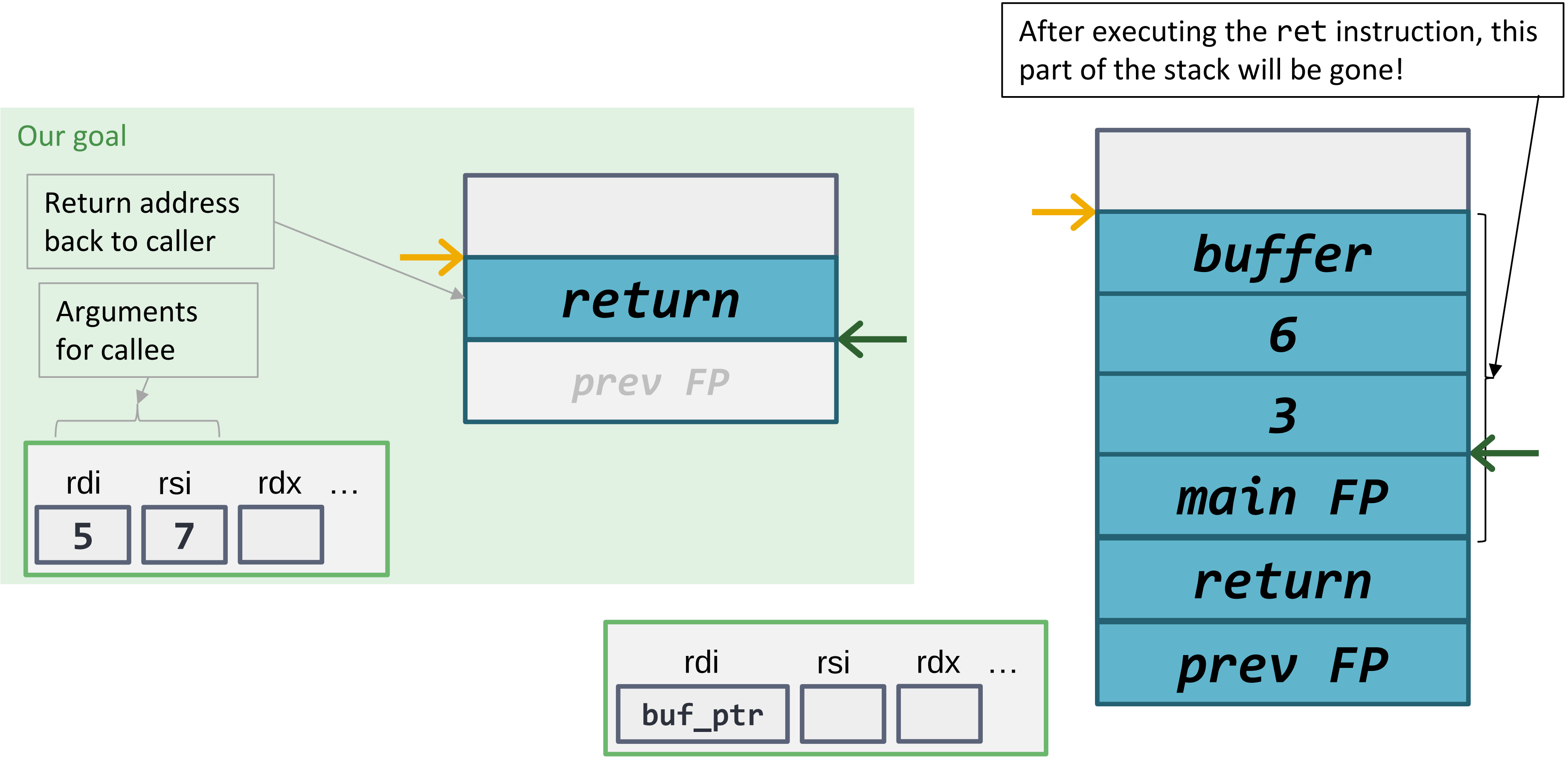
```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp - 16]
  call  gets
  mov   rsi, [rbp - 8]
  mov   rdi, [rbp - 4]
  call  foo
  leave
  ret
```

```
pop  rip
```

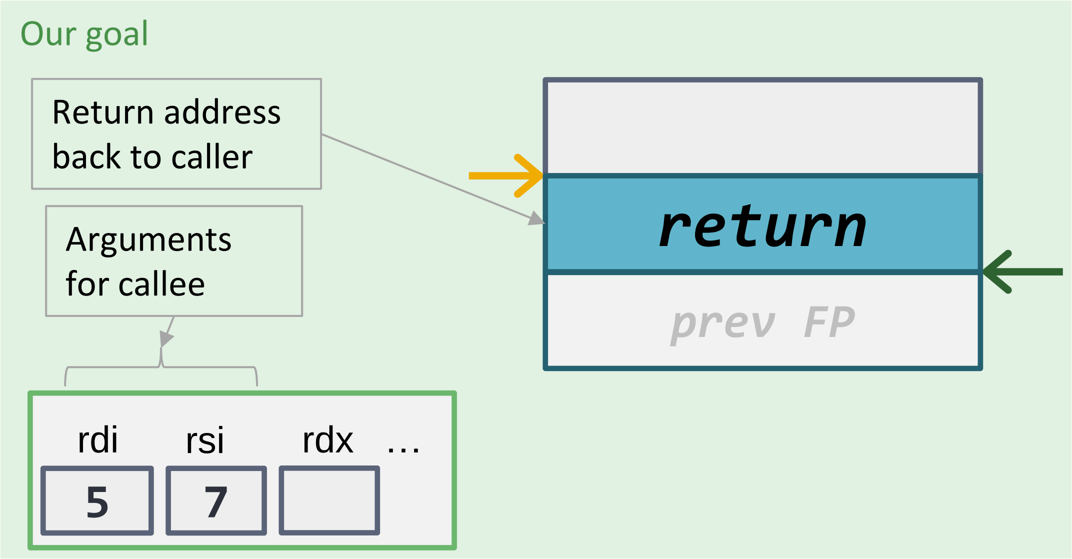
After executing the ret instruction, this part of the stack will be gone!



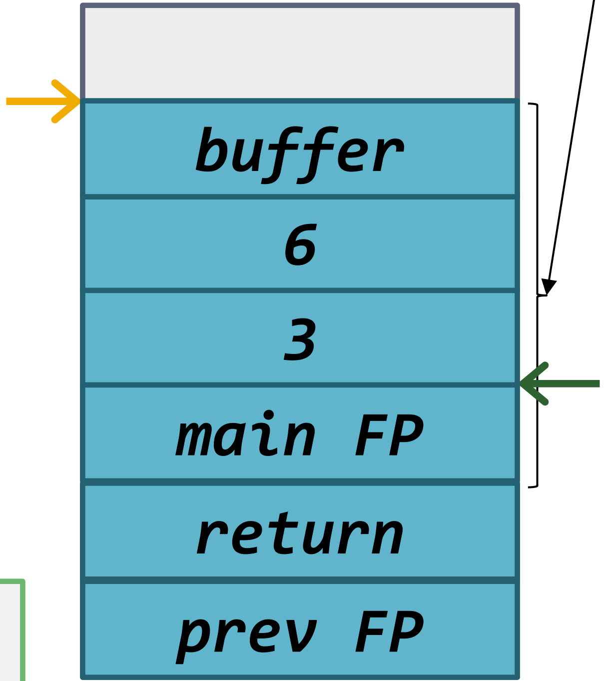
Rewind to before gets call



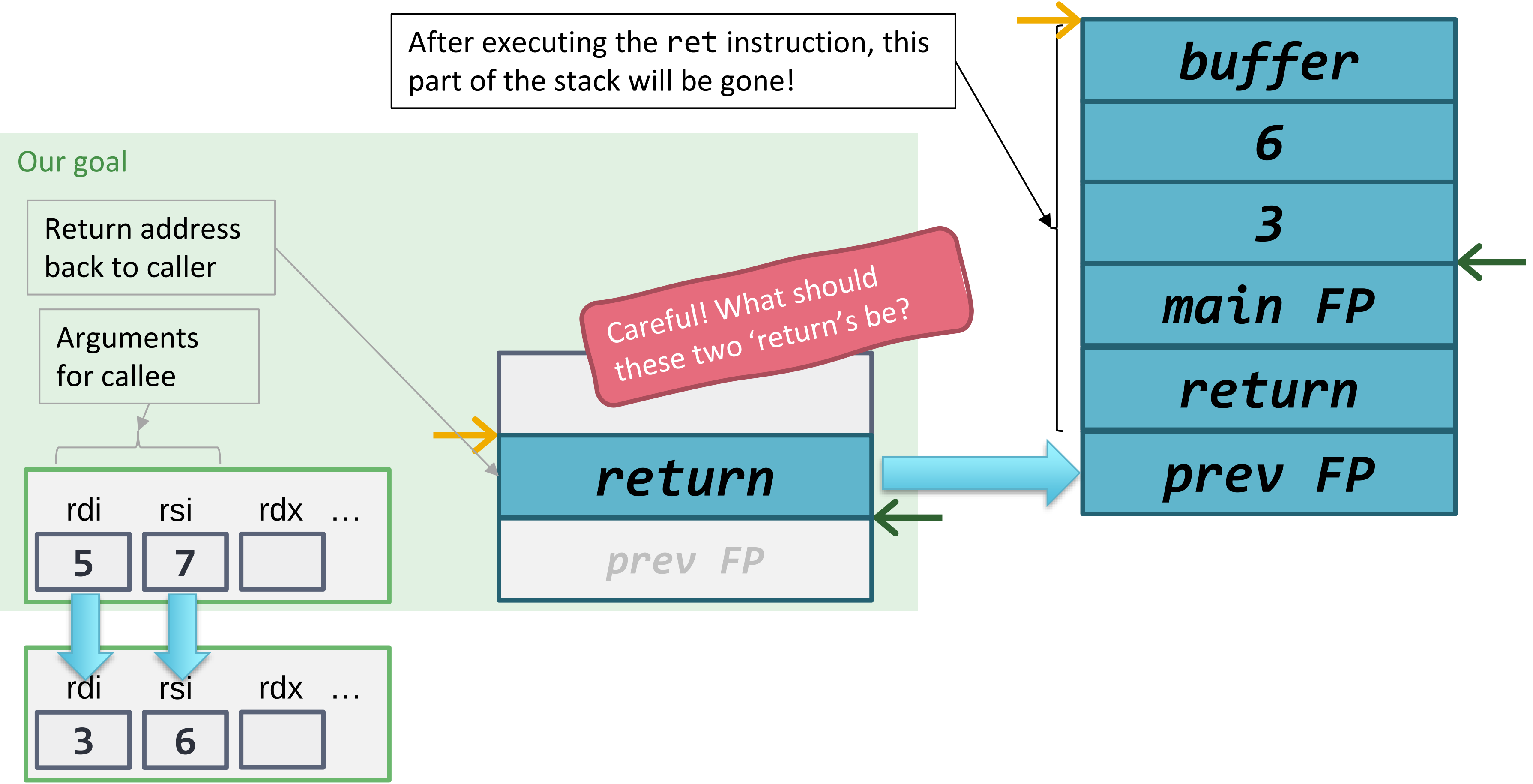
Rewind to before foo call



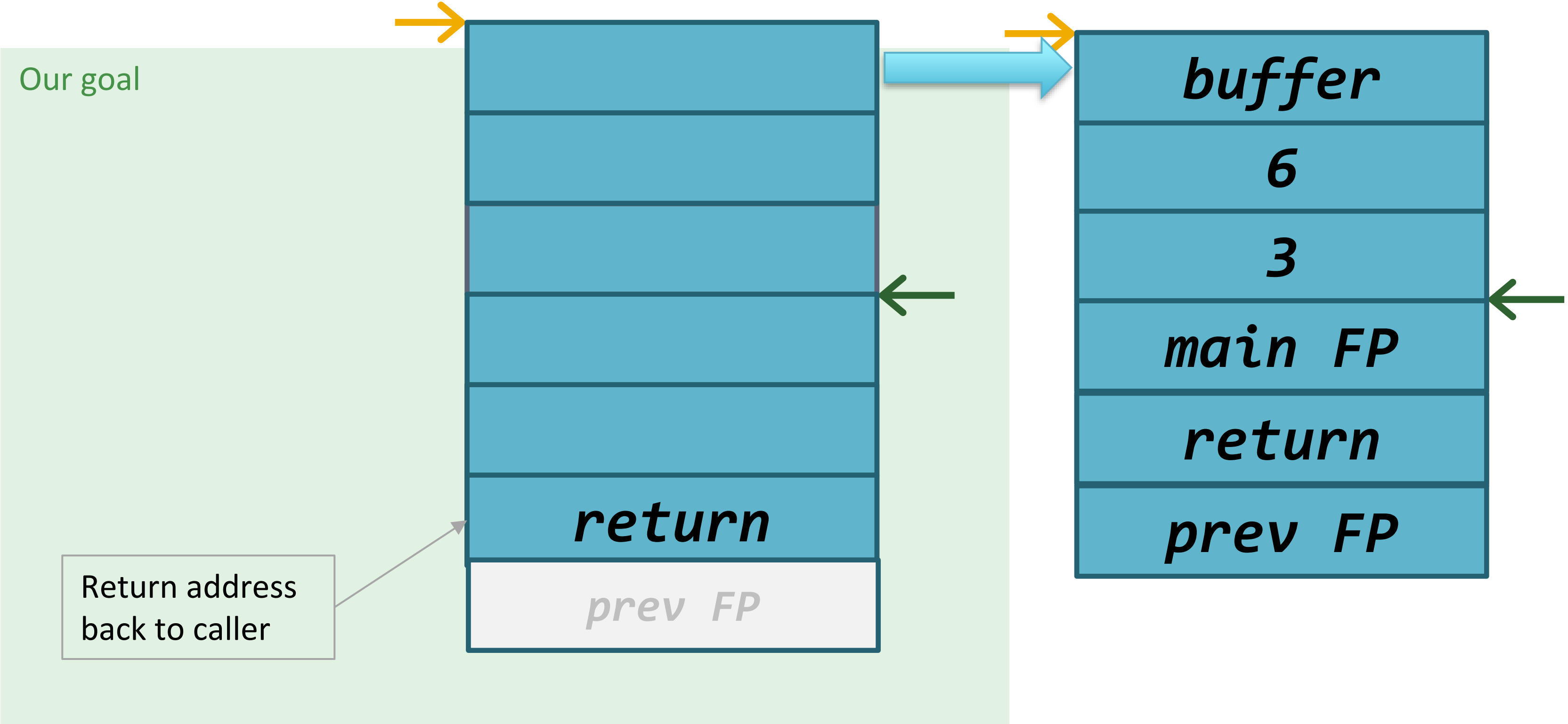
After executing the ret instruction, this part of the stack will be gone!



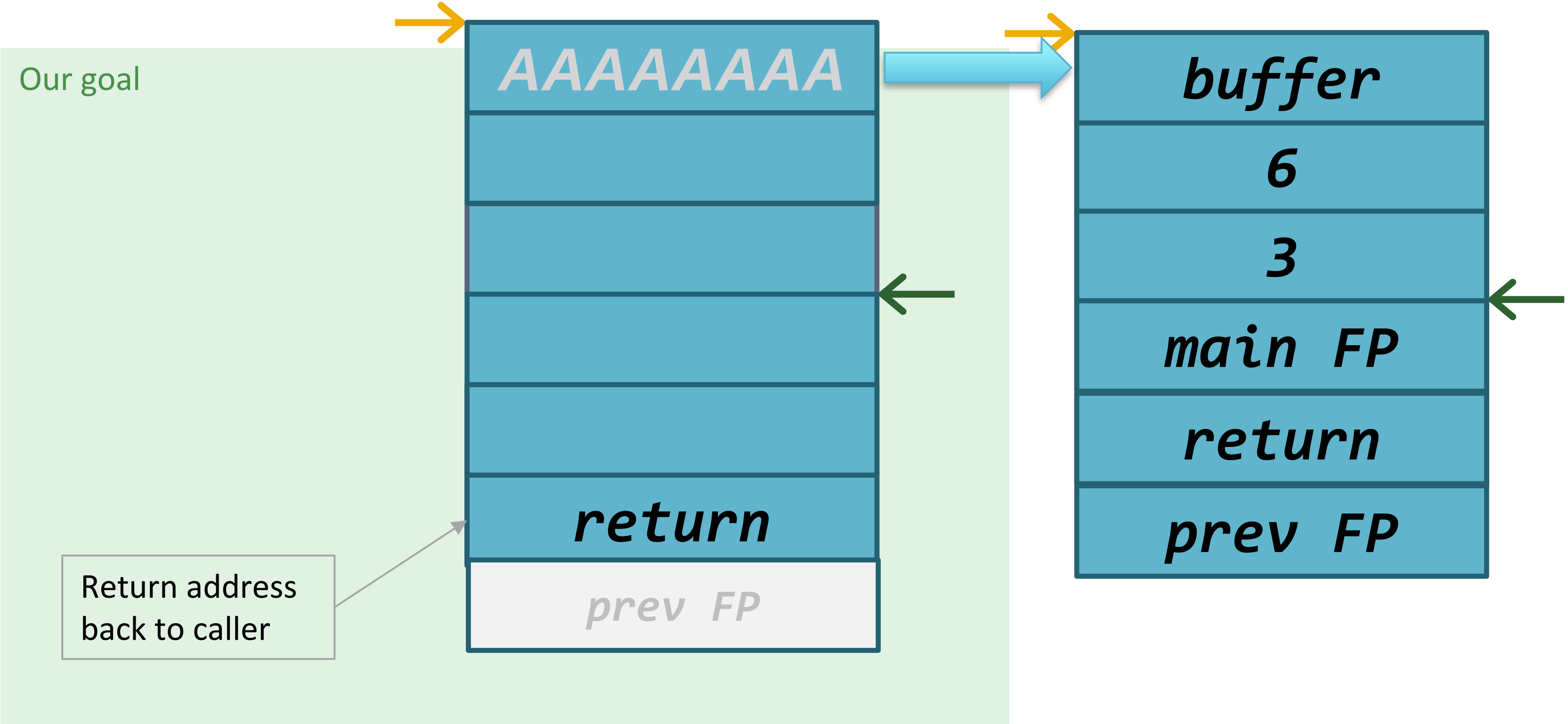
On your marks, get set, …



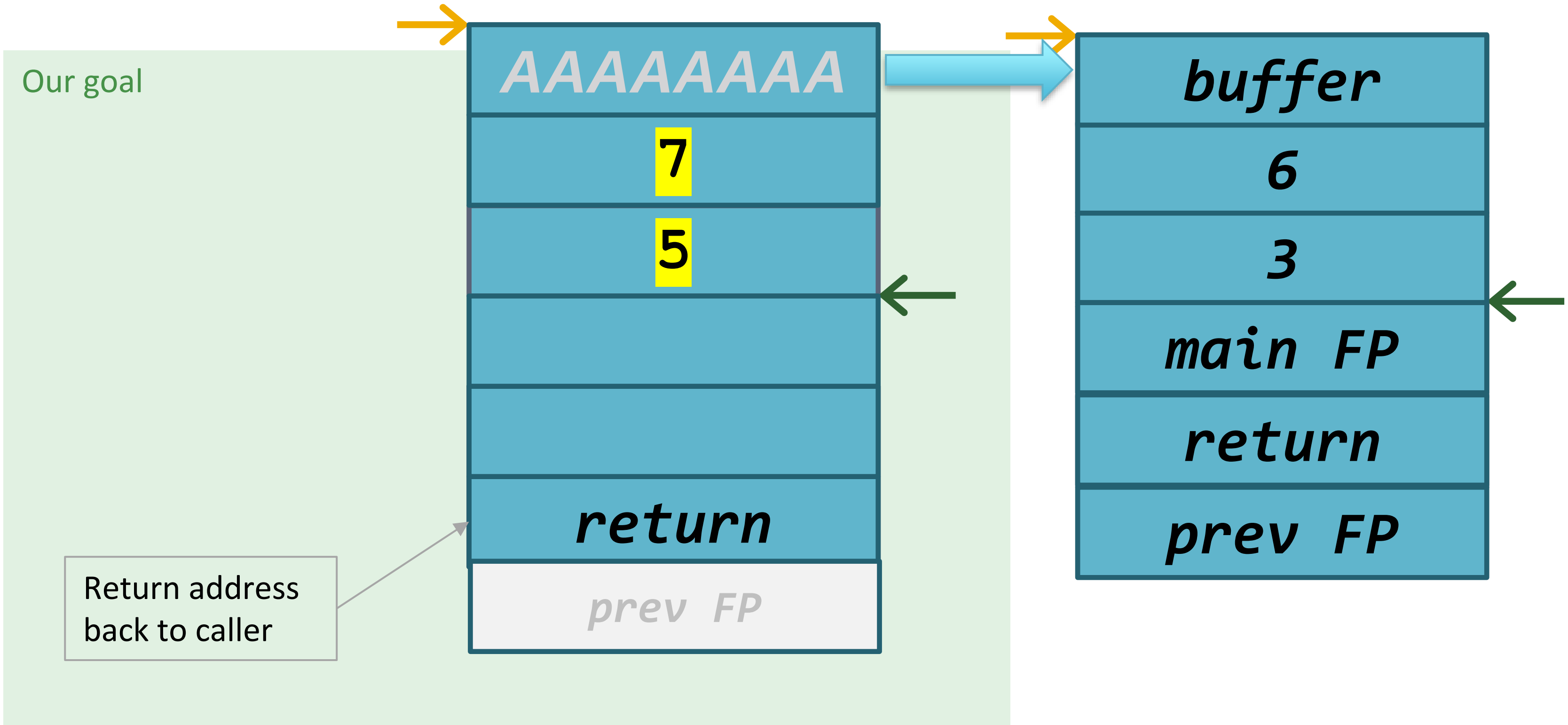
Attacker 2.0 – during gets call



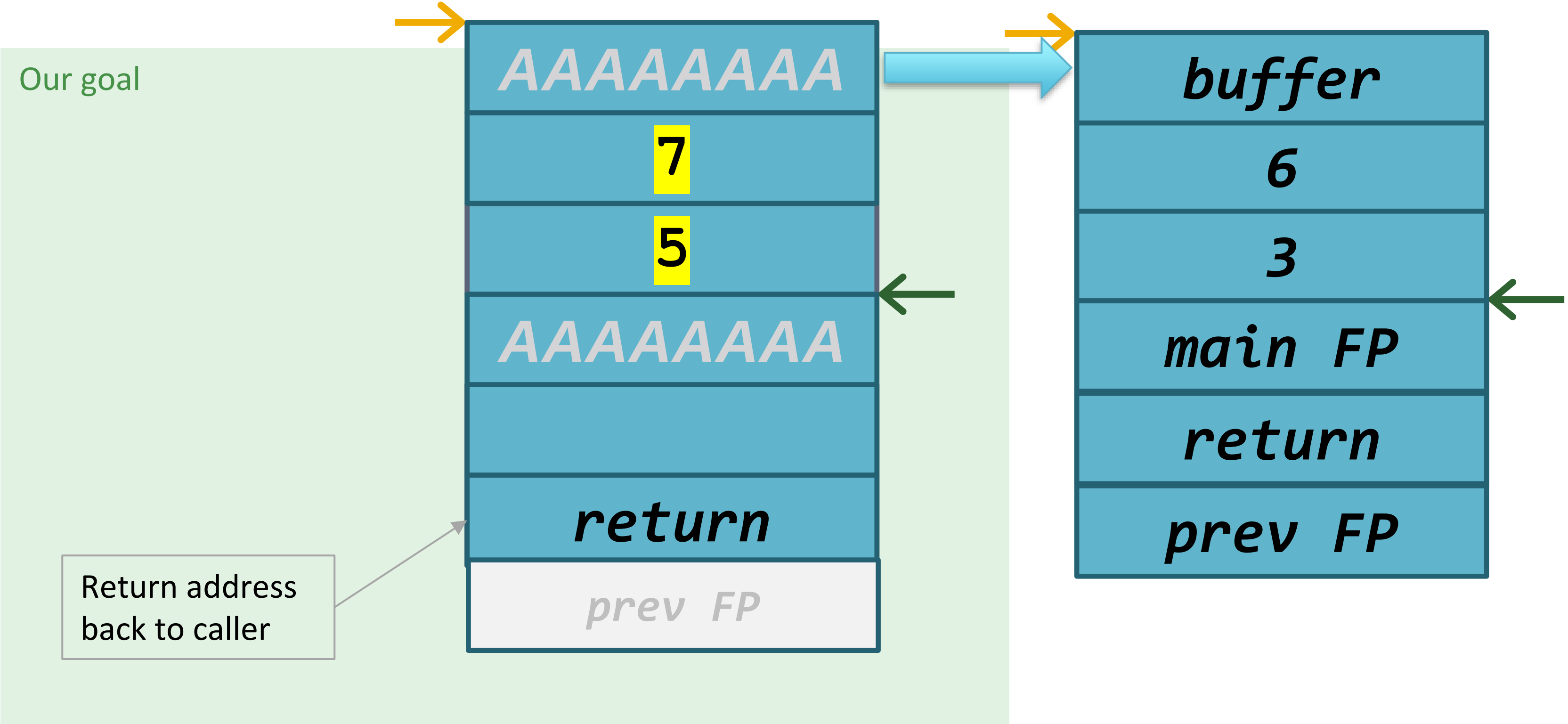
Attacker 2.0 – during gets call



Attacker 2.0 – during gets call

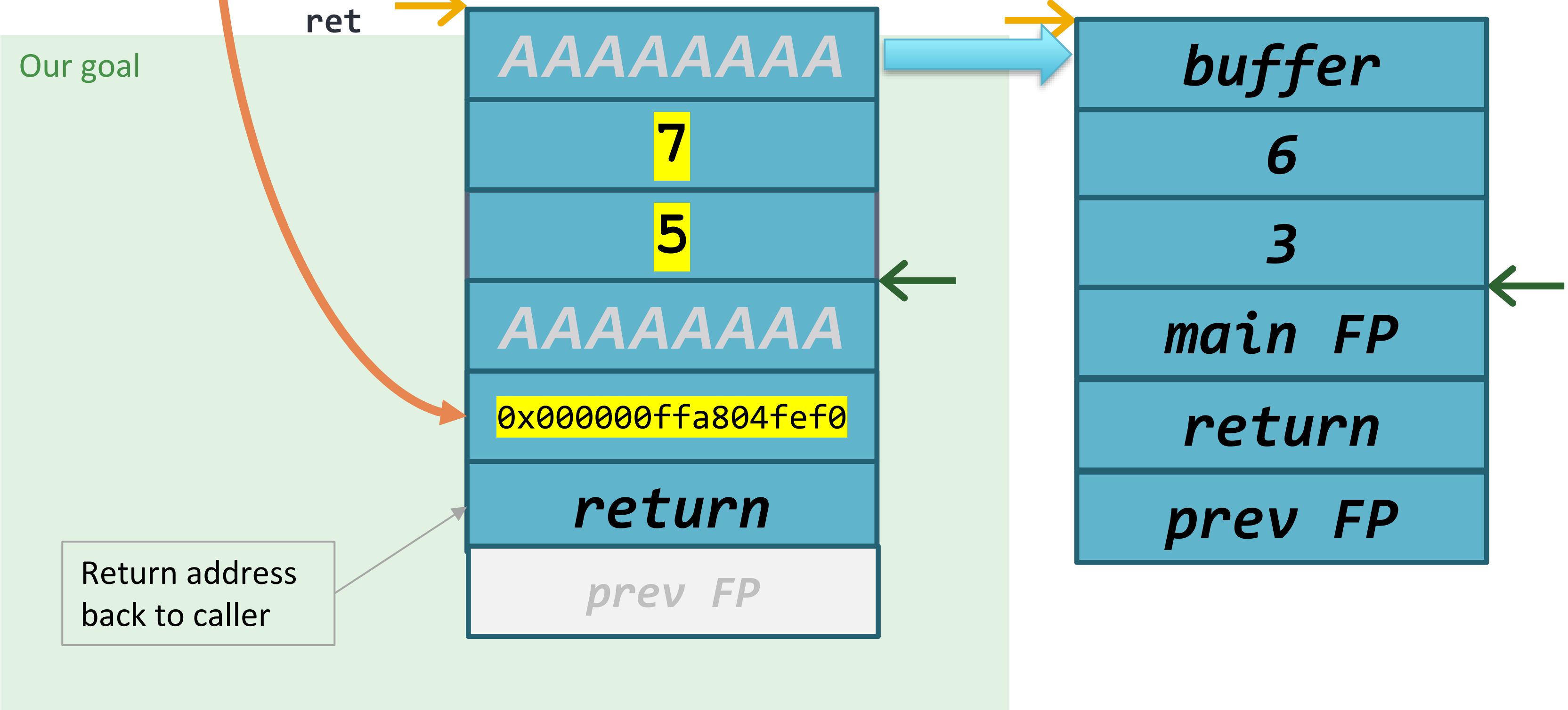


Attacker 2.0 – during gets call



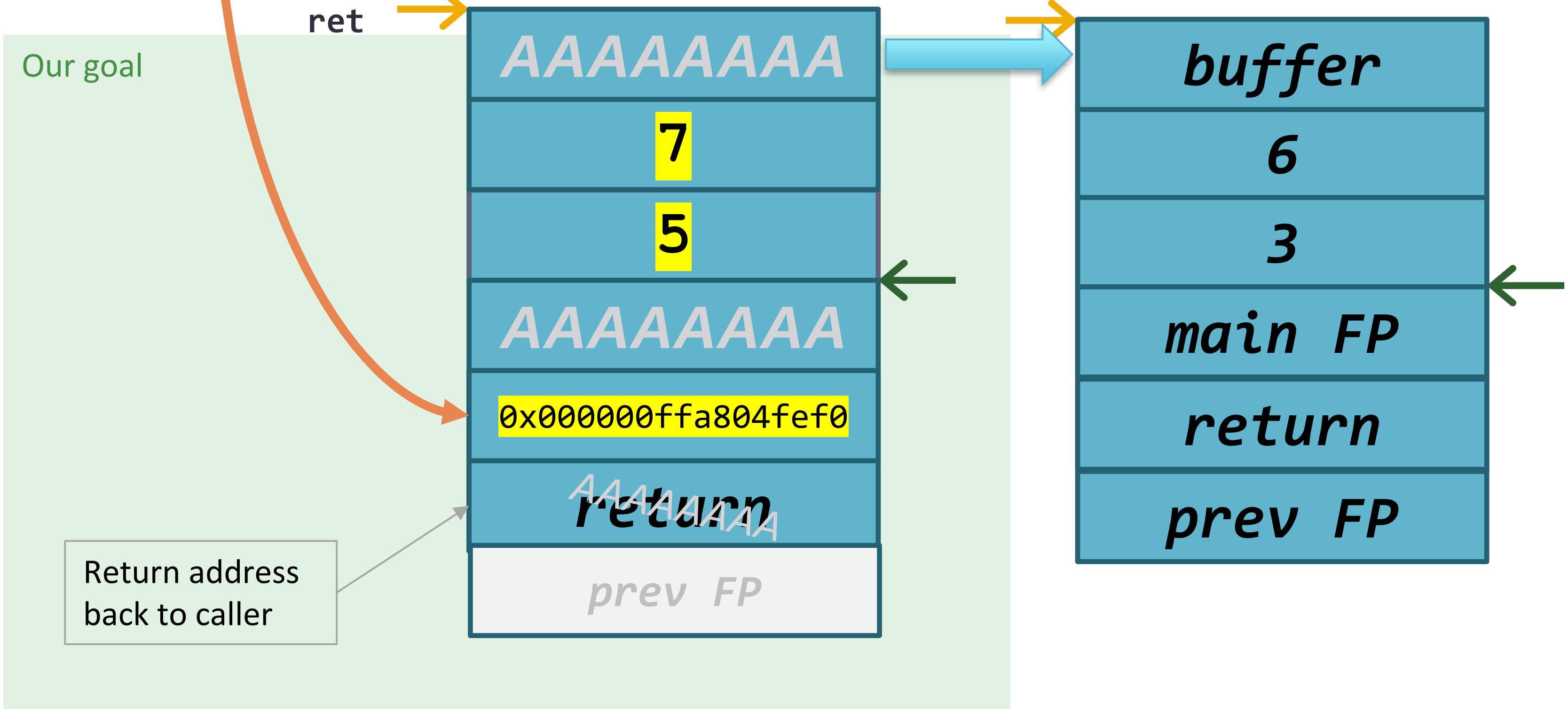
Attacker 2.0 – during gets call

```
0x000000ffa804fef0  ATTACK:
                      push  rbp
                      mov   rbp, rsp
                      ...
                      leave
                      ret
```



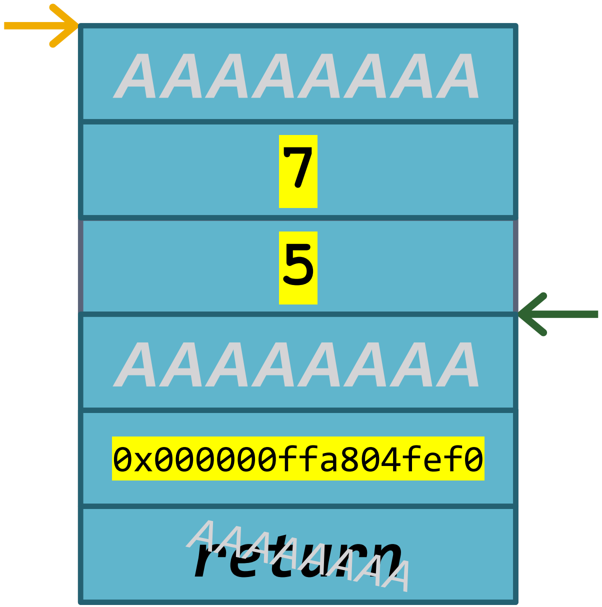
Attacker 2.0 – during gets call

```
0x000000ffa804fef0  ATTACK:
                       push   rbp
                       mov    rbp, rsp
                       ...
                       leave
                       ret
```



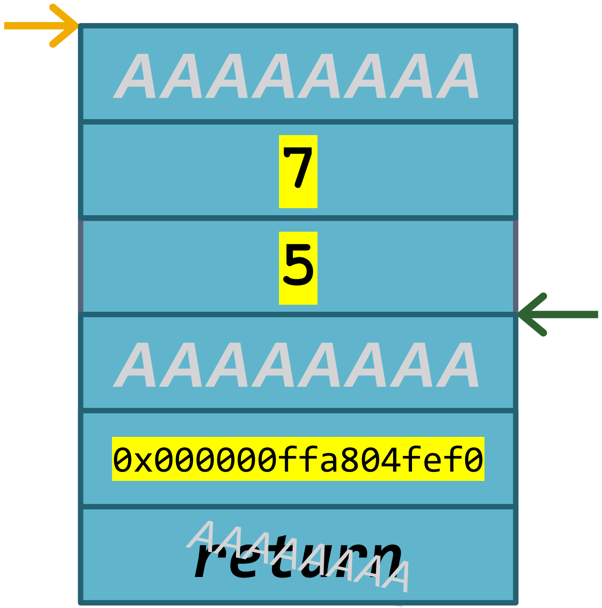
Attack surface after gets

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp - 16]
  call  gets
  mov   rsi, [rbp - 8]
  mov   rdi, [rbp - 4]
  call  foo
  leave
  ret
```



Attack surface after gets

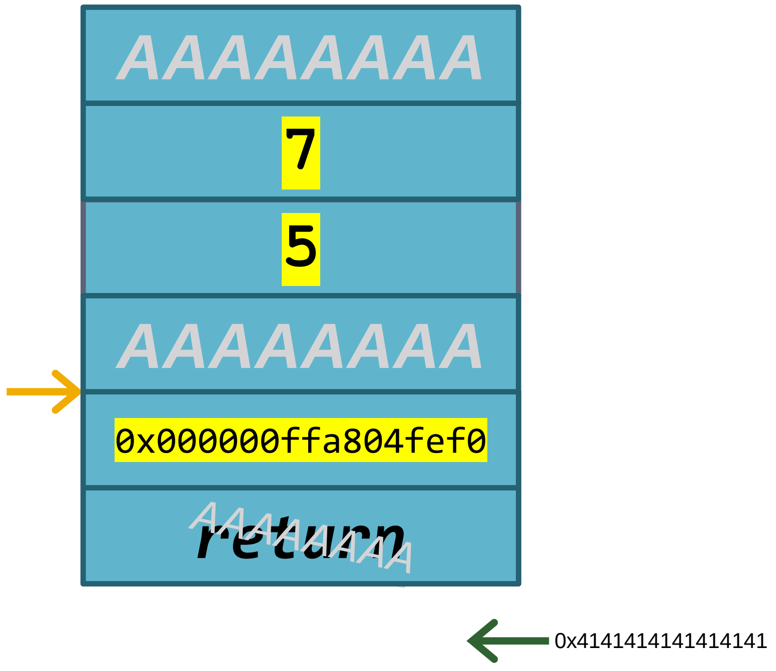
```
vulnerable:
  push   rbp
  mov    rbp, rsp
  push   3
  push   6
  sub    rsp, 8
  lea    rdi, [rbp – 16]
  call   gets
  mov    rsi, [rbp – 8]
  mov    rdi, [rbp – 4]
  call   foo
  leave
  ret
```



Attack surface after gets

```
vulnerable:
  push  rbp
  mov   rbp, rsp
  push  3
  push  6
  sub   rsp, 8
  lea   rdi, [rbp - 16]
  call  gets
  mov   rsi, [rbp - 8]
  mov   rdi, [rbp - 4]
  call  foo
  leave
  ret
```

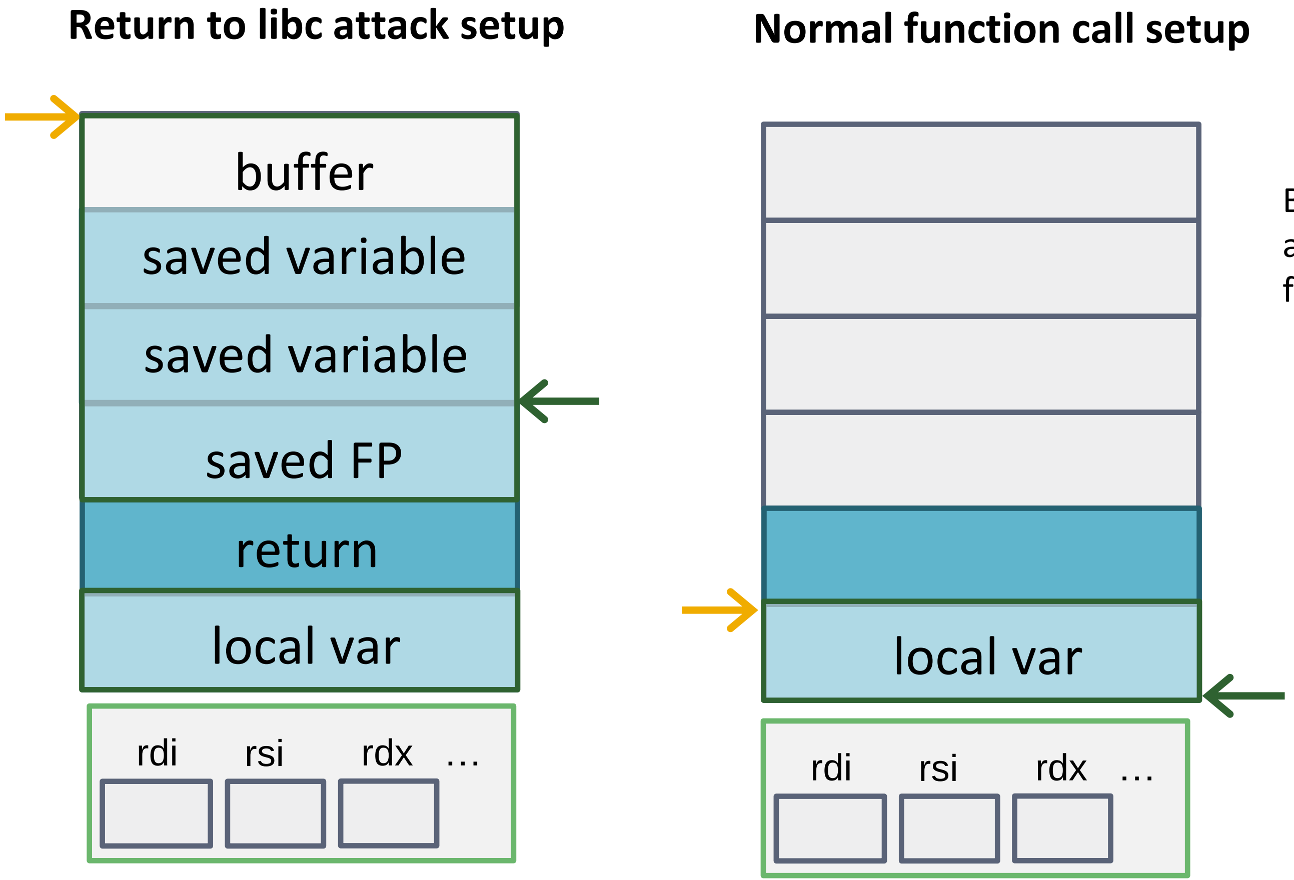
```
pop  rip
```



Let's see it in action

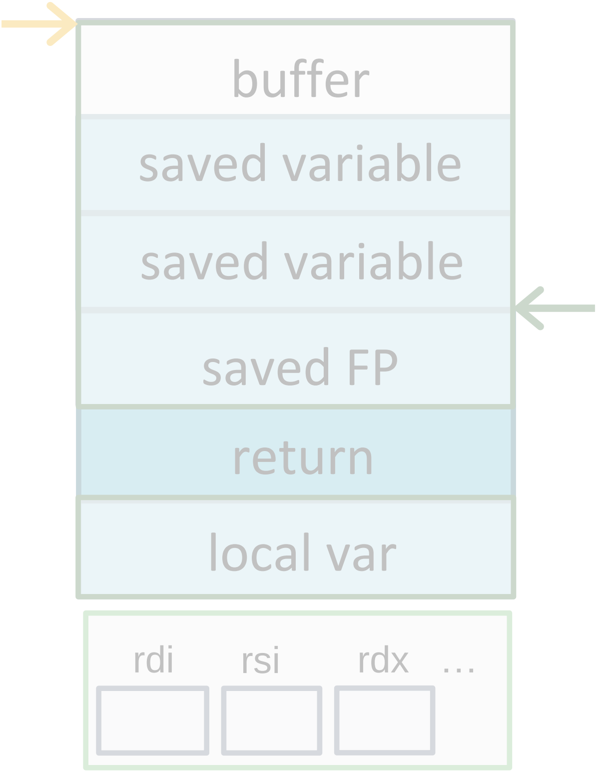
Before call to `gets`

Before pushing arguments for function call

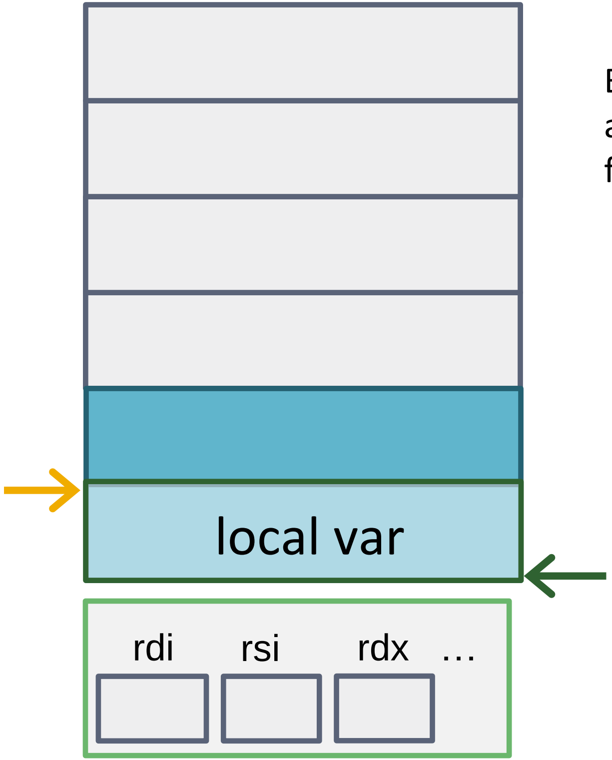


Let's see it in action

Return to libc attack setup



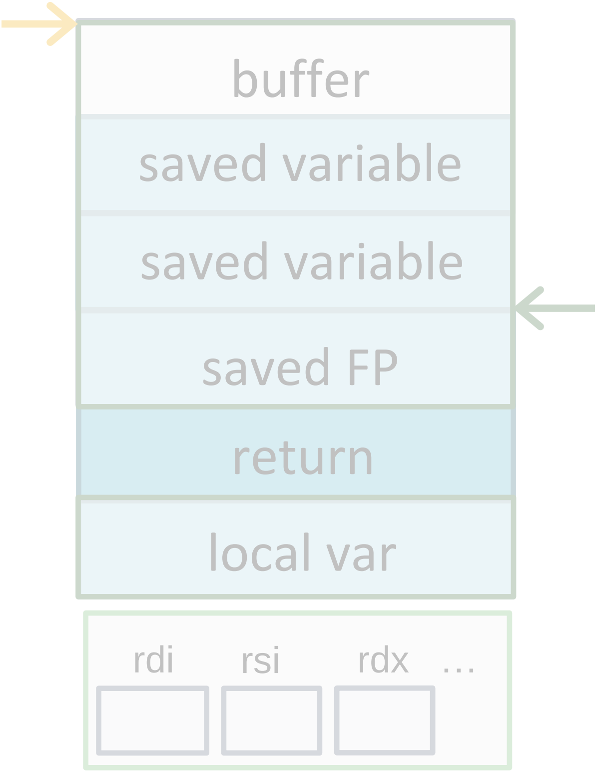
Normal function call setup



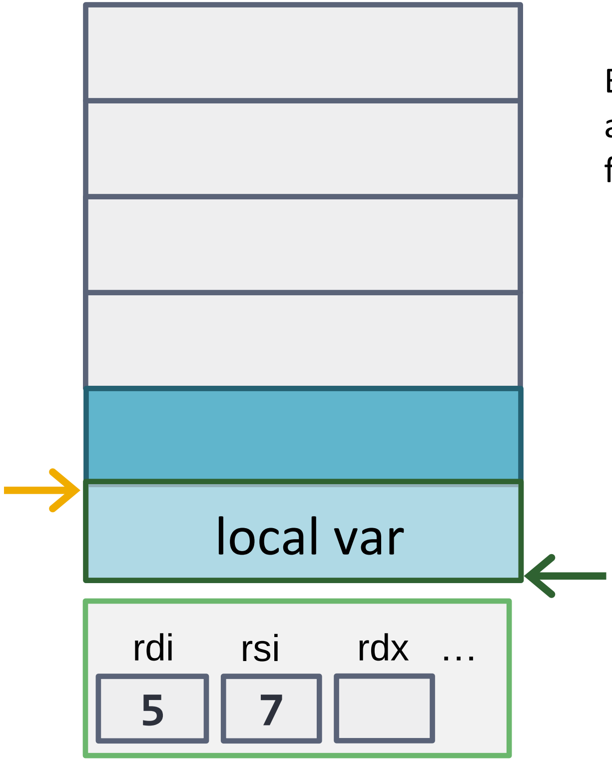
Before pushing arguments for function call

Let's see it in action

Return to libc attack setup



Normal function call setup

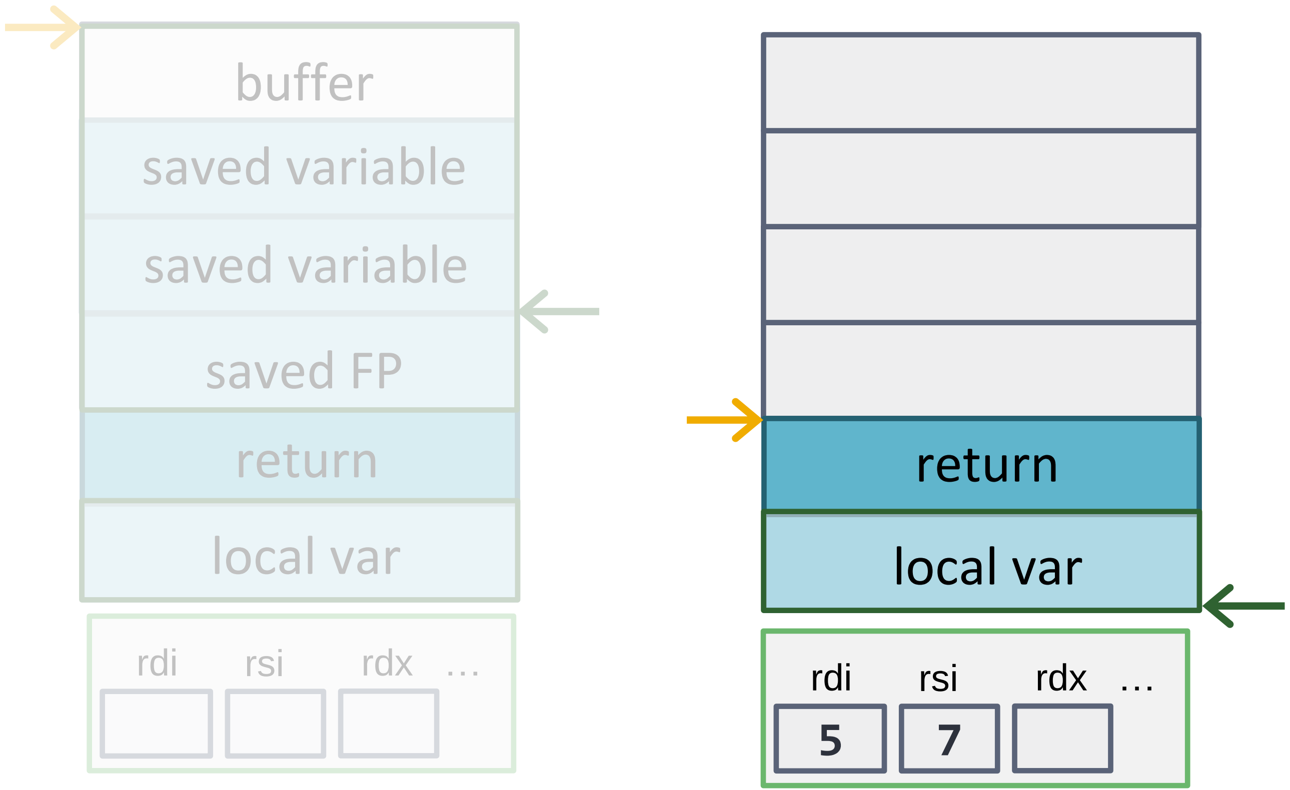


Before pushing arguments for function call

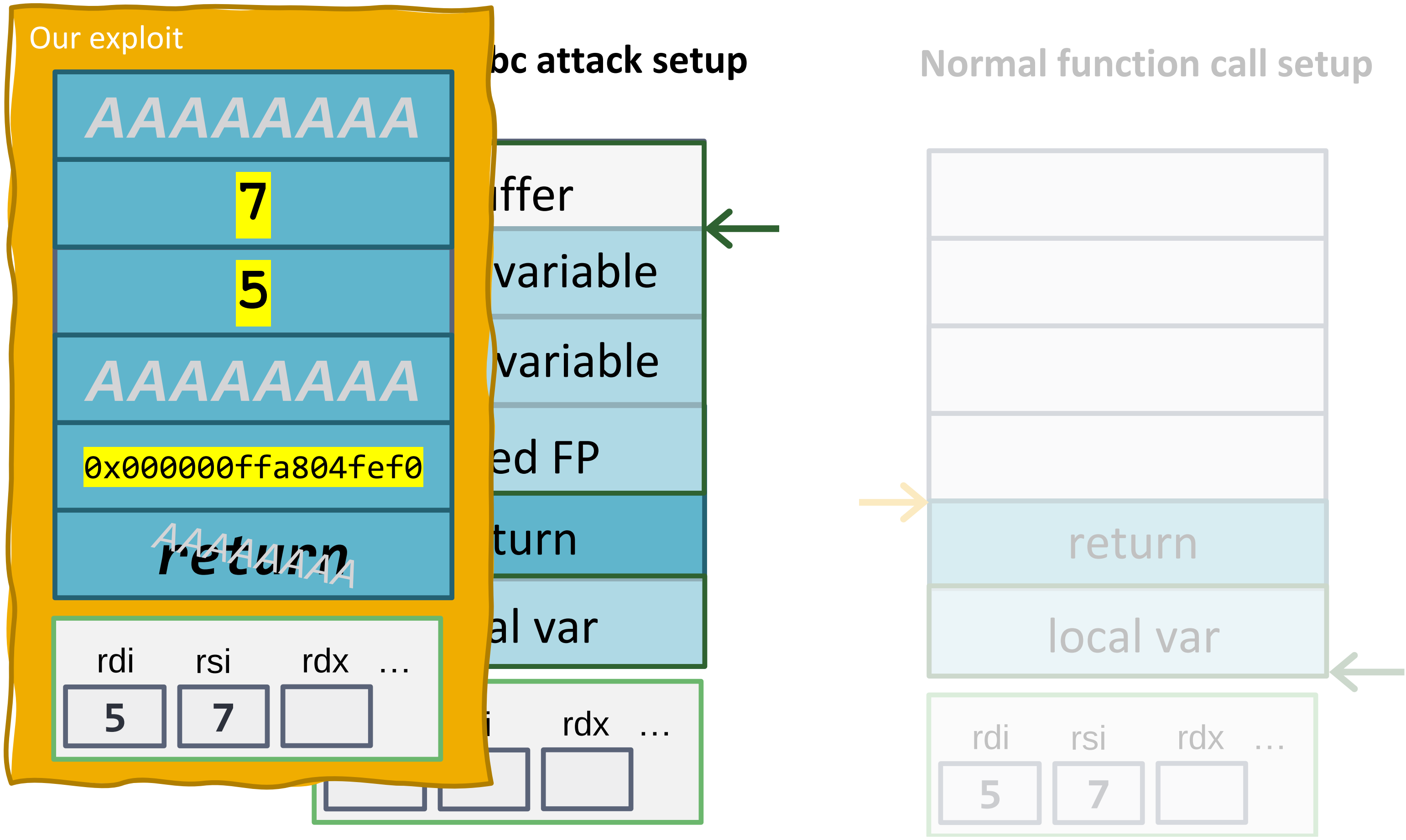
Let’s see it in action

Return to libc attack setup

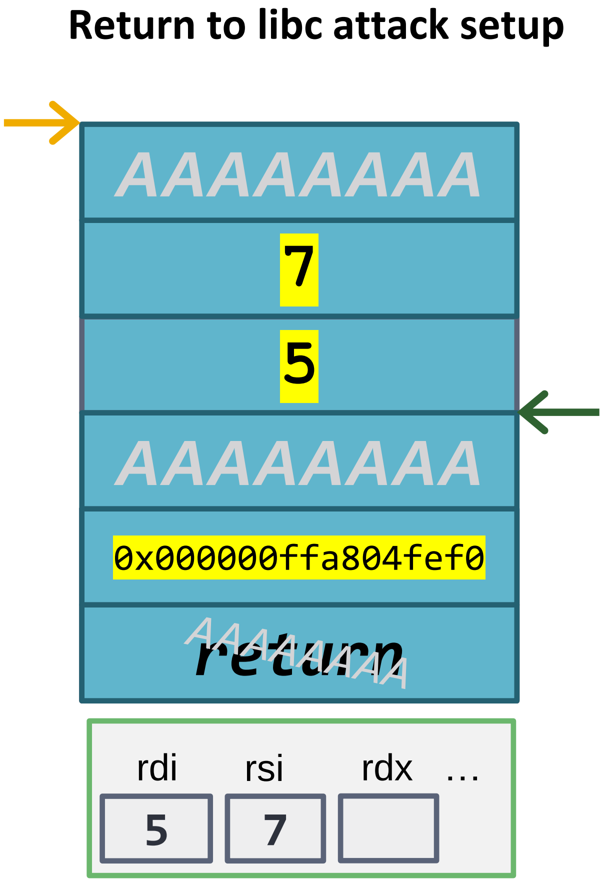
Normal function call setup



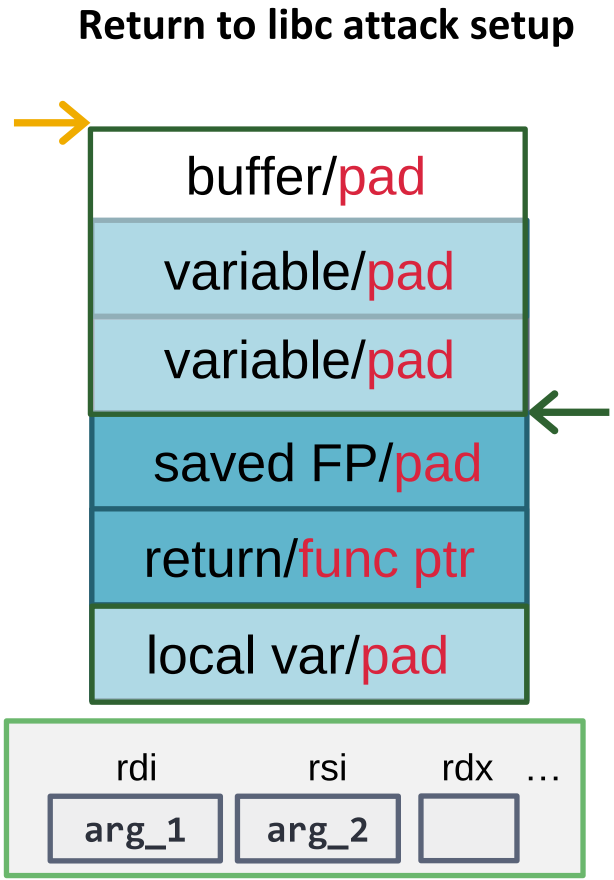
Let’s see it in action

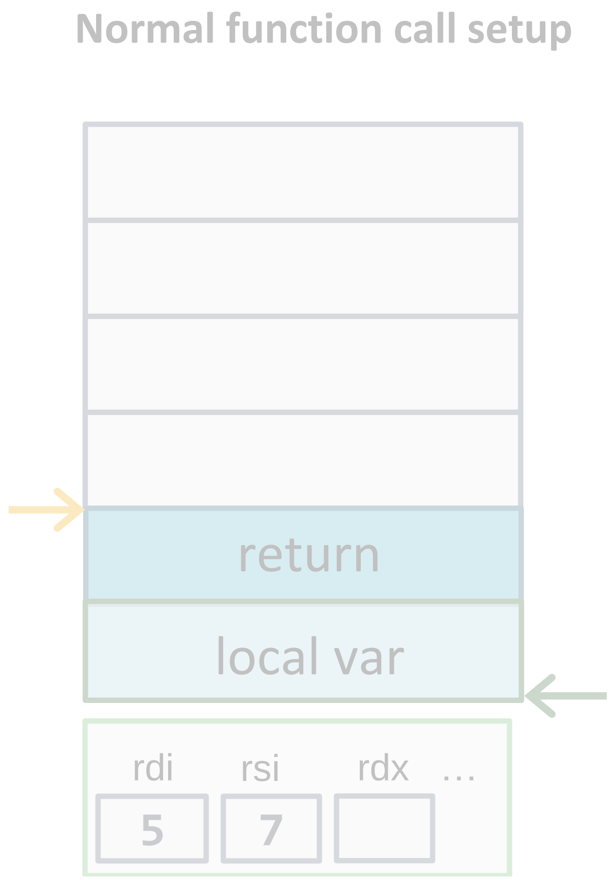


Let’s see it in action



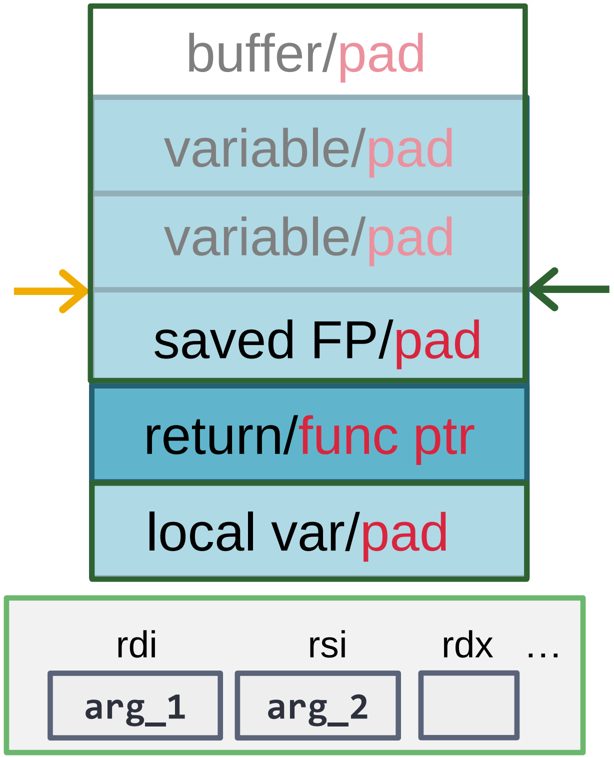
Let’s see it in action



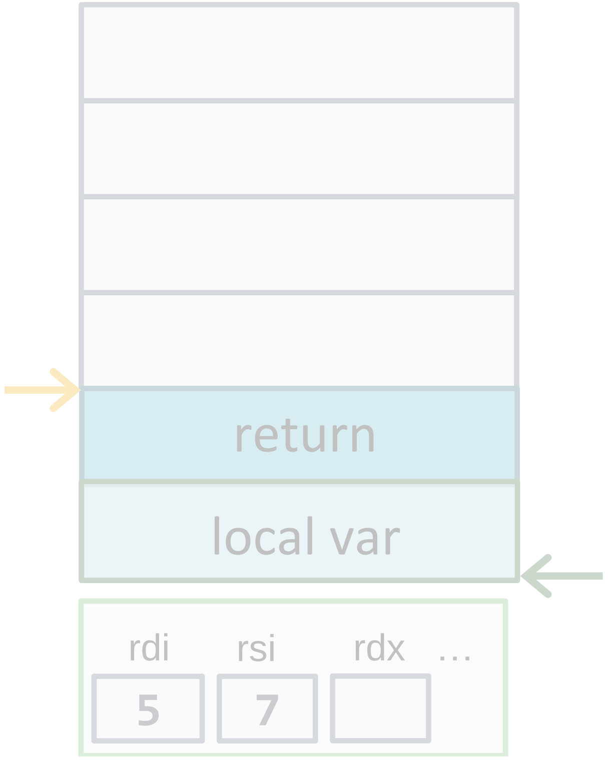


Let’s see it in action

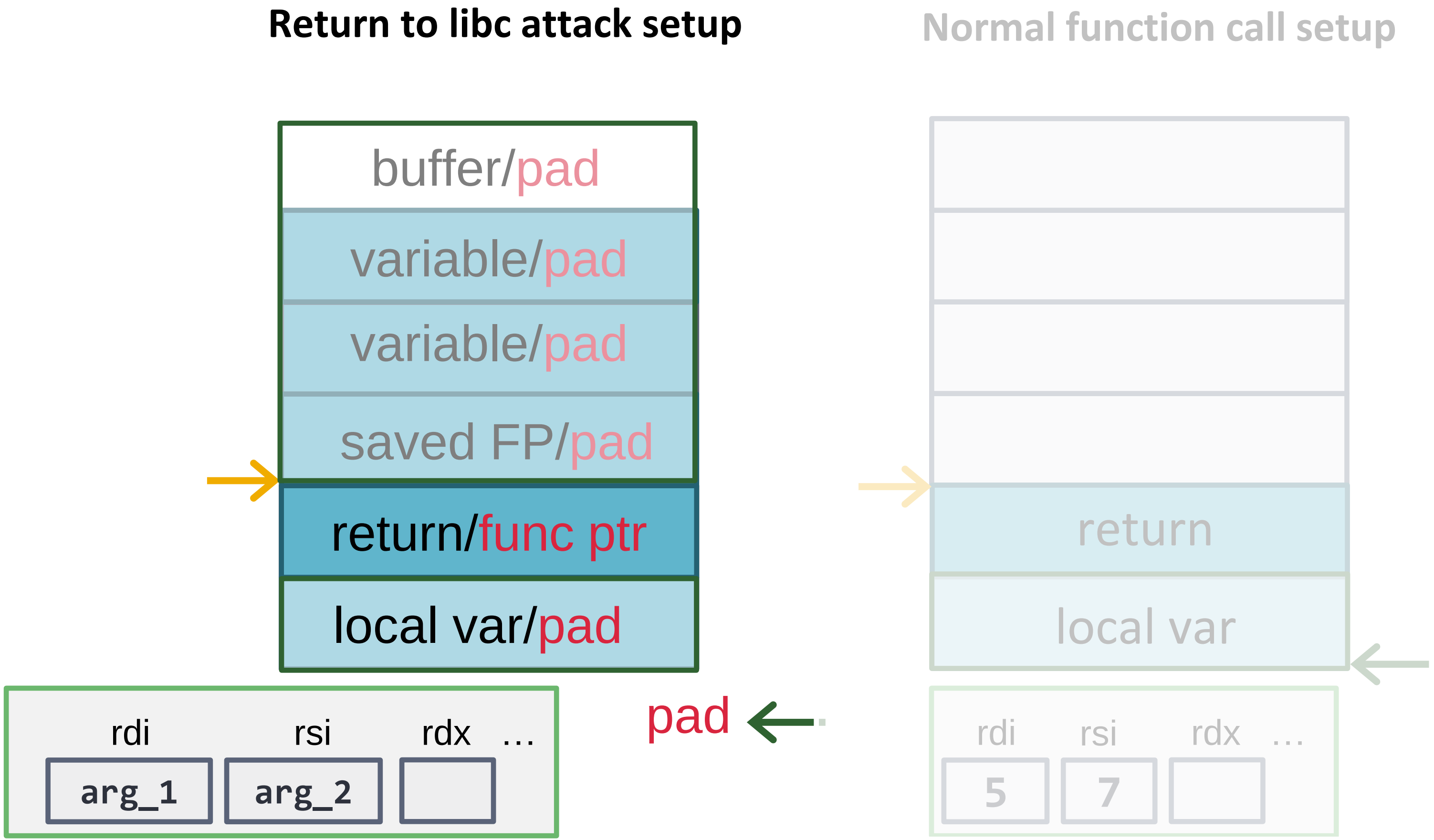
Return to libc attack setup



Normal function call setup



Let’s see it in action

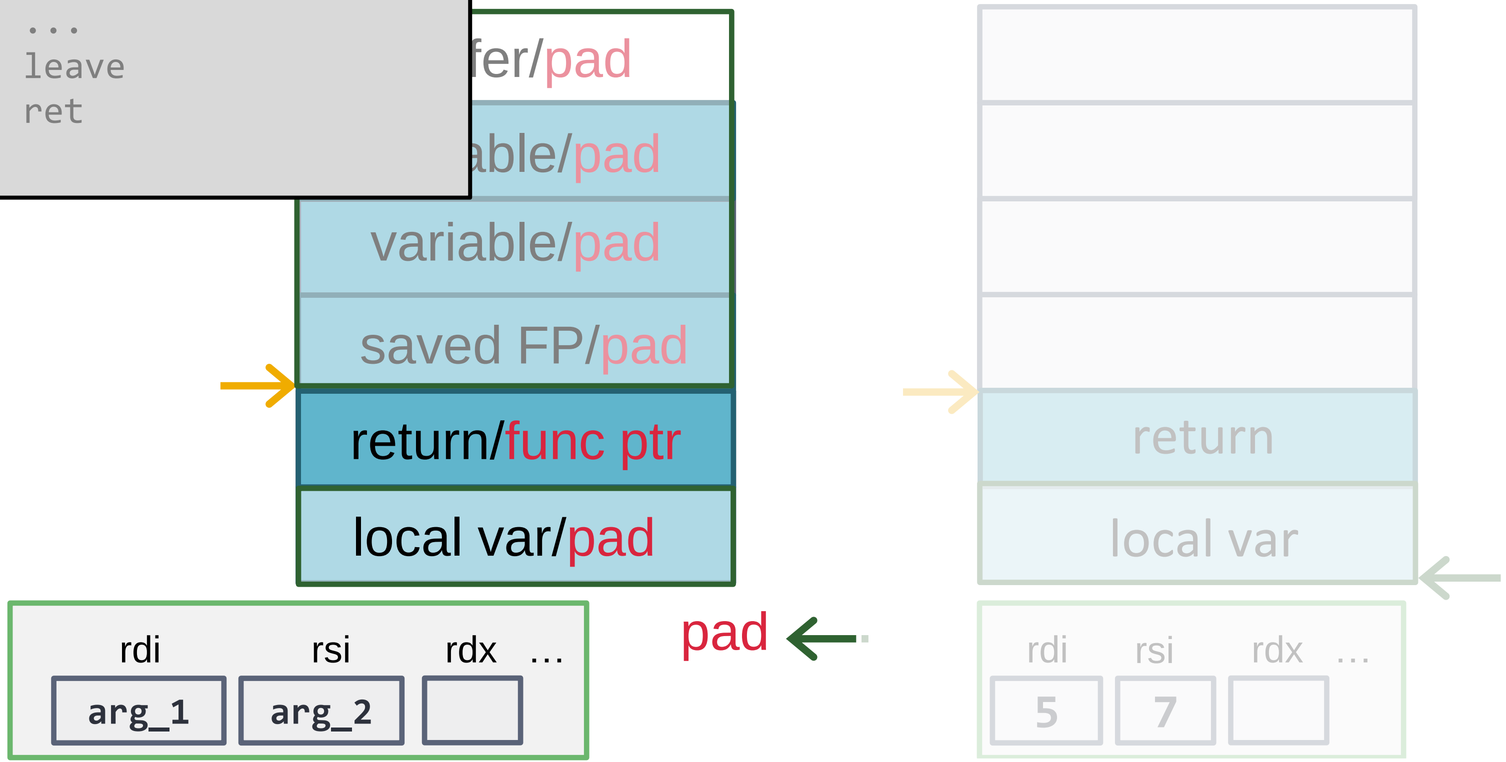


Let’s see it in action

```
ATTACK:
   push  rbp
   mov   rbp, rsp
   ...
   leave
   ret
```

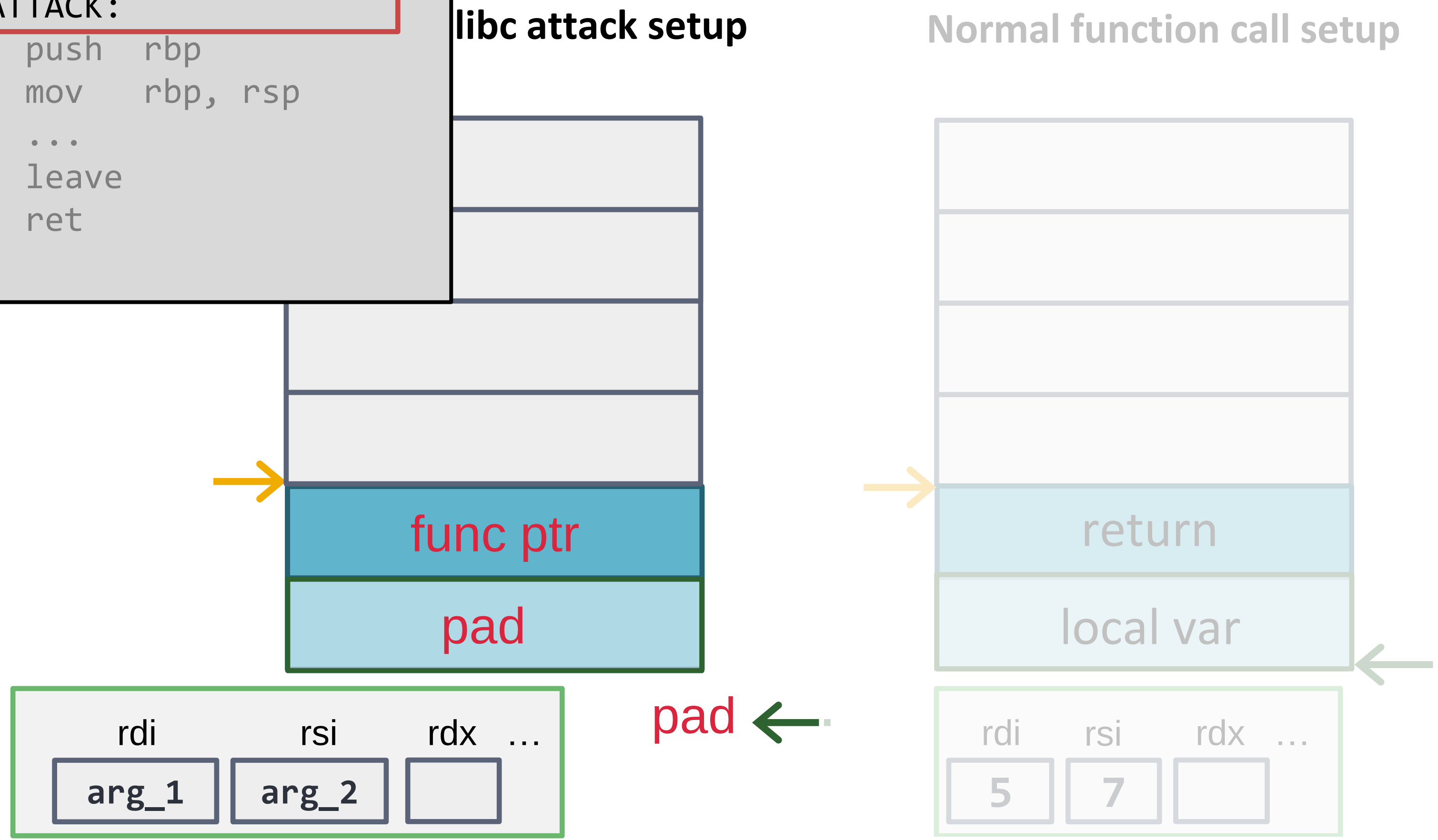
libc attack setup

Normal function call setup



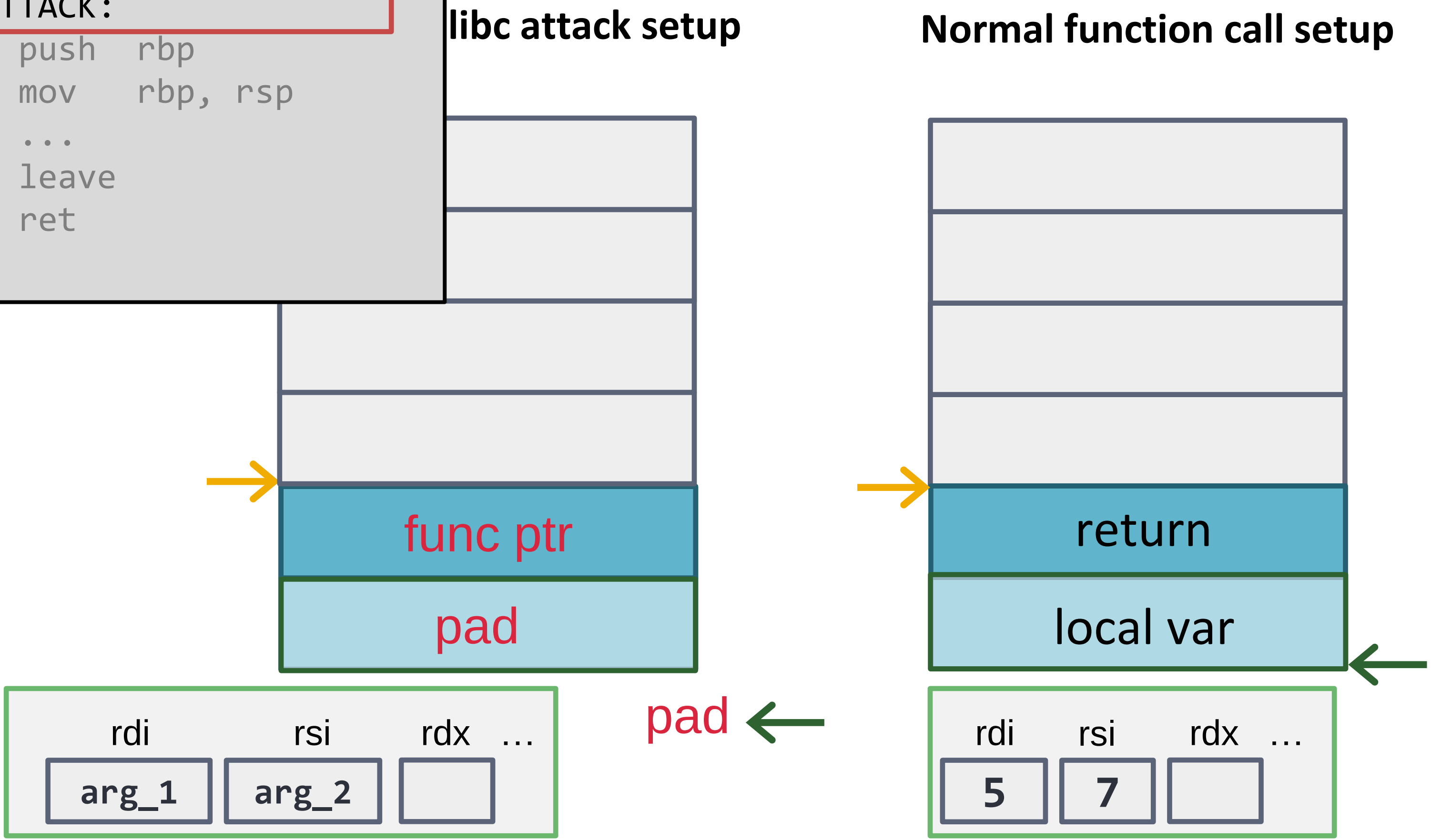
Let’s see it in action

```
ATTACK:
   push  rbp
   mov   rbp, rsp
   ...
   leave
   ret
```



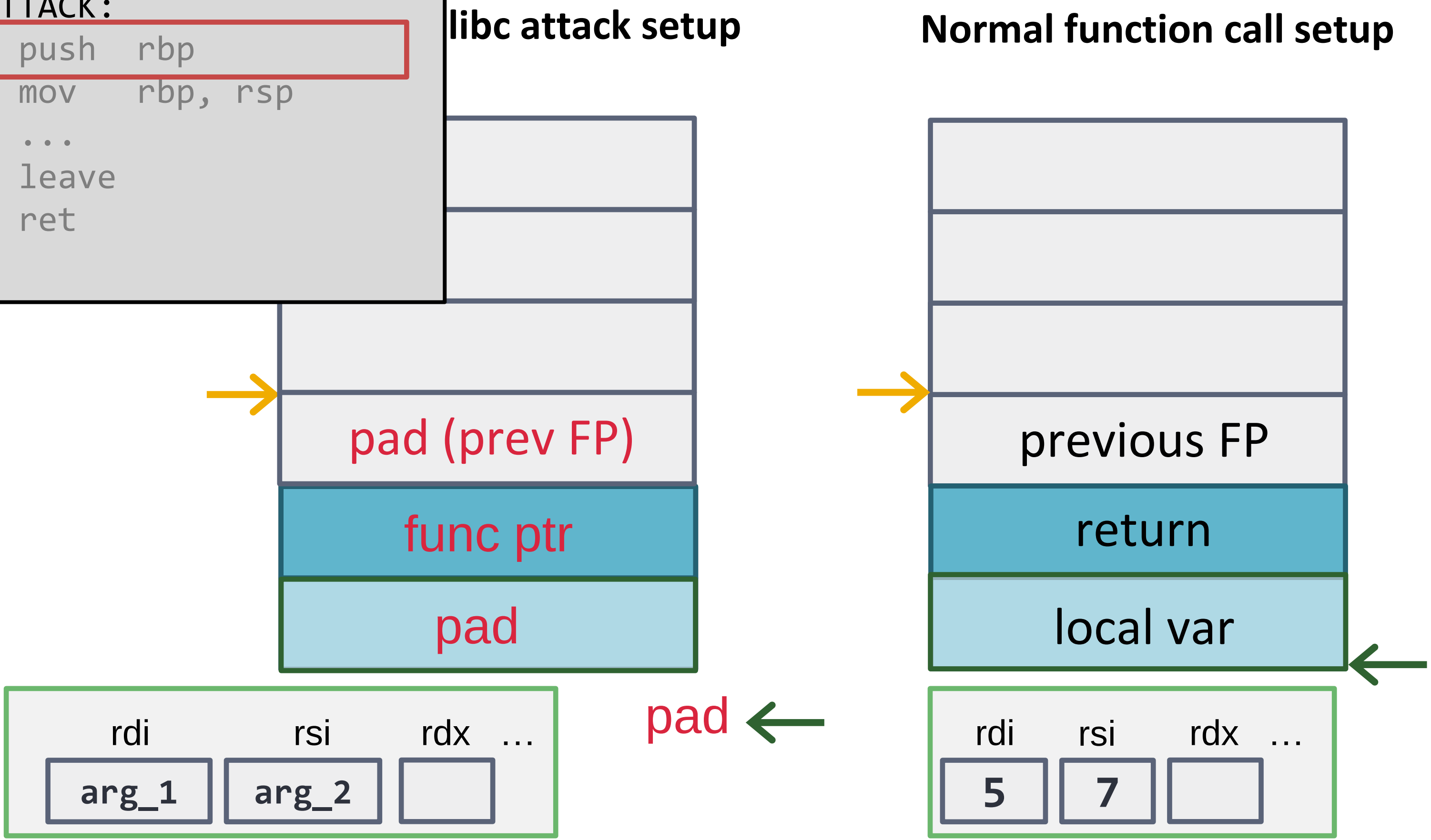
Let’s see it in action

```
ATTACK:
   push  rbp
   mov   rbp, rsp
   ...
   leave
   ret
```



Let’s see it in action

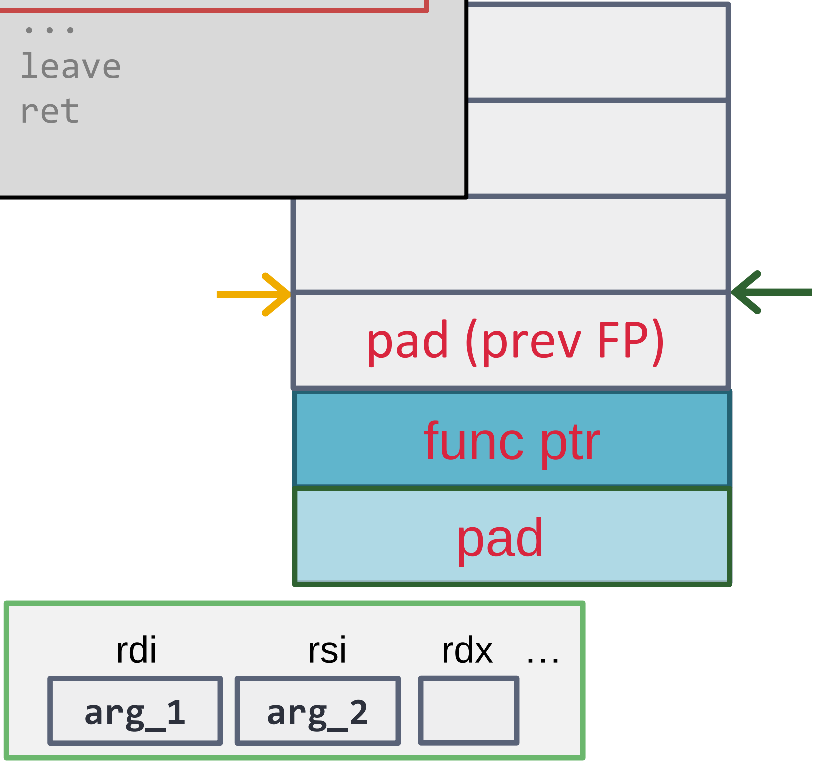
```
ATTACK:
  push  rbp
  mov   rbp, rsp
  ...
  leave
  ret
```



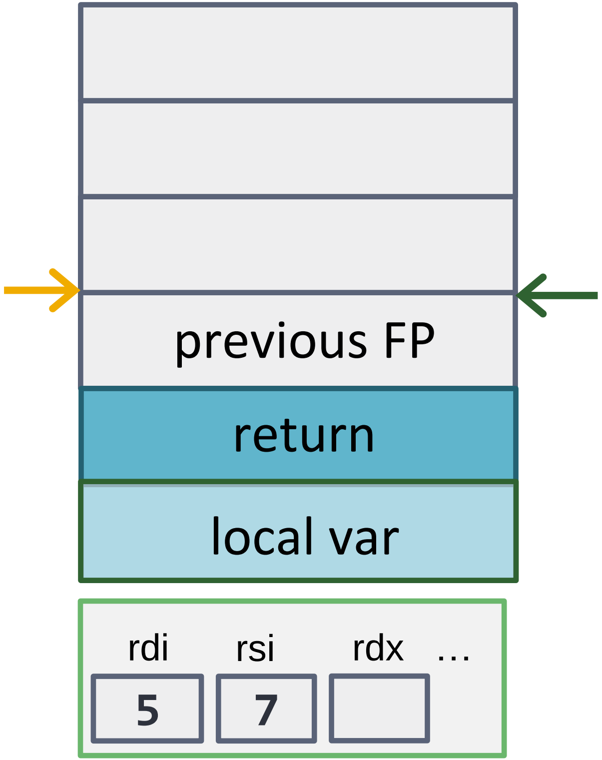
Let’s see it in action

```
ATTACK:
   push  rbp
   mov   rbp, rsp
   ...
   leave
   ret
```

libc attack setup



Normal function call setup



Return to libc

Invoke any function that exists in the binary

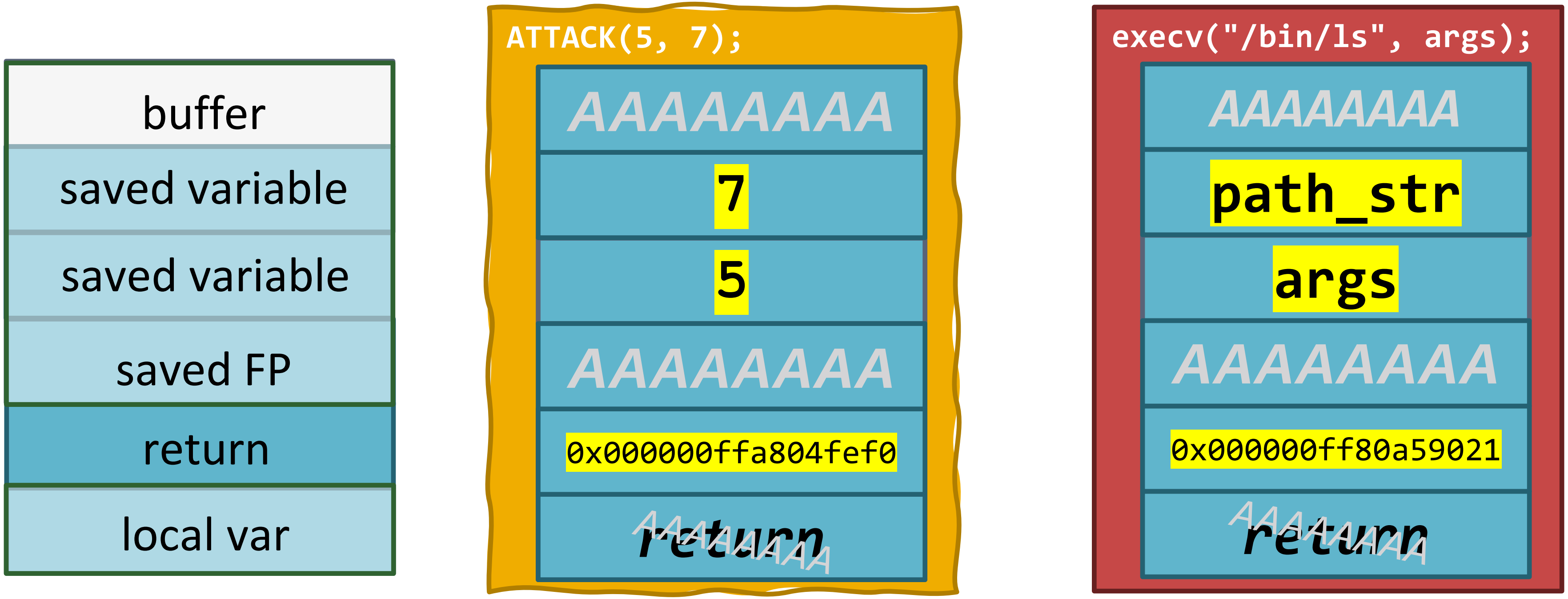
execv is a popular one

The **execv**(), **execvp**(), and **execvpe**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

Return to libc

```
int main() {
    char *args[] = {"/bin/ls", NULL};
    execv("/bin/ls", args);
}
```

Exercise: Walk through previous example using the below payload, and assuming foo takes double length values instead of int.



Return to libc

#include <sys/mman.h>

int mprotect(void **addr*, **size_t** *len*, **int** *prot*);

Description

The *mprotect*() function shall change the access protections to be that specified by *prot* for those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* argument should be either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE, and PROT_EXEC.

Return to libc: defense

Defender:

The attackers are calling other functions

Defender:

Let’s remove functions we don’t need!

Cat-and-Mouse Exploitation

Buffer Overflow

Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

Return Oriented Programming (ROP)

ROP

Defender:

Take out functions that can launch shells

Attacker:

Use the instructions that are still there

ROP

Return Oriented Programming

Return to libc without calling full functions

Build arbitrary functionality via “gadgets”

- Turing complete

**The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)**

Hovav Shacham*
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA
hovav@hovav.net

ABSTRACT

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set. using the short sequences we find in a specific distribution of GNU libc, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our *thesis*.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing libc to recover the instruction sequences that can be used in

ROP Gadget

Small section of code

Contains a very small number of instructions

Ends in a `ret`

Not an existing function body

ROP

```
arg[10] = 0x00
```

```
foo:
  push rbp
  mov rsp, rbp
  mov rax, [rbp + 4]
  add rax, 10
  mov [rax], 0x00
  sub rax, 10
  leave
  ret
```

```
var = var - 10
```

```
foo+0x20:
  sub rax, 10
  leave
  ret
```

ROP

Variable-length instructions: The achilles heel of x86.

$$\underbrace{\texttt{01 01}}_{\texttt{add [rcx], rax}}\quad\underbrace{\texttt{c3}}_{\texttt{ret}}$$

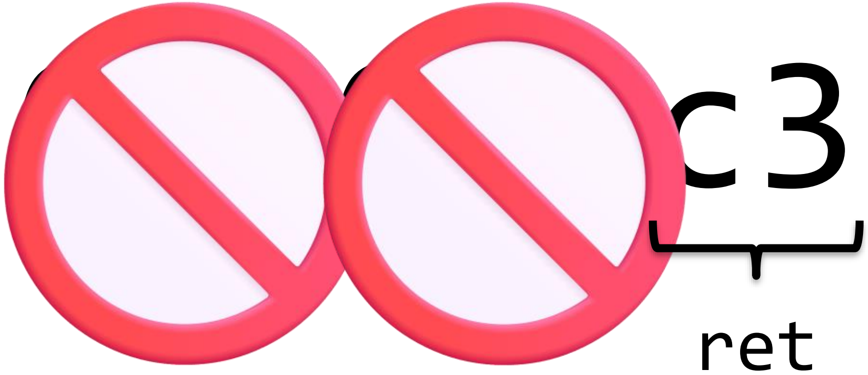
ROP

Variable-length instructions: The achilles heel of x86.

$$\underbrace{\texttt{01 c3}}_{\texttt{add rbx, rax}}$$

ROP

Variable-length instructions: The achilles heel of x86.



ROP

Bytes in the Code Section:

`00 F7 C7 07 00 00 00 0f 95 53 c3`

Full Gadget:

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

$\underbrace{\texttt{c3}}_{\texttt{ret}}$

Full Gadget:

```
ret
```

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

push rbx ret

Full Gadget:

```
push rbx
ret
```

ROP

rip

$$\texttt{00 F7 C7 07 00 00 00 0f }\underbrace{\texttt{95}}_{\texttt{xchg rbp, rax}}\ \underbrace{\texttt{53}}_{\texttt{push rbx}}\ \underbrace{\texttt{c3}}_{\texttt{ret}}$$

Full Gadget:

```
xchg rbp, rax
push rbx
ret
```

ROP

rip

00 F7 C7 07 00 00 00 0f 95 53 c3

setne [rbx - 61]

Full Gadget:

```
setne [rbx-61]
<no return>
```

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

Full Gadget:

<none – invalid instruction>

ROP

rip

`00 F7 C7 07 00 00 00 0f 95 53 c3`

Full Gadget:

`<none - invalid instruction>`

ROP

rip

`00 F7 C7 07 00 00 00 0f 95 53 c3`

Full Gadget:

`<none – invalid instruction>`

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

Full Gadget:

```
<none - invalid instruction>
```

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

mov rdi, 0x0F000000

push rbx

ret

xchg rbp, rax

Full Gadget:

```
mov rdi, 0x0F000000
xchg rbp, rax
push rbx
ret
```

ROP

rip

`00 F7 C7 07 00 00 00 0f 95 53 c3`

`test rdi, 0x00000007`

`setne [rbx – 61]`

Full Gadget:

```
test rdi, 0x00000007
setne [rbx-61]
<no return>
```

ROP

rip

```
00 F7 C7 07 00 00 00 0f 95 53 c3
```

$\underbrace{\texttt{00 F7}}_{\texttt{add bh, dh}}\ \underbrace{\texttt{C7 07 00 00 00 0f}}_{\texttt{mov rdi, 0x0F000000}}\ \underbrace{\texttt{95}}_{\texttt{xchg rbp, rax}}\ \underbrace{\texttt{53}}_{\texttt{push rbx}}\ \underbrace{\texttt{c3}}_{\texttt{ret}}$

Full Gadget:

```
add bh, dh
mov rdi, 0x0F000000
xchg rbp, rax
push rbx
ret
```

ROP

```
8052867:        74 1c                   je     8052885 <__sprintf_chk@plt+0x8ab5>
8052869:        8b 6d 04                mov    ebp,DWORD PTR [ebp+0x4]
805286c:        83 c3 01                add    ebx,0x1
805286f:        85 ed                   test   ebp,ebp
8052871:        75 e5                   jne    8052858 <__sprintf_chk@plt+0x8a88>
8052873:        8b 54 24 30             mov    edx,DWORD PTR [esp+0x30]
8052877:        83 44 24 0c 08          add    DWORD PTR [esp+0xc],0x8
805287c:        8b 44 24 0c             mov    eax,DWORD PTR [esp+0xc]
8052880:        39 42 04                cmp    DWORD PTR [edx+0x4],eax
8052883:        77 bf                   ja     8052844 <__sprintf_chk@plt+0x8a74>
8052885:        83 c4 1c                add    esp,0x1c
8052888:        89 d8                   mov    eax,ebx
805288a:        5b                      pop    ebx
805288b:        5e                      pop    esi
805288c:        5f                      pop    edi
805288d:        5d                      pop    ebp
805288e:        c3                      ret
805288f:        90                      nop
8052890:        56                      push   esi
8052891:        53                      push   ebx
8052892:        31 d2                   xor    edx,edx
8052894:        8b 5c 24 0c             mov    ebx,DWORD PTR [esp+0xc]
8052898:        8b 74 24 10             mov    esi,DWORD PTR [esp+0x10]
805289c:        0f b6 0b                movzx  ecx,BYTE PTR [ebx]
805289f:        84 c9                   test   cl,cl
80528a1:        74 1c                   je     80528bf <__sprintf_chk@plt+0x8aef>
80528a3:        90                      nop
80528a4:        8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
80528a8:        89 d0                   mov    eax,edx
80528aa:        83 c3 01                add    ebx,0x1
80528ad:        c1 e0 05                shl    eax,0x5
80528b0:        29 d0                   sub    eax,edx
80528b2:        31 d2                   xor    edx,edx
80528b4:        01 c8                   add    eax,ecx
                                                                   11351,73-81    50%
```

`0xc3 : ret`

Could be part of another instruction!

Empirically, about one in every 178 bytes

ROP chains

Gadget 1:

```
mov rax, 0x10; ret
```

Gadget 2:

```
add rax, rbp; ret
```

Gadget 3:

```
mov [rax+8], rax;
ret
```

Gadget 4:

```
mov rbp, rsp; ret
```

ROP chains

Gadget 1:

```
mov rax, 0x10; ret
```

Gadget 2:

```
add rax, rbp; ret
```

Gadget 3:

```
mov [rax+8], rax;
ret
```

Gadget 4:

```
mov rbp, rsp; ret
```

buffer
local var
saved FP
ret
local var
prev FP
prev ret

ROP chains

Gadget 1:

```
mov rax, 0x10; ret
```

Gadget 2:

```
add rax, rbp; ret
```

Gadget 3:

```
mov [rax+8], rax;
ret
```

Gadget 4:

```
mov rbp, rsp; ret
```

pad
pad
pad
&gadget 1
&gadget 2
&gadget 3
&gadget 4

ROP chains

ROP Chain:

```
mov rax, 0x10; ret
add rax, rbp; ret
add rax, rbp; ret
mov [rax+8], rax; ret
mov rbp, rsp
ret
```

ROP chains

ROP Chain:

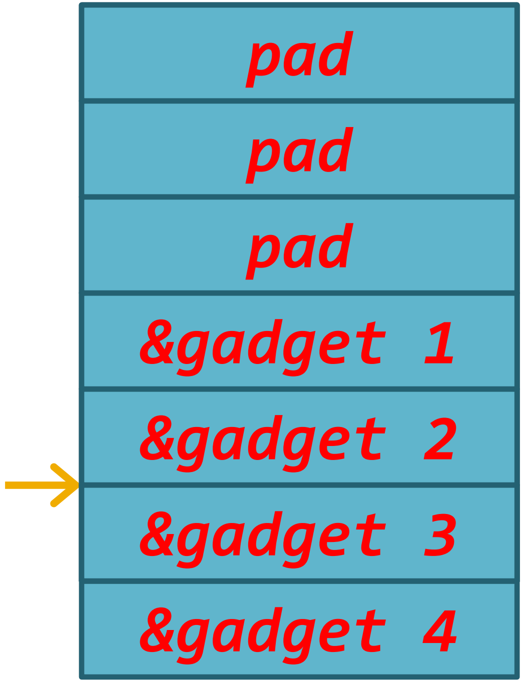
```
mov rax, 0x10; ret
add rax, rbp; ret
add rax, rbp; ret
mov [rax+8], rax; ret
mov rbp, rsp
ret
```

pad
pad
pad
&gadget 1
&gadget 2
&gadget 3
&gadget 4

ROP chains

ROP Chain:

```
mov rax, 0x10; ret
add rax, rbp; ret
add rax, rbp; ret
mov [rax+8], rax; ret
mov rbp, rsp
ret
```



ROP chains

ROP Chain:

```
mov rax, 0x10; ret
add rax, rbp; ret
add rax, rbp; ret
mov [rax+8], rax; ret
mov rbp, rsp
ret
```

ROP chains

ROP Chain:

```
mov rax, 0x10; ret
add rax, rbp; ret
add rax, rbp; ret
mov [rax+8], rax; ret
mov rbp, rsp
ret
```

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR

ASLR

Defender:

We can’t take out all the rets from our code

Defender:

Let’s just move around where the code lives

ASLR

Address Space Layout Randomization

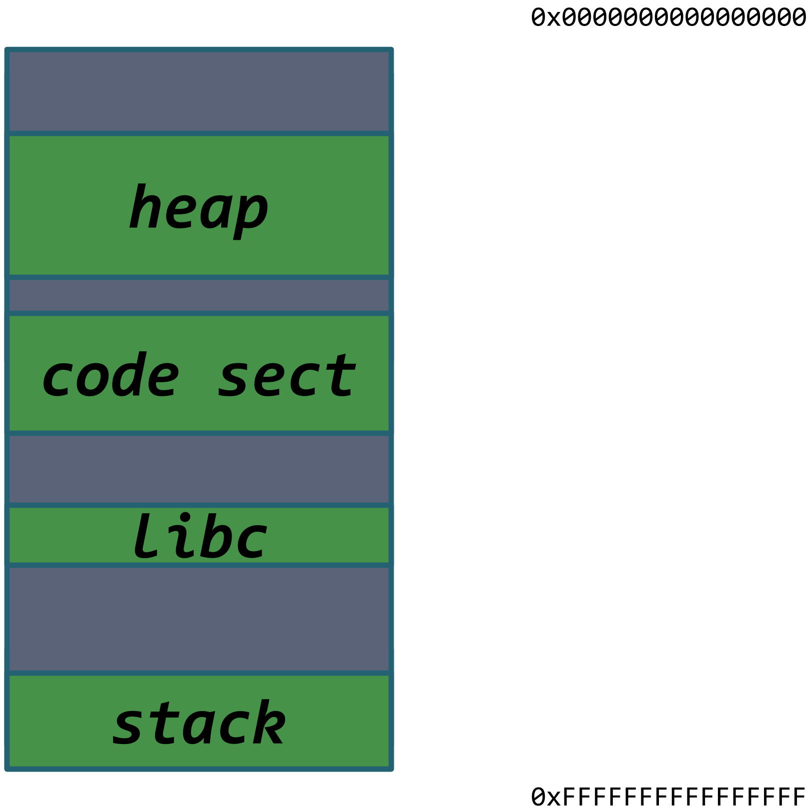
Make it extremely hard to predict references

Requires many changes to compilation and/or loading

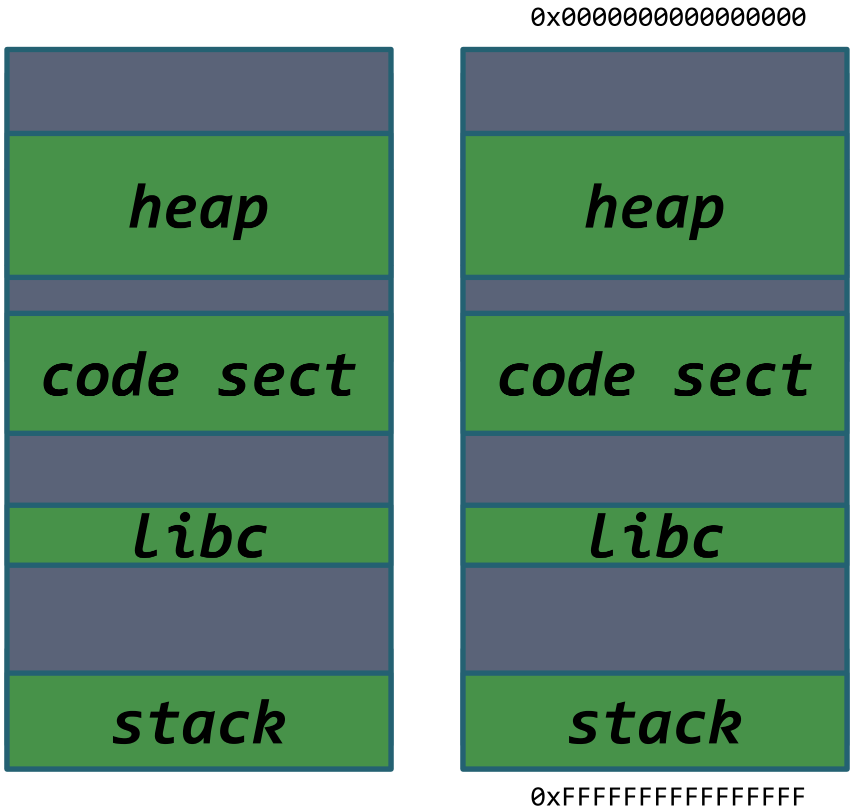
- Code must be “relocatable” or “position independent”

<Details are out-of-scope>

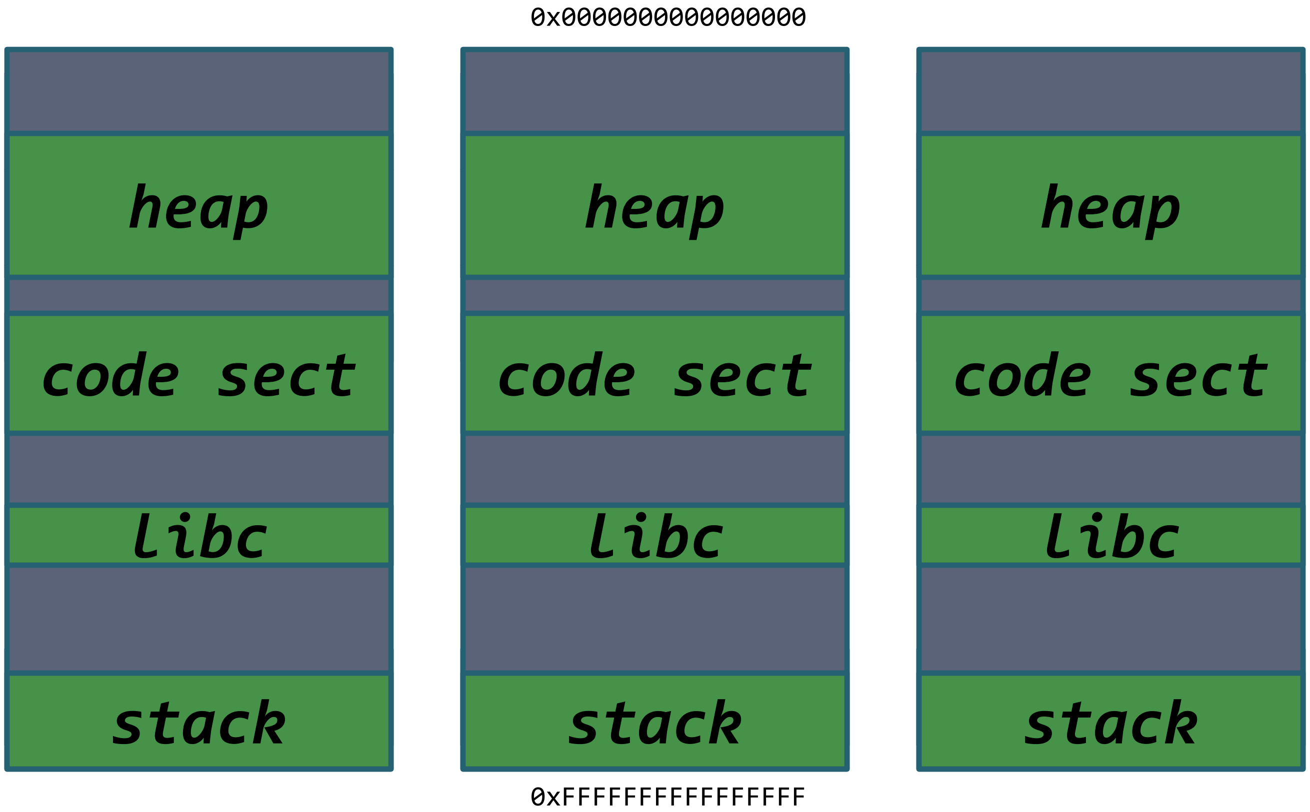
Memory Layout (no ASLR)



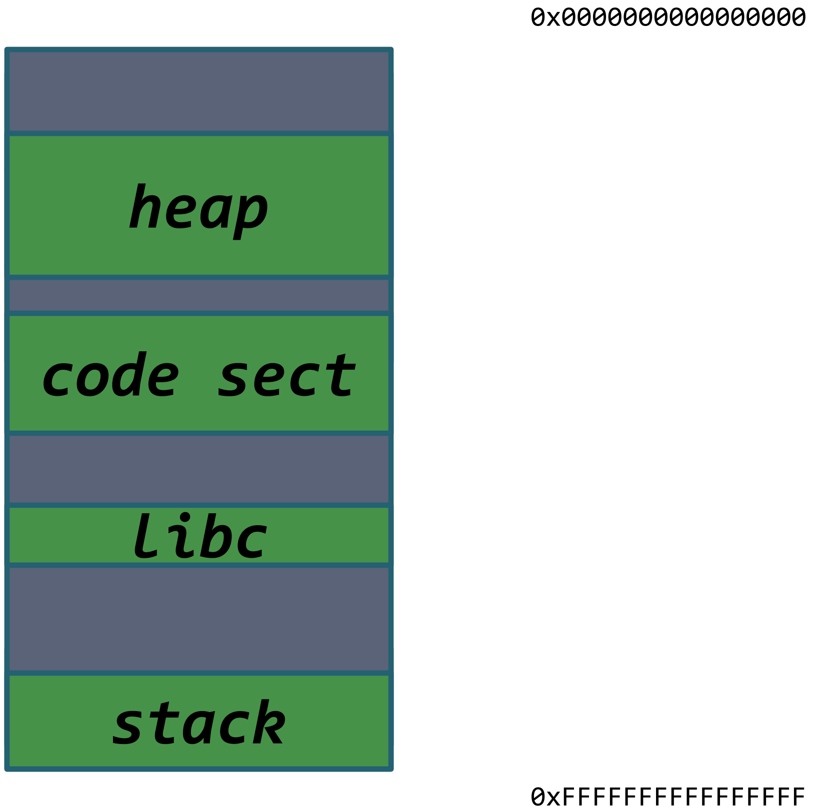
Memory Layout (no ASLR)



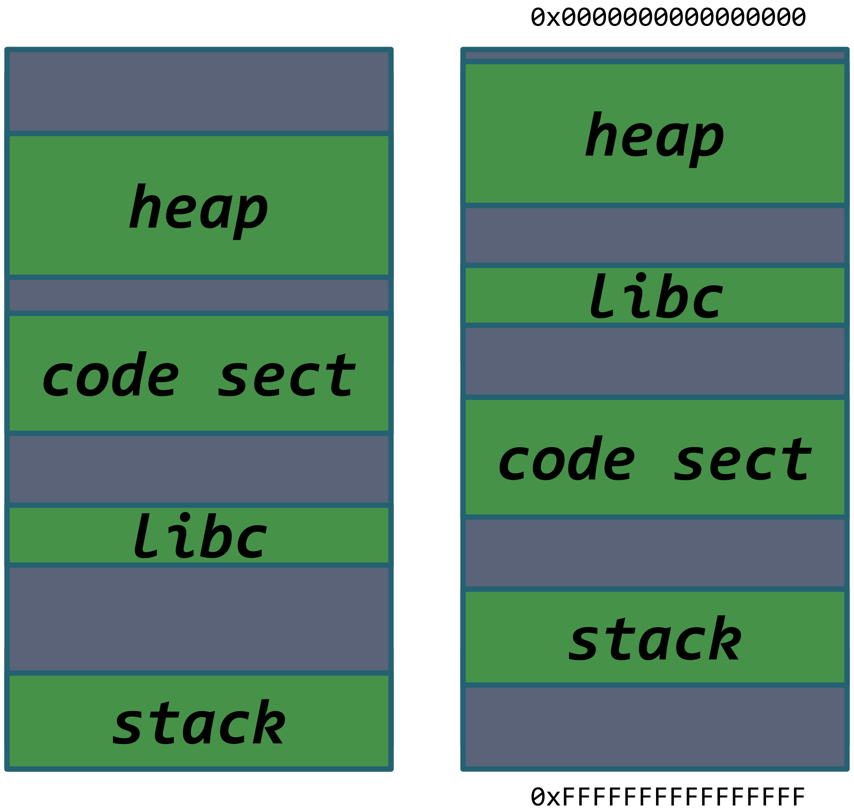
Memory Layout (no ASLR)



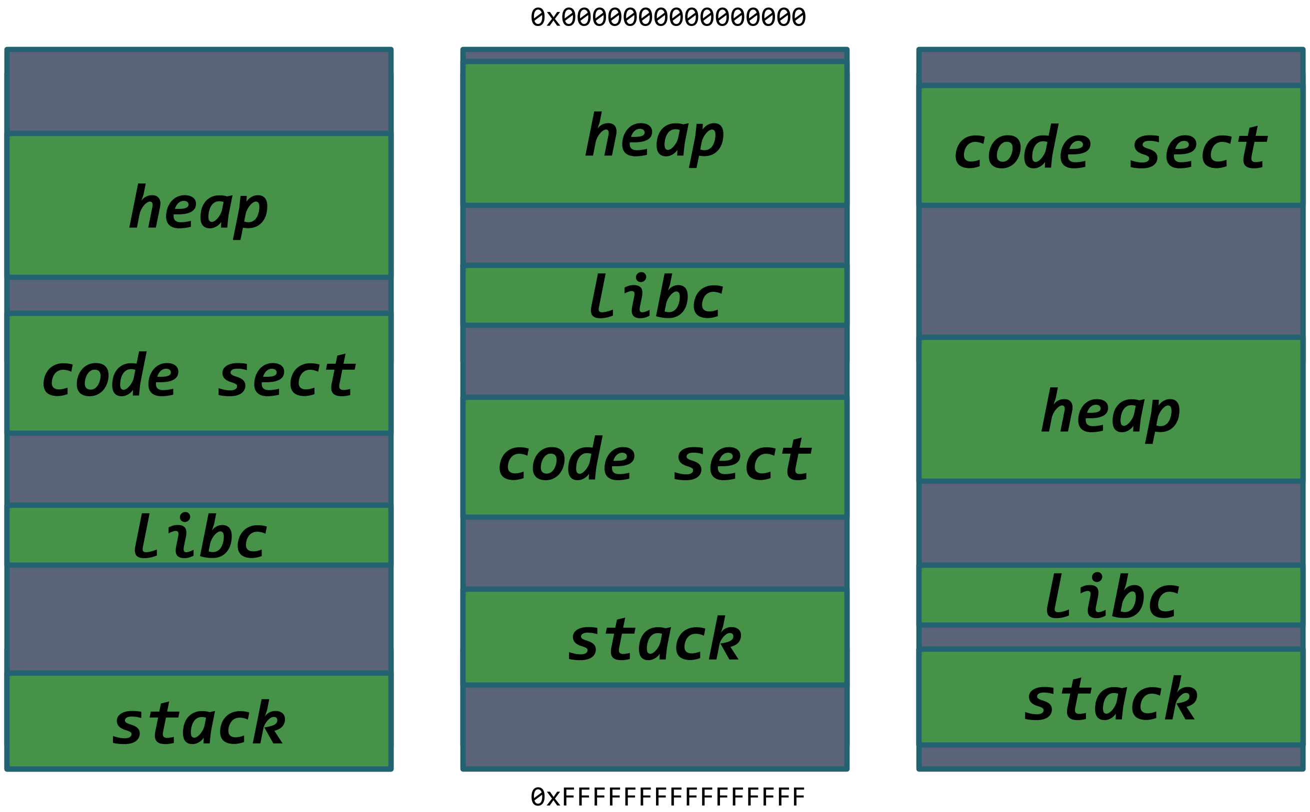
Memory Layout (with ASLR)



Memory Layout (with ASLR)



Memory Layout (with ASLR)



ASLR

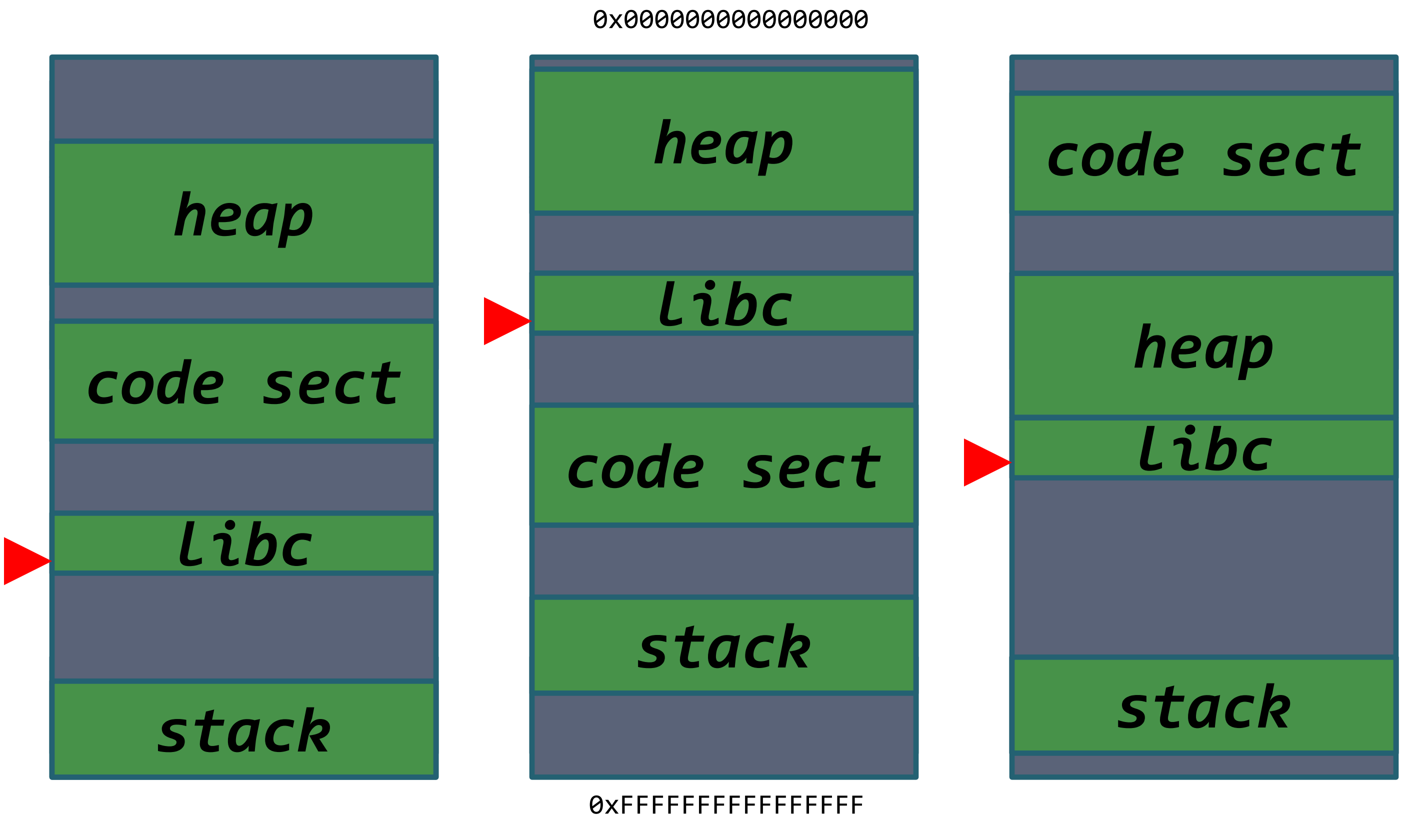
How can we defeat ASLR?

Hint:

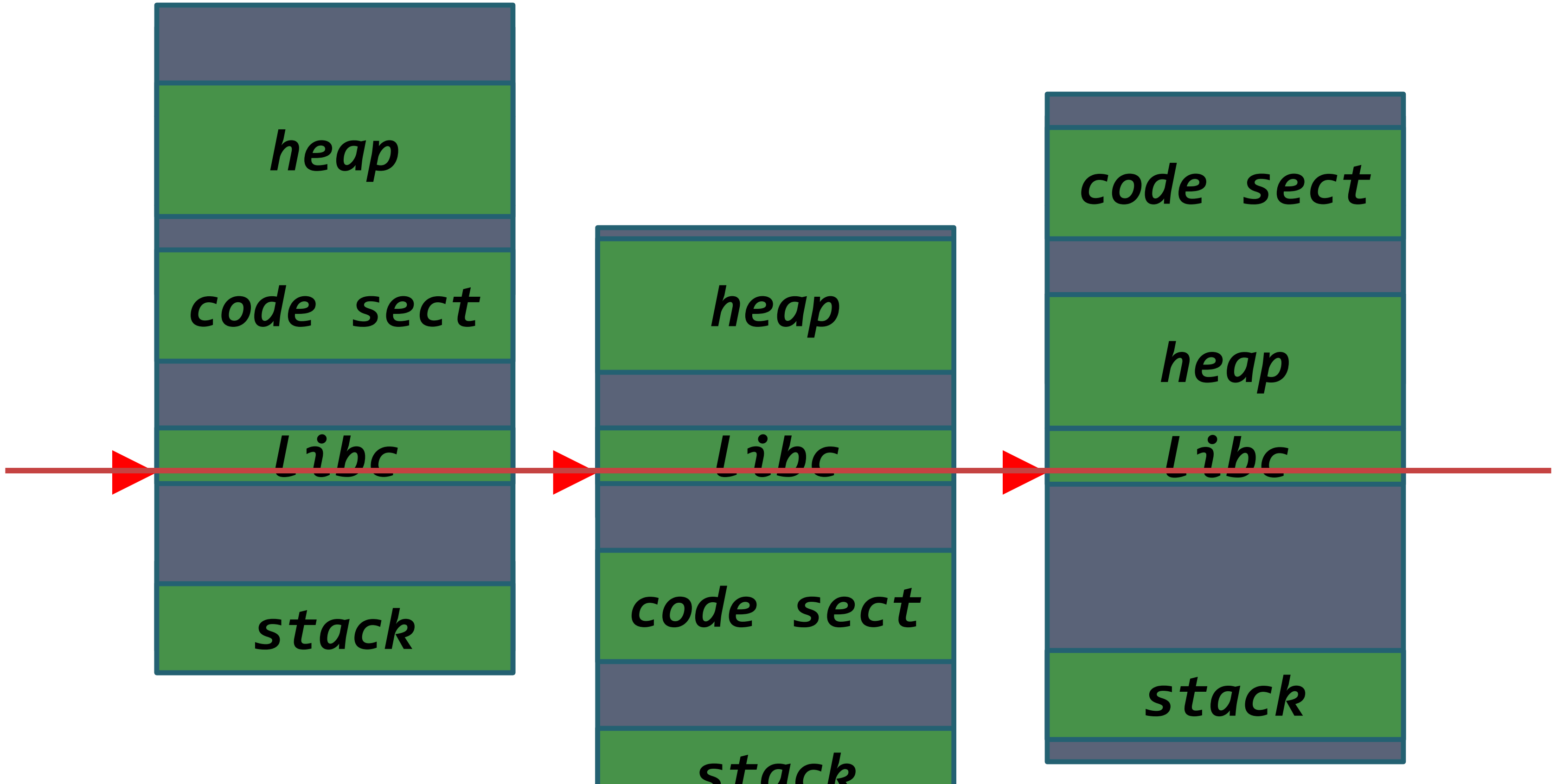
All of libc is at a single offset.

Over-read a single pointer in libc!

Memory Layout (with ASLR)



Memory Layout (with ASLR)



ASLR

Everything must be relocatable to be effective

A single code section that can be referenced may provide enough ROP gadgets for exploitation

Attacker may disclose the offset of an entire chunk!

Fine-grained ASLR shuffles code within the chunks.

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR

Stack Canaries

Stack Canaries

Defender:

Attackers keep overwriting return addresses!

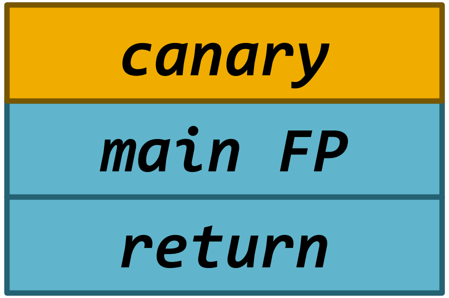
Defender:

We shall protect the return address!

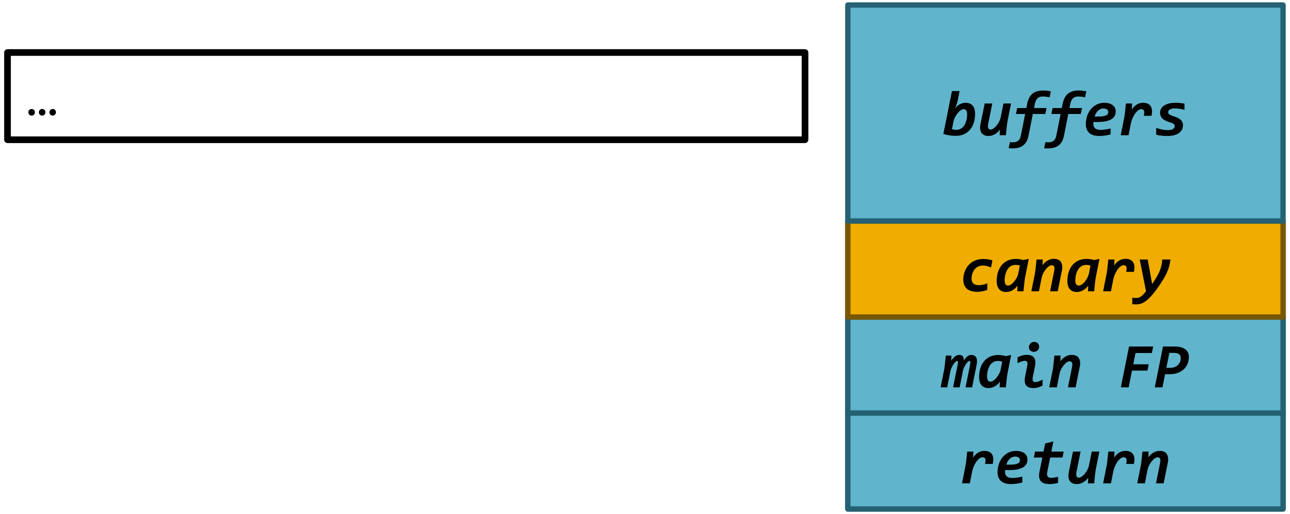
Keep a canary in the coal mine!

Stack Canaries

```
# on function call:

canary = secret
```

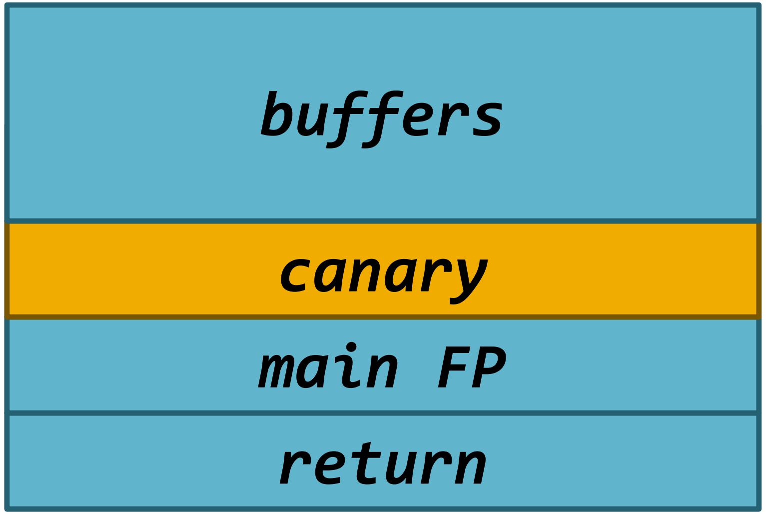


Stack Canaries



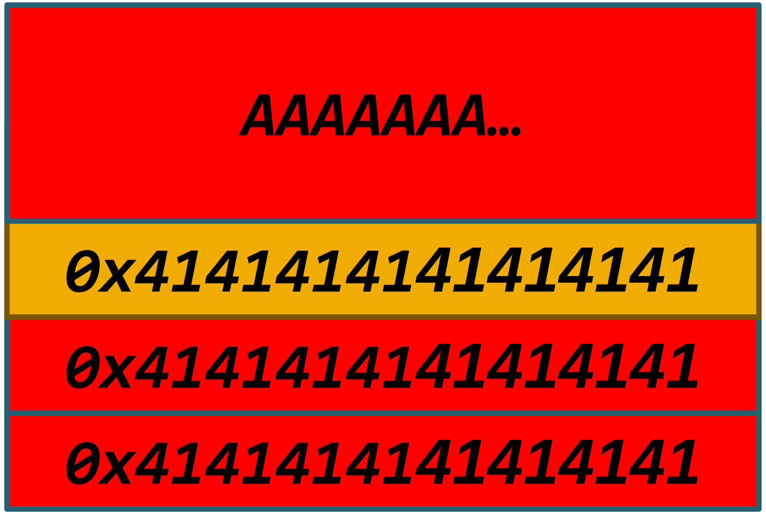
Stack Canaries

```
# vulnerability:

strcpy(buffer, str)
```



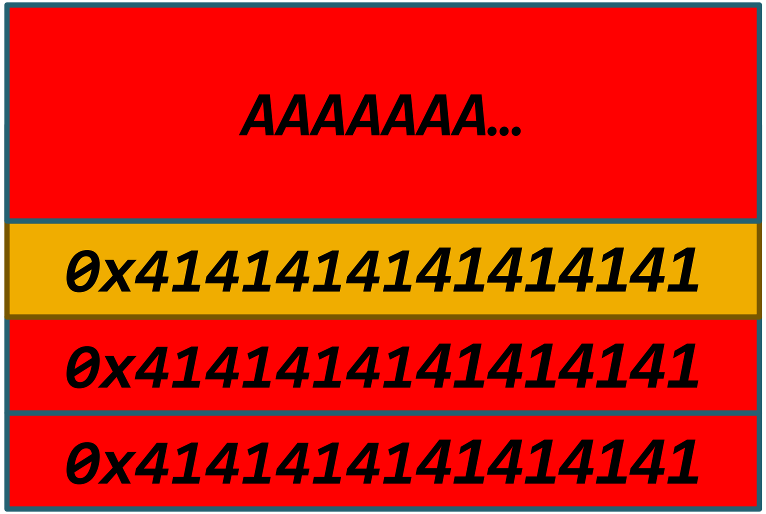
Stack Canaries

```
# vulnerability:

strcpy(buffer, str)
```



Stack Canaries

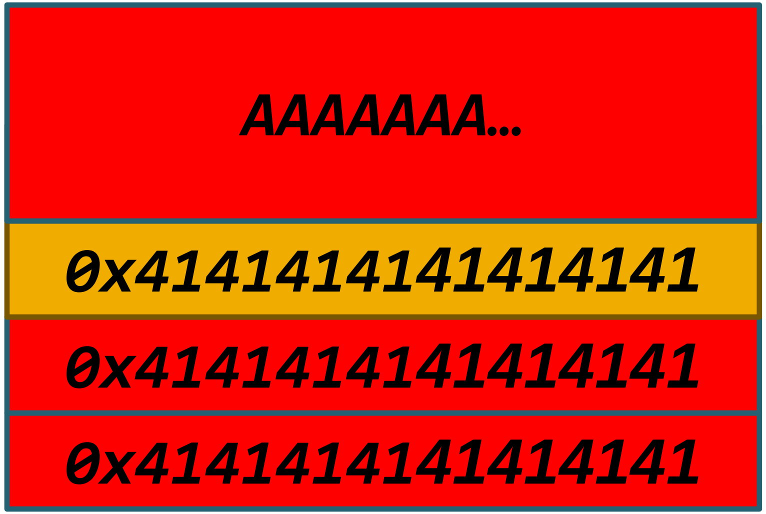
```
# on leave:

if canary != expected:
  goto stack_chk_fail
return
```



Stack Canaries

*** stack smashing detected ***

```
# on leave:

if canary != expected:
  goto stack_chk_fail
return
```



Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read

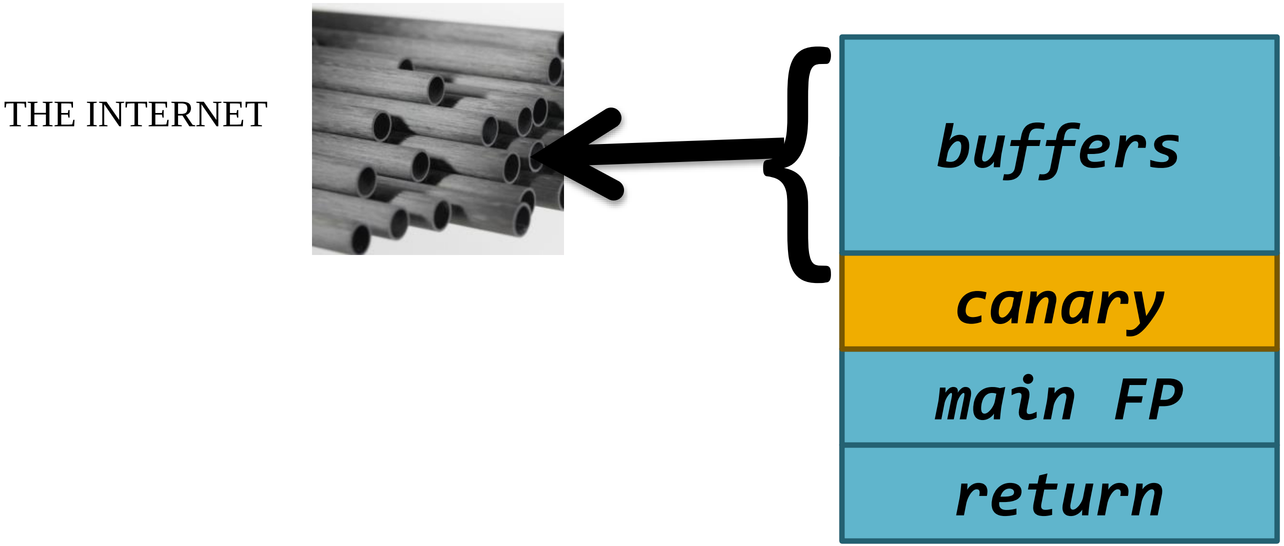
Buffer Over-read

```
int getField(int socket, char* field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  read(socket, field, fieldLen);
  return fieldLen;
}
```

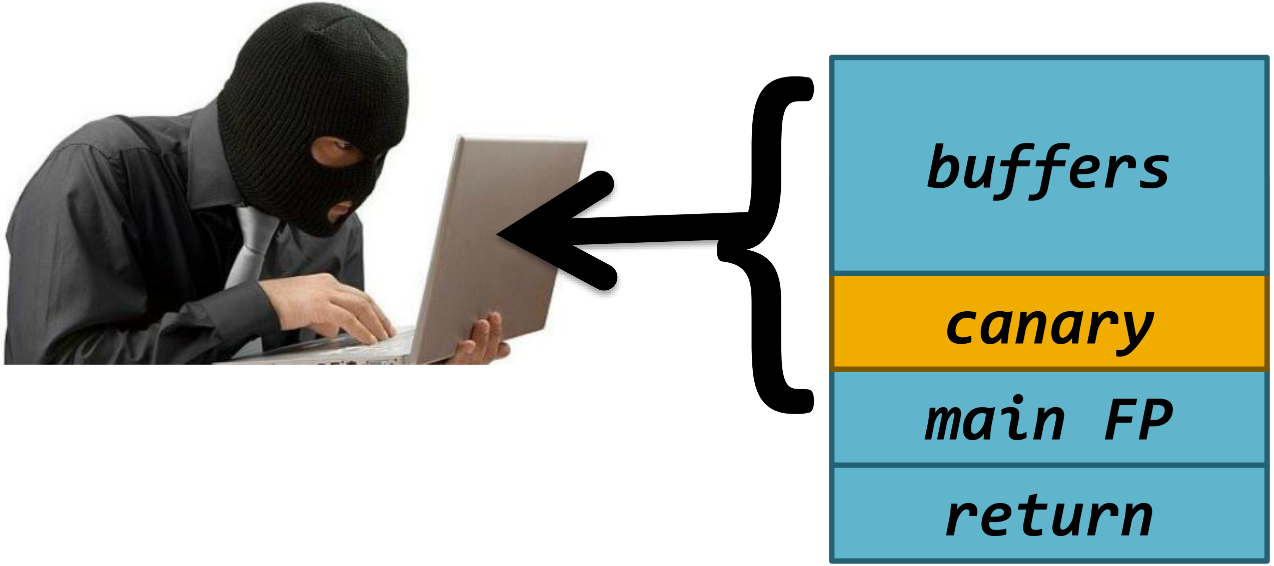
Buffer Over-read

```
int sendField(int socket, char* field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  write(socket, field, fieldLen);
  return fieldLen;
}
```

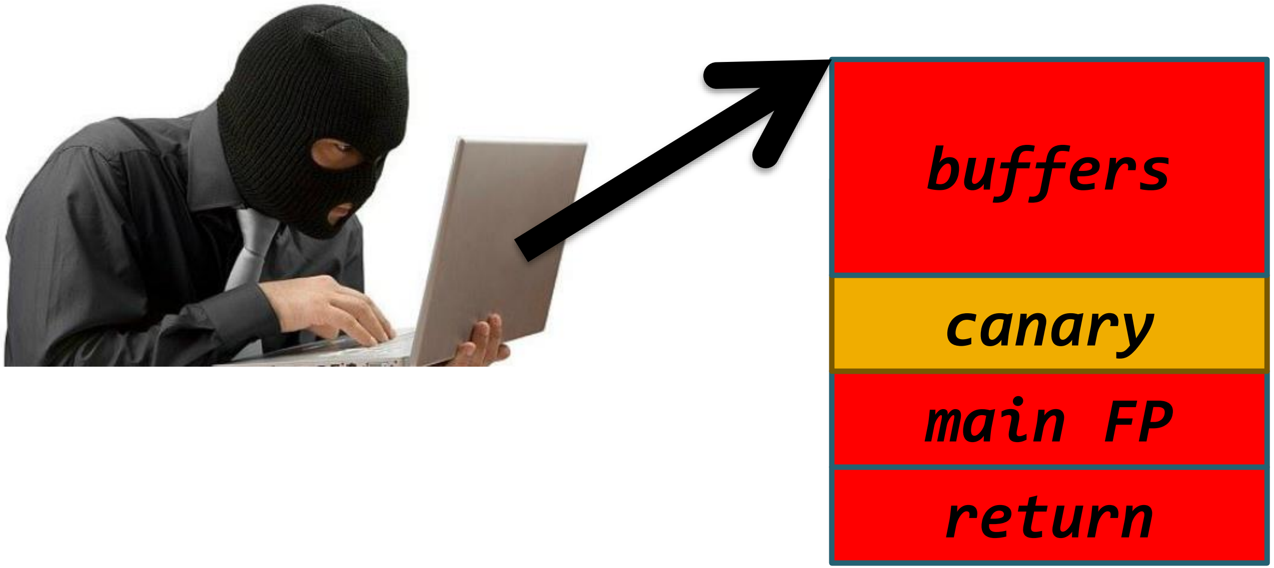
Buffer Over-read



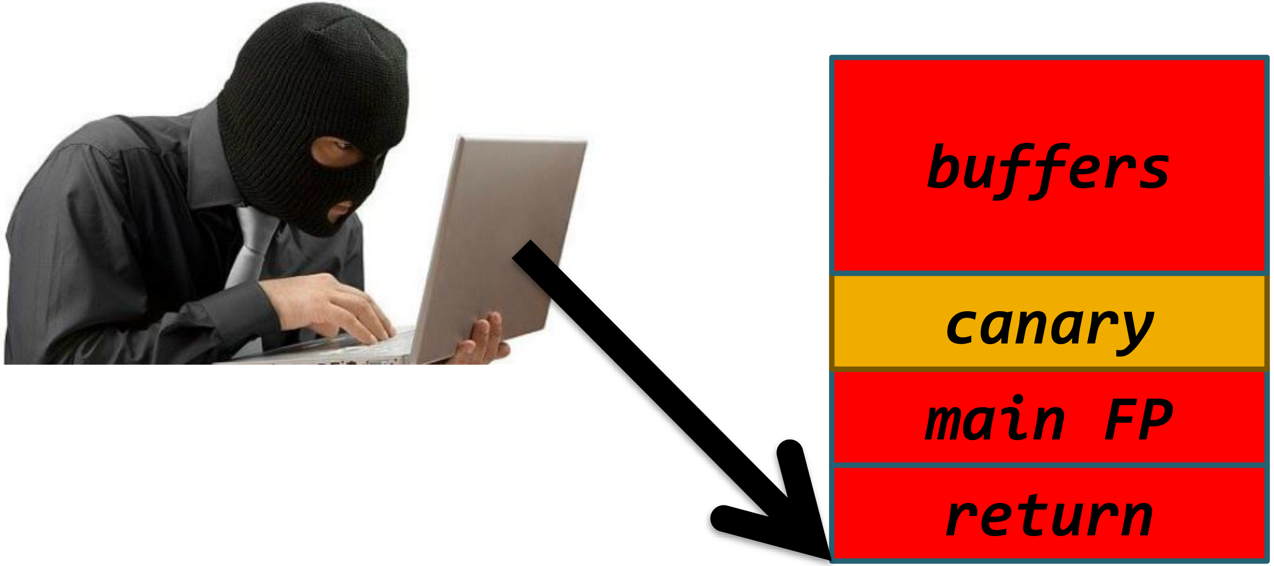
Buffer Over-read



Buffer Over-read

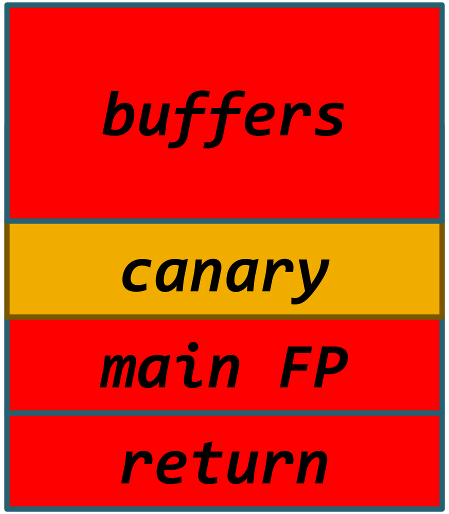


Buffer Over-read

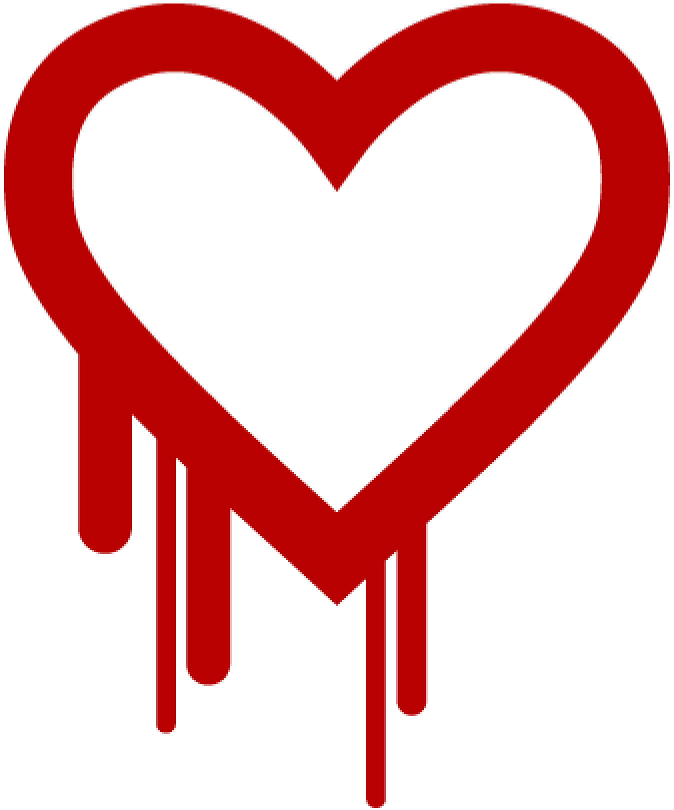


Buffer Over-read

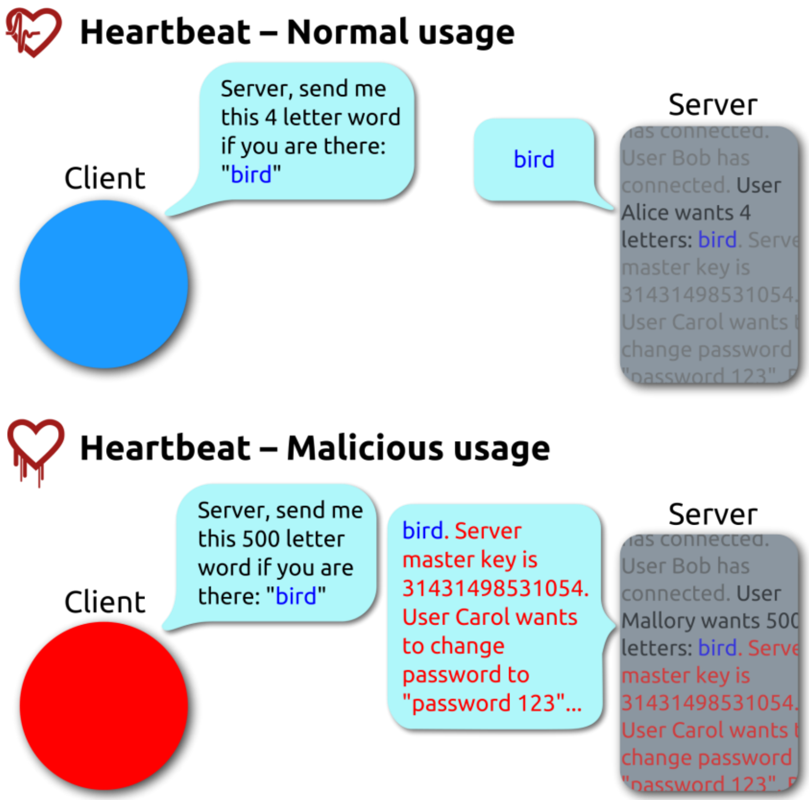
```
# on return:

if canary != expected:
  goto stack_chk_fail
return
```



Heartbleed



Heartbleed



Source: en.wikipedia.org/wiki/Heartbleed

…totally unrelated

Andrew Ayer
@__agwa

This will be OpenSSL's first "CRITICAL" vulnerability since 2016. Examples of "CRITICAL" vulnerabilities include "significant disclosure of the contents of server memory (potentially revealing user details), …

> **Mark J Cox** @iamamoose · Oct 25
> OpenSSL 3.0.7 update to fix Critical CVE out next Tuesday 1300-1700UTC. Does not affect versions before 3.0. mta.openssl.org/pipermail/open...

10:37 AM · Oct 25, 2022 · Twitter Web App

318 Retweets **35** Quote Tweets **498** Likes

Andrew Ayer @__agwa · Oct 25
Replying to @__agwa
… vulnerabilities which can be easily exploited remotely to compromise server private keys or where remote code execution is considered likely in common situations" 😱(openssl.org/policies/gener...)

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Integer Casting

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Integer Overflow

Integer Overflow

Unsafe:

- `strcpy` and friends (`str*`)
- `sprintf`
- `gets`

Use instead:

- `strncpy` and friends (`strn*`)
- `snprintf`
- `fgets`

Integer Overflow

Defender:

OK fine, let's replace `strcpy` with `str`**`n`**`cpy`

Attacker:

Find values of `n` that break `str`**`n`**`cpy`

Integer Overflow

```
void foo(int *array, int len) {
      int *buf;
      buf = malloc(len * sizeof(int));
      if (!buf)
            return;

      int i;
      for (i=0; i<len; i++) {
            buf[i] = array[i];
      }
}
```

Integer Overflow

```
void foo(int *array, int len) {
        int *buf;
        buf = malloc(len * sizeof(int));
        if (!buf)
                return;

        int i;
        for (i=0; i<len; i++) {
                buf[i] = array[i];
        }
}
```

What if len is very large?

Integer Overflow

len = 1,073,742,024 (≈ 1 billion)

`0x400000c8`

Integer Overflow

len = 1,073,742,024 (≈ 1 billion)

`0x400000c8`

len * 4 = 4,294,968,096 (≈ 4 billion)

`0x100000320`

Integer Overflow

len = 1,073,742,024 (≈ 1 billion)

`0x400000c8`

len * 4 = 4,294,968,096 (≈ 4 billion)

`0x100000320`

Cannot be represented in 32 bits

Integer Overflow

len = 1,073,742,024 (≈ 1 billion)

`0x400000c8`

len * 4 = 4,294,968,096 (≈ 4 billion)

`0x100000320`

Cannot be represented in 32 bits

`0x00000320` as uint32

Integer Overflow

len = 1,073,742,024 (≈ 1 billion)

`0x400000c8`

len * 4 = 4,294,968,096 (≈ 4 billion)

`0x100000320`

Cannot be represented in 32 bits

`0x00000320` as uint32

len * 4 = 800

Integer Overflow

```
void foo(int *array, int len) {
    int *buf;
    buf = malloc(len * sizeof(int));
    if (!buf)
        return;

    int i;
    for (i=0; i<len; i++) {
        buf[i] = array[i];
    }
}
```

size 800 buffer

Write ≈ 1 billion elements

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Signed/Unsigned Integers

Signed vs. Unsigned Integers

```
int sendField(int socket, char *field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  write(socket, field, fieldLen);
  return fieldLen;
}
```

Signed vs. Unsigned Integers

```
int sendField(int socket, char *field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  if (fieldLen > 10) {
    return; // Not this time :-D
  }
  write(socket, field, fieldLen);
  return fieldLen;
}
```

Signed vs. Unsigned Integers

Signed type!

Negative Number

```
int sendField(int socket, char *field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  if (fieldLen > 10) {
    return; // Not this time :-D
  }
  write(socket, field, fieldLen);
  return fieldLen;
}
```

Signed vs. Unsigned Integers

```
int sendField(int socket, char *field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  if (fieldLen > 10) {
    return; // Not this time :-D
  }
  write(socket, field, fieldLen);
  return fieldLen;
}
```

Signed type!

Negative Number

Passes Signed Check

Signed vs. Unsigned Integers

```
int sendField(int socket, char *field){
  int fieldLen = 0;
  read(socket, &fieldLen, 4);
  if (fieldLen > 10) {
    return; // Not this time :-D
  }
  write(socket, field, fieldLen);
  return fieldLen;
}
```

Signed type!

Negative Number

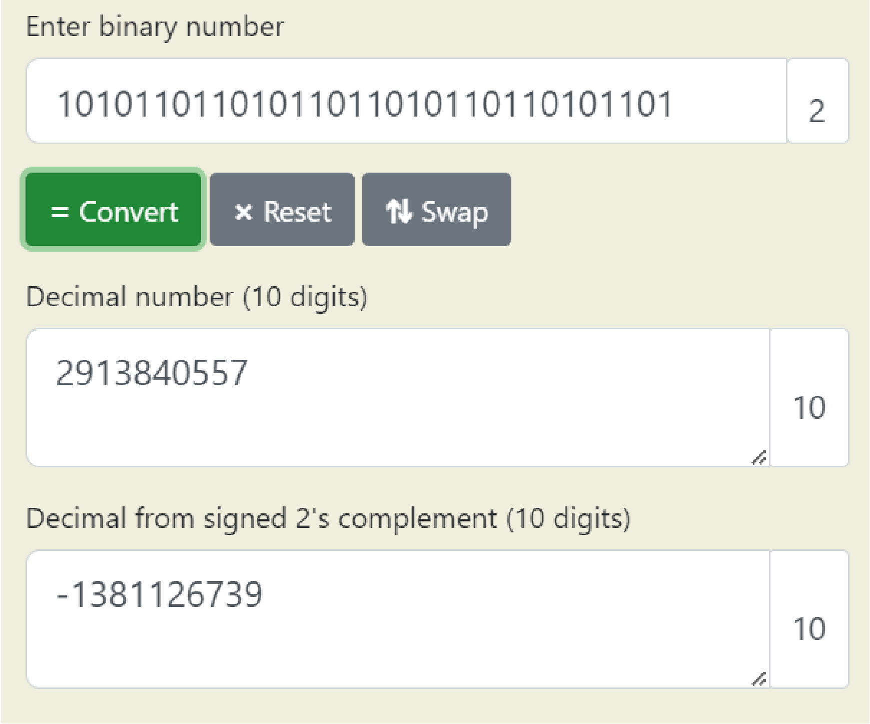
Passes Signed Check

Treated as a very large number
(third argument is type size_t, an unsigned integer)

Signed vs. Unsigned Integers

fieldLen as size_t

fieldLen as int



The SAME bits give us either a HUGE number (unsigned) or a TINY number (signed)
It depends on how the bits are *interpreted*

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Integer Casting

Automated Testing

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Integer Casting

Automated Testing

Automated Testing

Automated Testing

The Problem With Computers:

Vulnerabilities are hard to find by hand

Defender:

(and attacks use them ☹)

Attacker:

(and attacks use them ☺)

Defender and Attacker *(in unison)*:

Automate the process!

Automated Testing

Finding vulnerabilities manually is very hard

If source is available:

- Tons of potential vulnerabilities in code base

If closed source:

- Reverse Engineering is laborious

Automated Testing

Memory Analysis Tools

Incredibly useful for finding memory leaks

Execute in a virtual environment

& perform dynamic run-time checks

Does the program access uninitialized memory?

Does the program use memory after it’s freed?

Automated Testing

Static Analysis Tools

- Look for dangerous coding patterns and practices
- Usually requires complete source code
- Large number of false-positives

- Are integers mixing signed and unsigned usage?
- Are all variables initialized when declared?

Automated Testing

Taint Analysis Tools

Trace value usage throughout code

Attempt to identify when untrusted data is used

Is a user-supplied value used to index an array?

Is an unsafe value used to shell-out?

Automated Testing

Fuzzers

“Brute Force Testing”

Generate inputs and monitor program’s behavior

More advanced optimize for code coverage

If I give you really long strings, will you crash?

If I give you random data, will you crash?

If I give you broken formats, will you crash?

Cat-and-Mouse Exploitation

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Extraneous function removal

ROP

ASLR
Stack Canaries

Buffer Over-read
Integer Casting

Automated Testing

Toolbox of Exploitation Techniques

Toolbox of Exploitation Techniques

Every vulnerability is different

- Some are not exploitable at all

Sometimes it takes multiple bugs to create an exploit ("Bug Chains")

- Buffer over-read (canary) + Buffer over-read (ASLR reference) + Buffer overflow (load exploit) + ROP chain (disable DEP) + Jump to shellcode

Taking the Easy Road

Don’t overly complicate the exploit

Is there an n-day?

Can you exploit a function without canaries?

Can you pivot from another application?

Can you brute-force a canary?

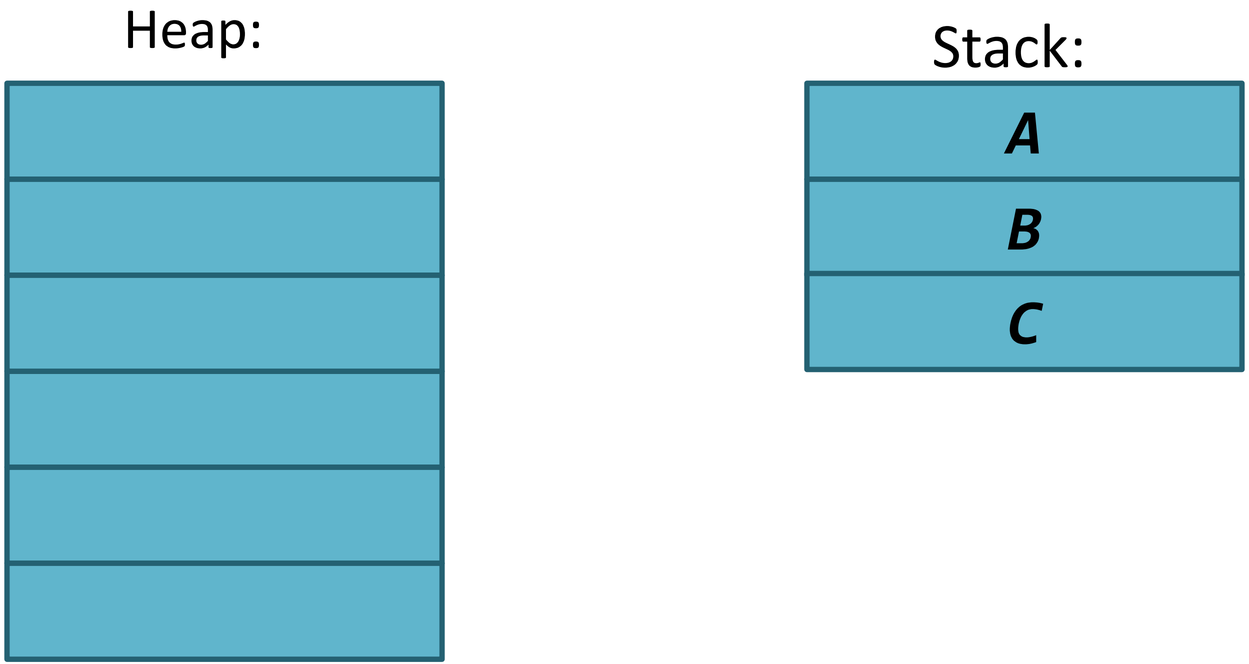
Use After Free

Common in multi-threaded programs that share variables

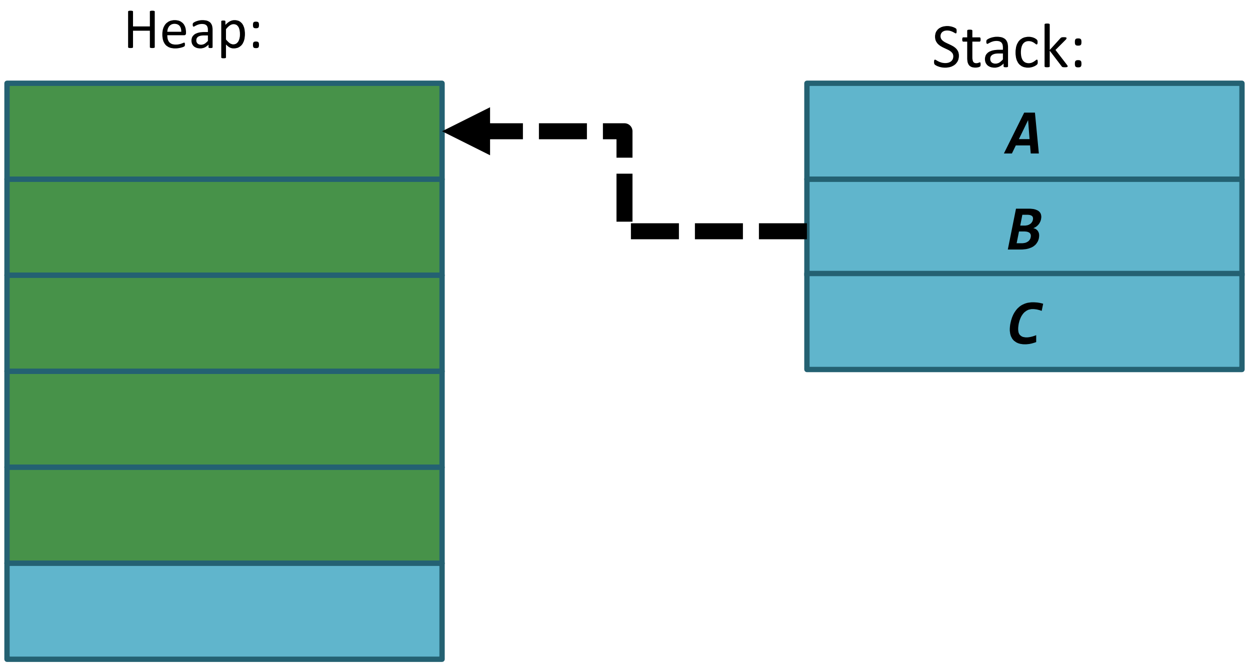
- Though can exist in single threaded programs

Sometimes caused by a race condition

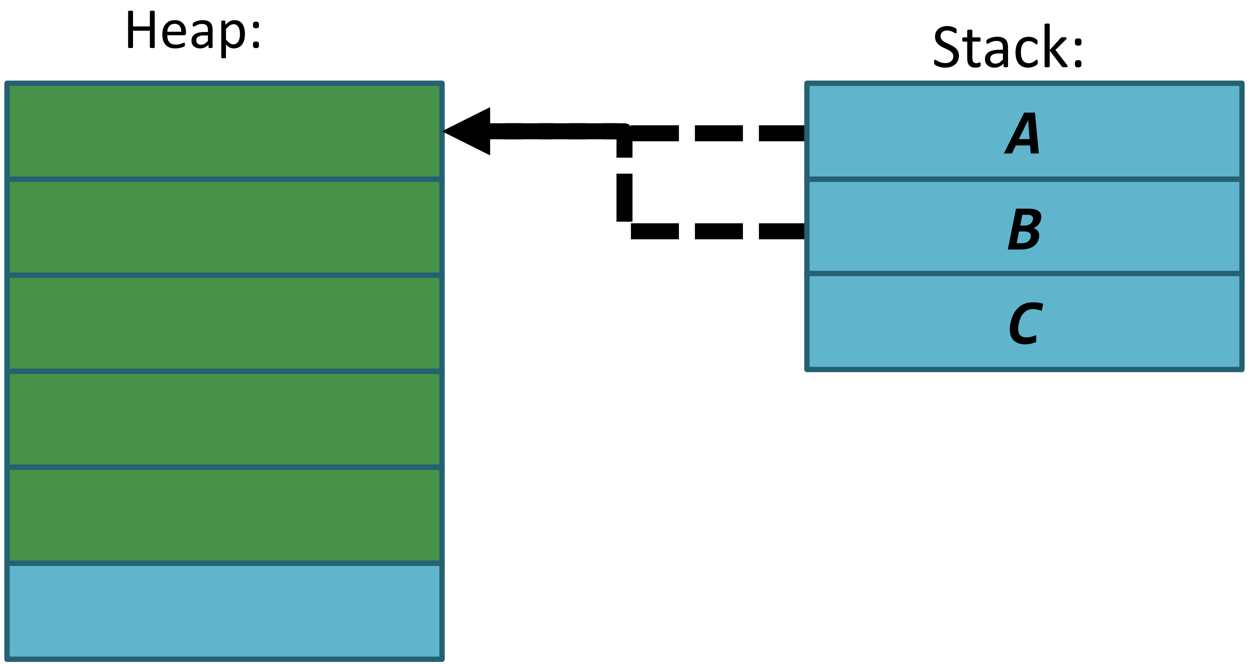
Use After Free



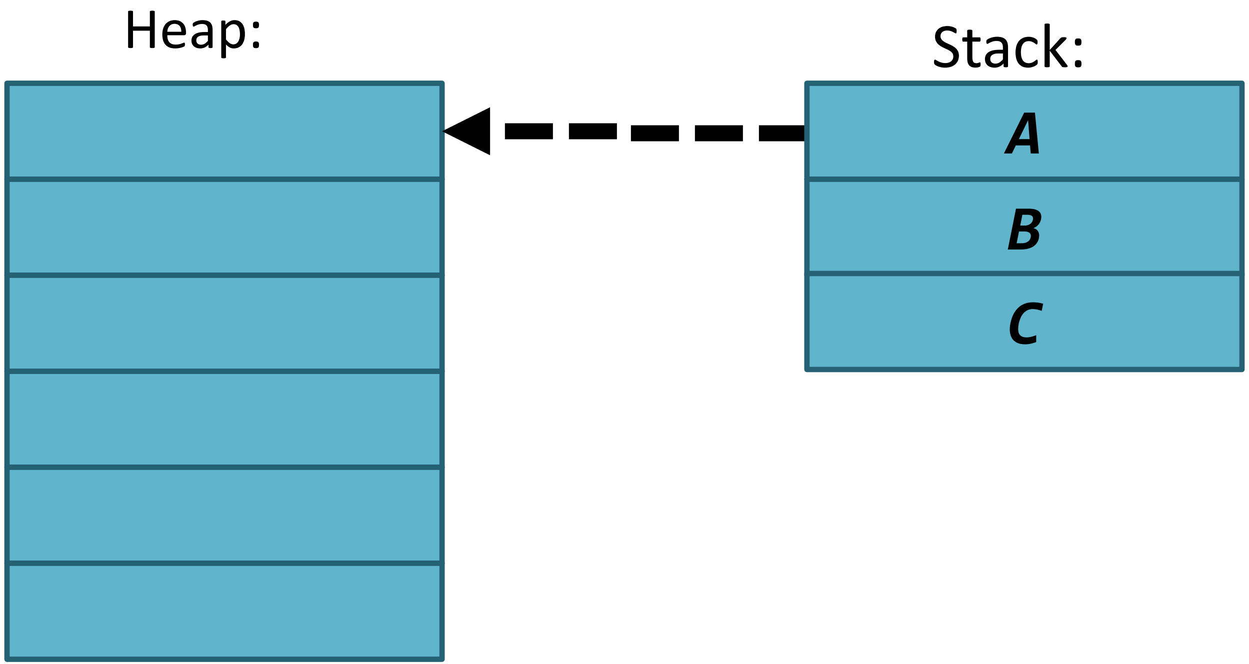
Use After Free



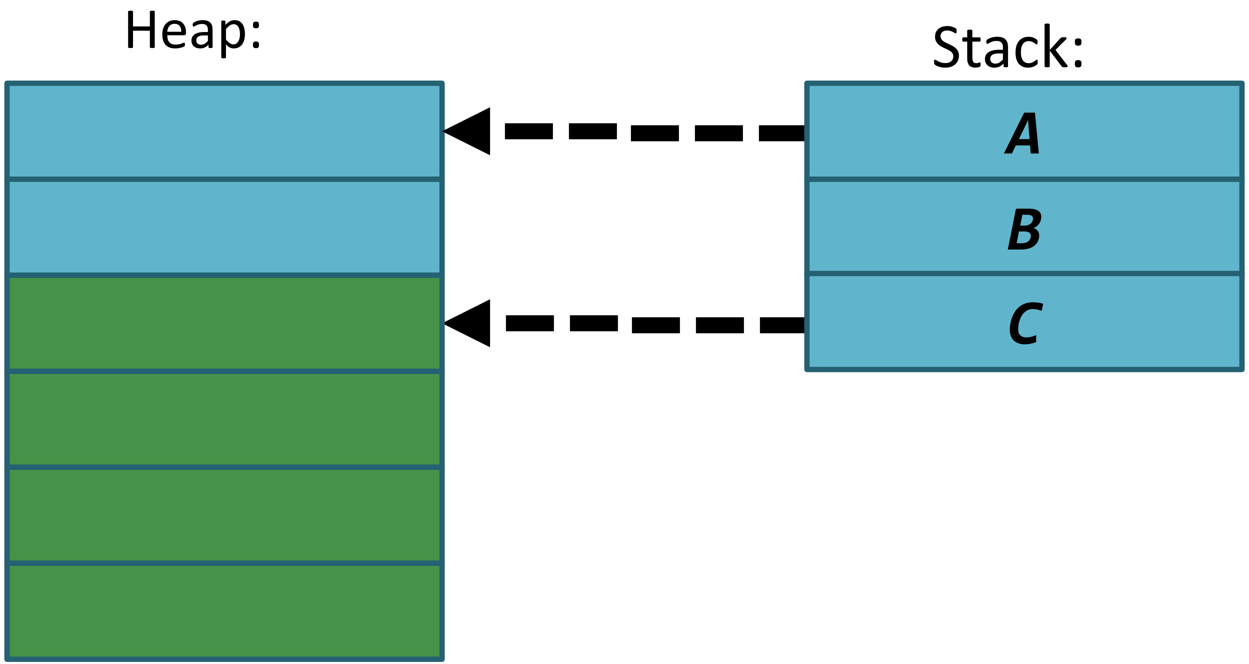
Use After Free



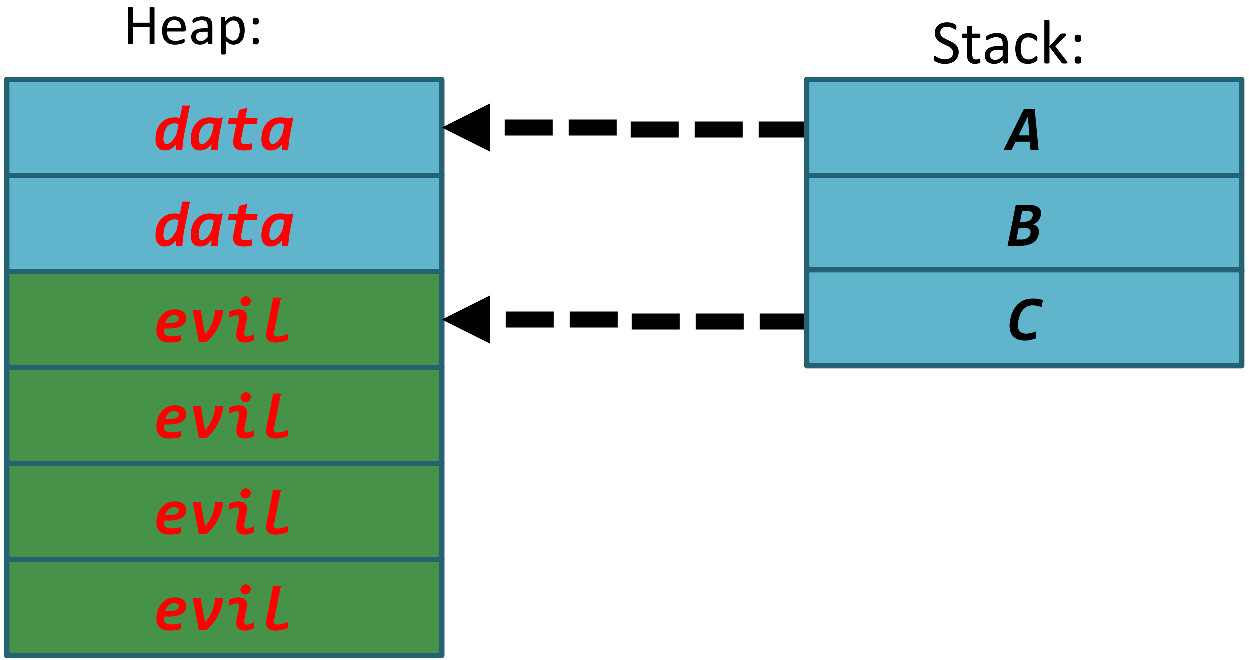
Use After Free



Use After Free



Use After Free



Format String Vulnerability

Attack programmer's lack of sanitization

```
printf("%s\n", argv[1]);
```

```
printf(argv[1]);
```

Number of arguments determined by the format of the string

```
./a.out "%p %p %p %p %p %p"
```

Further Reading

- "Smashing the Stack 21st Century" https://thesquareplanet.com/blog/smashing-the-stack-21st-century/
- Blexim's "Basic Integer Overflows" http://www.phrack.org/issues.html?issue=60&id=10
- ir0nstone's "Binary Exploitation Notes" https://ir0nstone.gitbook.io/notes/
- Aleph One's "Smashing the Stack for Fun and Profit" (uses 32-bit instead of x64, still a classic!) http://insecure.org/stf/smashstack.html

Note: further reading may not use x86-64 with Intel syntax, so be mindful of differences

Coming Up

Project 4 due November 14 at 6 p.m.

Lab 4 due October 31 at 6 p.m. (next Thursday)

Tuesday, Oct. 29

Malware

Thursday, Oct. 31

Access Control and Isolation