Control Hijacking (day 2)

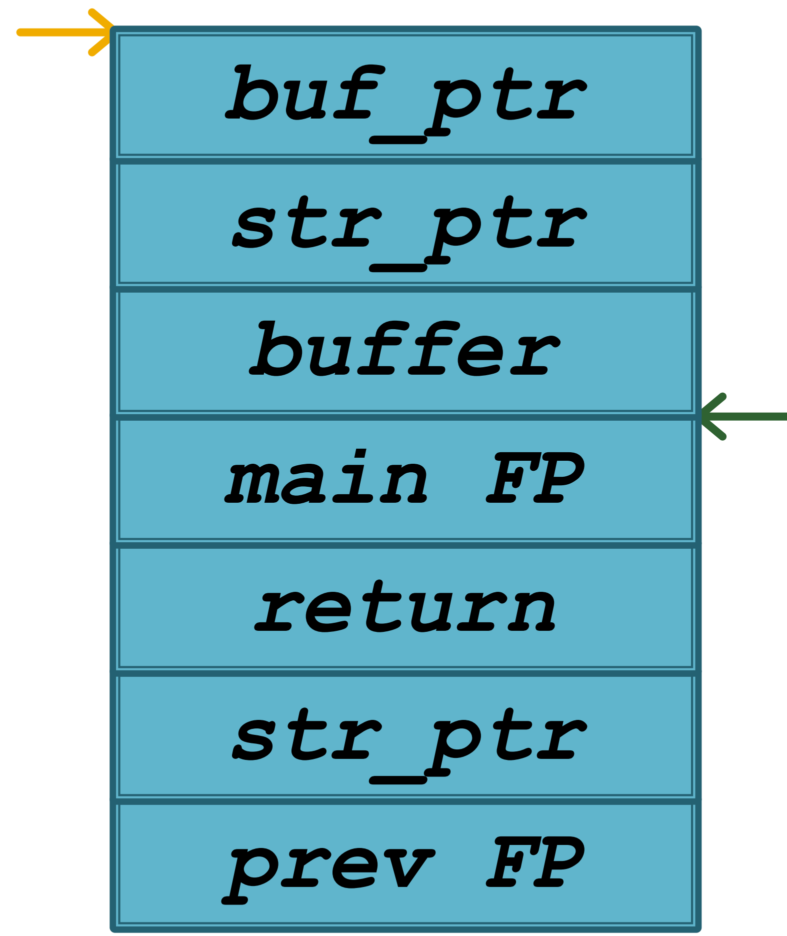
EECS 388: Introduction to Computer Security

Ben VanderSloot

*Based on slides by Eric Wustrow, Travis Finkenauer, and Drew Springall

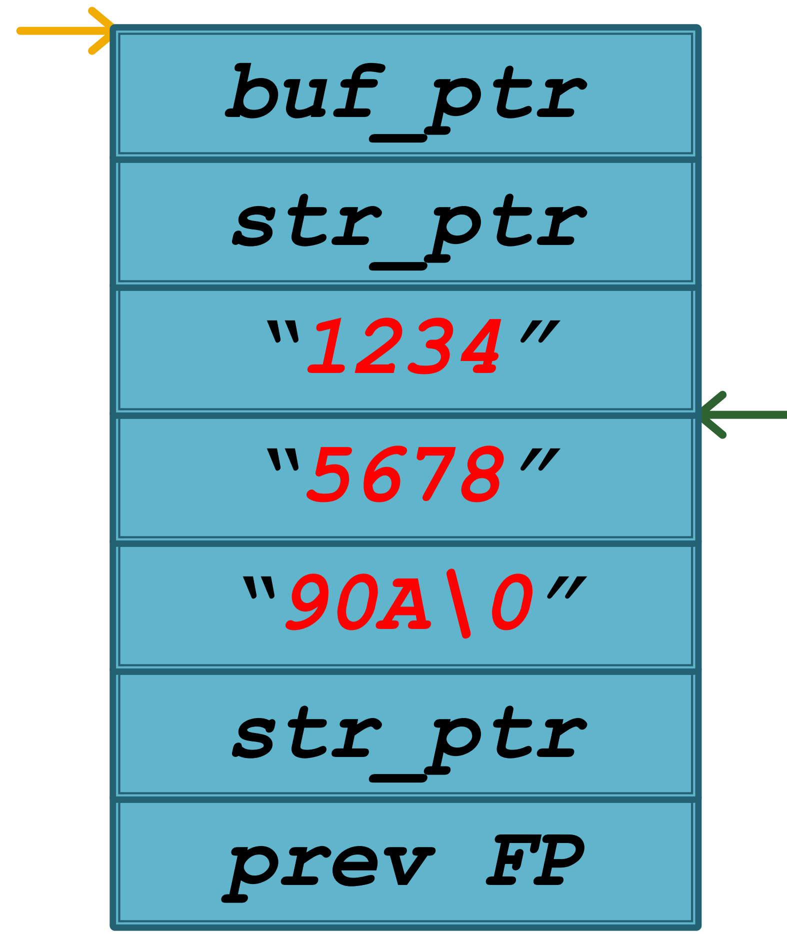
example.s (x86)

```
foo:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890A"
```

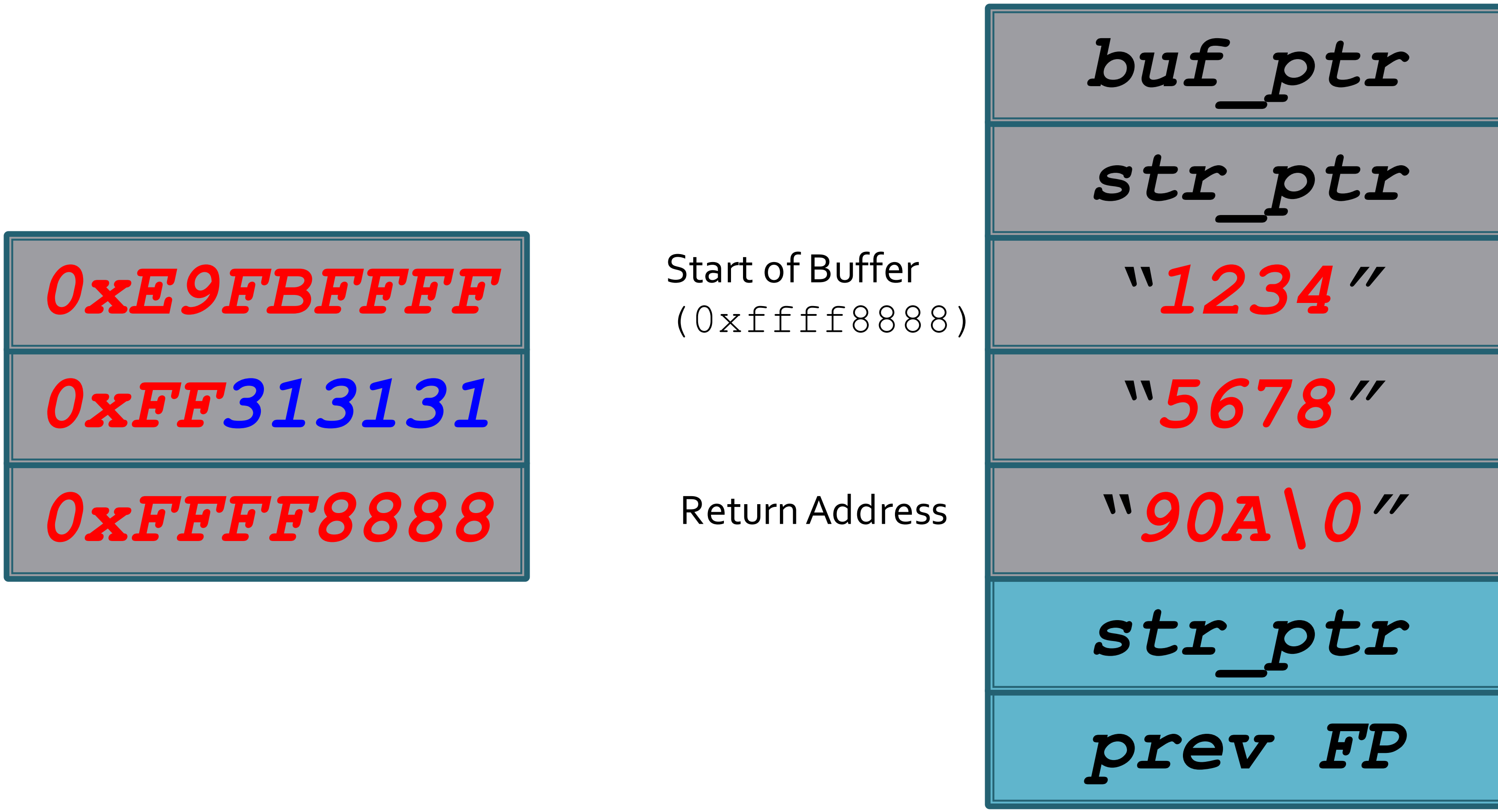


example.s (x86)

```
foo:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   ebp - 4
   call   strcpy
   leave
   ret
str_ptr: "1234567890A"
```

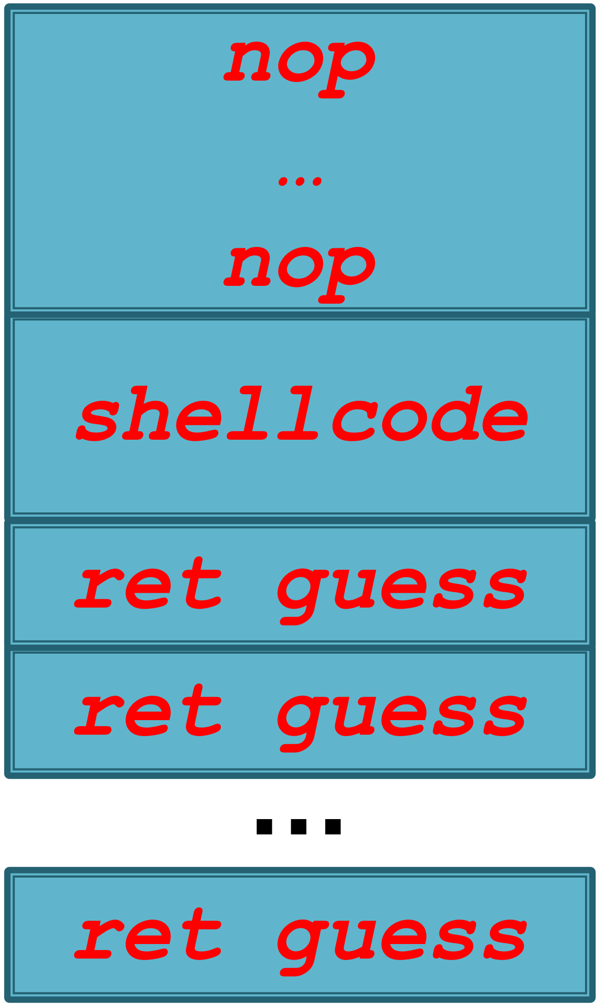


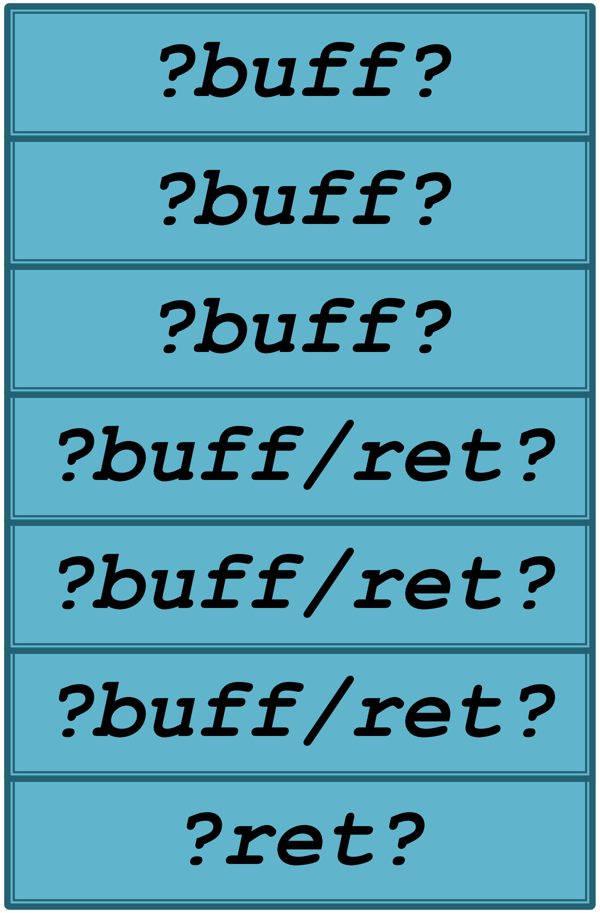
Stack Shellcode




```
b: e9 fb ff ff ff          jmp    b <_main+0xb>
```

Hard to guess address



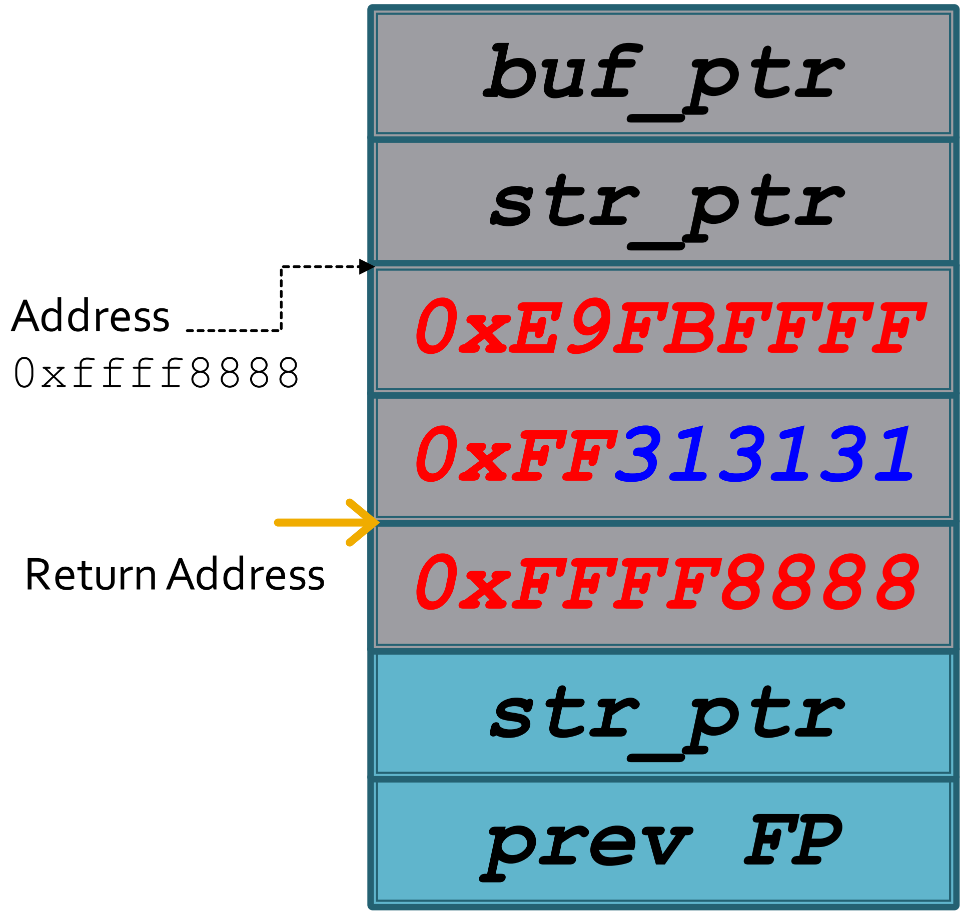


DEP

OS ERROR

CONTROL FLOW
IS INCORRECT

IMMEDIATELY
END PROCESS



Homework (not really)

Compile and read real assembly

```
gcc test.c -S -masm=intel -m32
```

Skim through a non-trivial program's source

```
objdump -d -M intel <executable>
```

How can you leverage an uncontrolled write?

How can you leverage control of EIP?

Homework (not really)

```
int main() {
  for (int  i=0; i<20; i++) {
    int* arr = malloc(16);
    arr[0] = i;
    arr[1] = i;
    arr[2] = i;
    arr[3] = i;
  }
}
```

Lubuntu 16.04 Appsec_1 [Running]

ubuntu@appsec-vm: ~/temp

File Edit Tabs Help

```
main:
.LFB2:
        .cfi_startproc
        lea     ecx, [esp+4]
        .cfi_def_cfa 1, 0
        and     esp, -16
        push    DWORD PTR [ecx-4]
        push    ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        mov     ebp, esp
        push    ecx
        .cfi_escape 0xf,0x3,0x75,0x7c,0x6
        sub     esp, 20
        mov     DWORD PTR [ebp-16], 0
        jmp     .L2
.L3:
        sub     esp, 12
        push    16
        call    malloc
        add     esp, 16
        mov     DWORD PTR [ebp-12], eax
        mov     eax, DWORD PTR [ebp-12]
        mov     edx, DWORD PTR [ebp-16]
        mov     DWORD PTR [eax], edx
        mov     eax, DWORD PTR [ebp-12]
        lea     edx, [eax+4]
        mov     eax, DWORD PTR [ebp-16]
        mov     DWORD PTR [edx], eax
        mov     eax, DWORD PTR [ebp-12]
        lea     edx, [eax+8]
        mov     eax, DWORD PTR [ebp-16]
        mov     DWORD PTR [edx], eax
        mov     eax, DWORD PTR [ebp-12]
        lea     edx, [eax+12]
                                                                  6,1           20%
```

ubuntu@appse... 17:54 Left ⌘

Homework (not really)

Compile and read real assembly

```
gcc test.c -S -masm=intel -m32
```

Skim through a non-trivial program's source

```
objdump -d -M intel <executable>
```

How can you leverage an uncontrolled write?

How can you leverage control of EIP?

Lubuntu 16.04 Appsec_1 [Running]

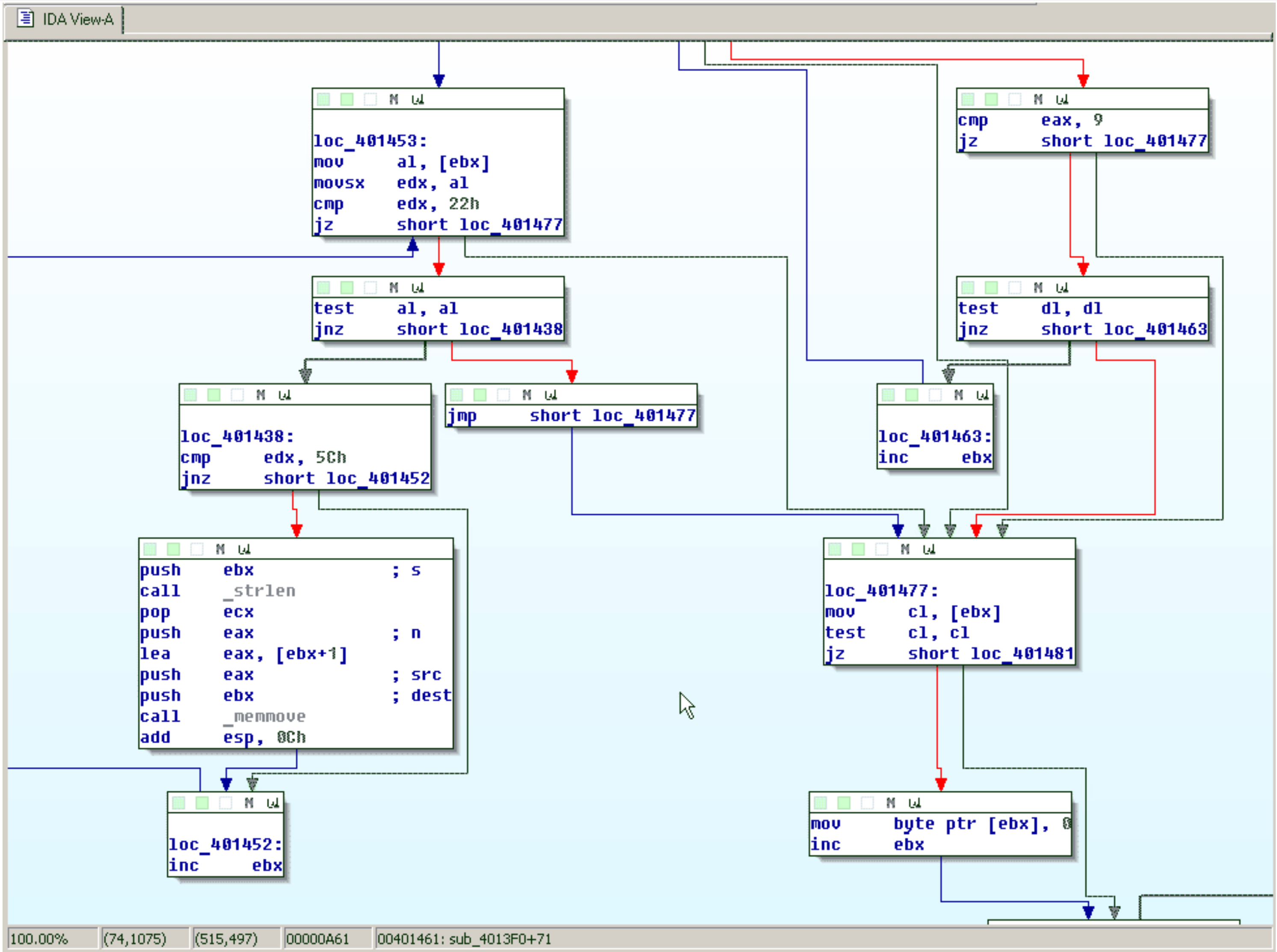
ubuntu@appsec-vm: /bin

File Edit Tabs Help

```
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%
```

ubuntu@appse... 17:21 Left ⌘

Homework (not really)



Homework (not really)

Compile and read real assembly

```
gcc test.c -S -masm=intel -m32
```

Skim through a non-trivial program's source

```
objdump -d -M intel <executable>
```

How can you leverage an uncontrolled write?

How can you leverage control of EIP?

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

Return to libc

Problem:

DEP prevents executing injected shellcode

Solution:

Reuse code that already exists

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

Make a **`ret`** behave like a **`call`**

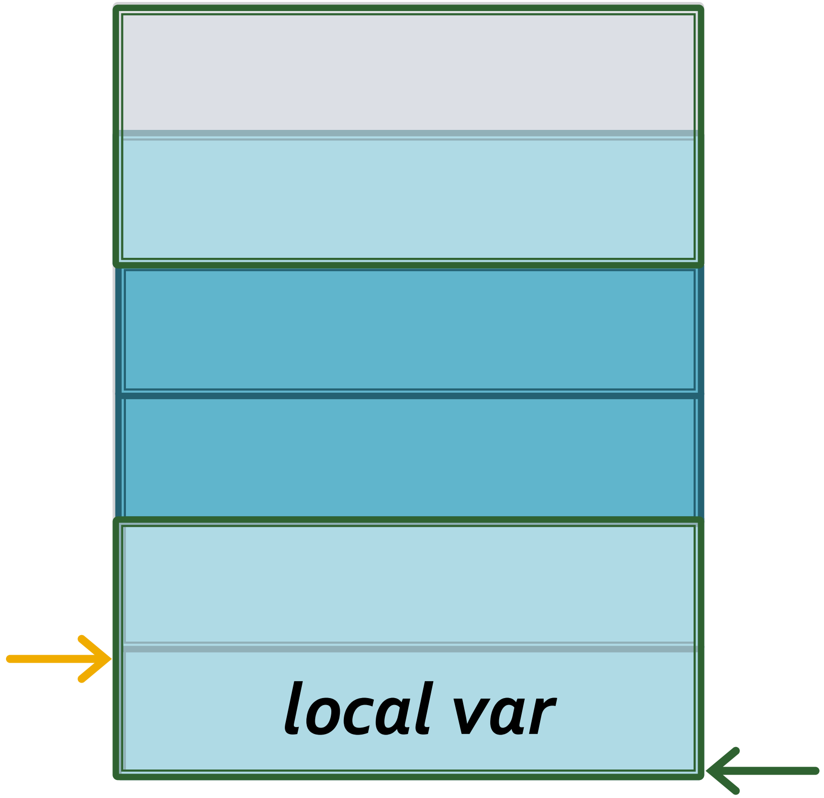
```
ret:
   pop eip
```

```
call:
   push eip + n
   jump foo
   ; mov eip, foo
```

What are the contents of the stack?

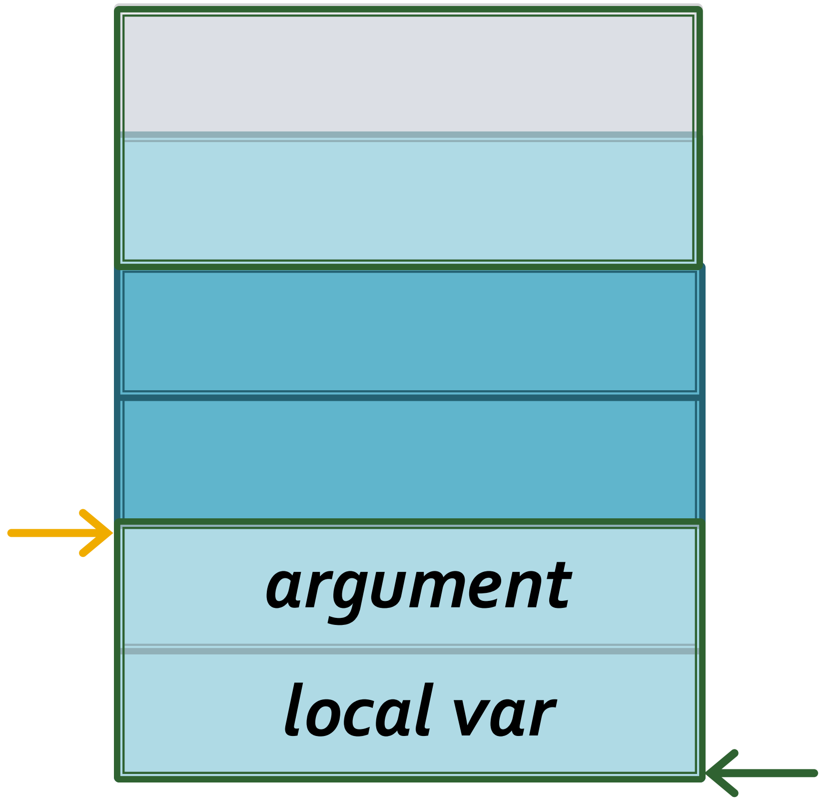
Return to libc

SETUP AS A FUNCTION CALL



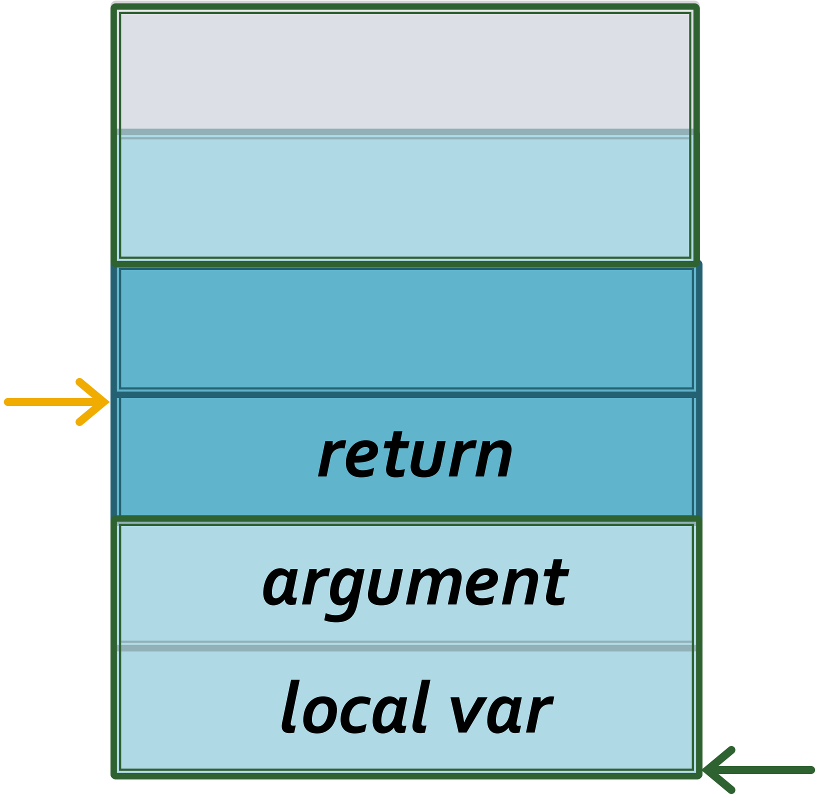
Return to libc

SETUP AS A FUNCTION CALL



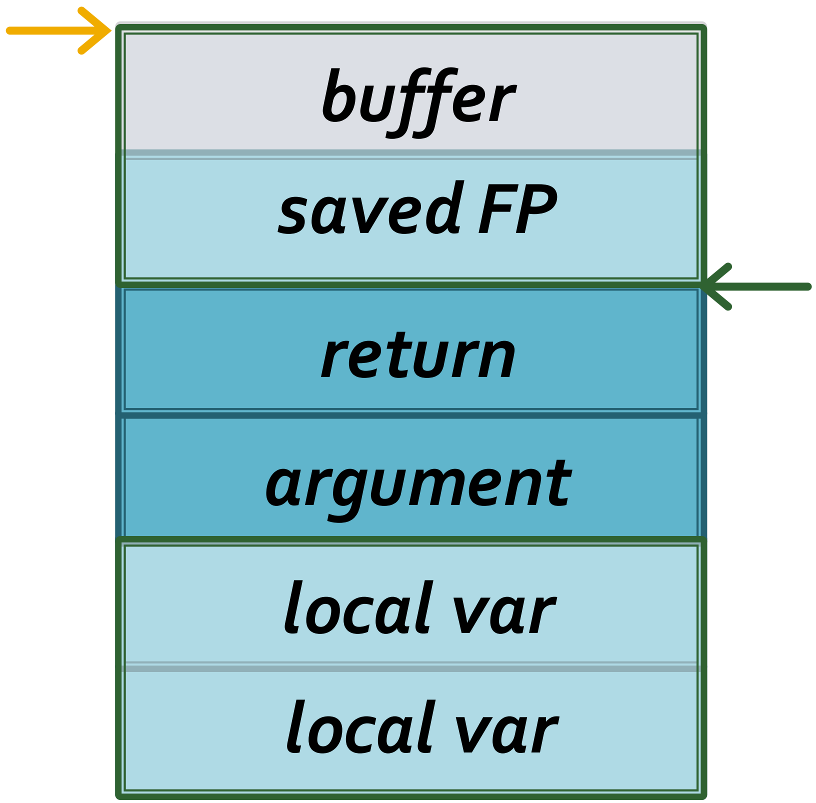
Return to libc

SETUP AS A FUNCTION CALL

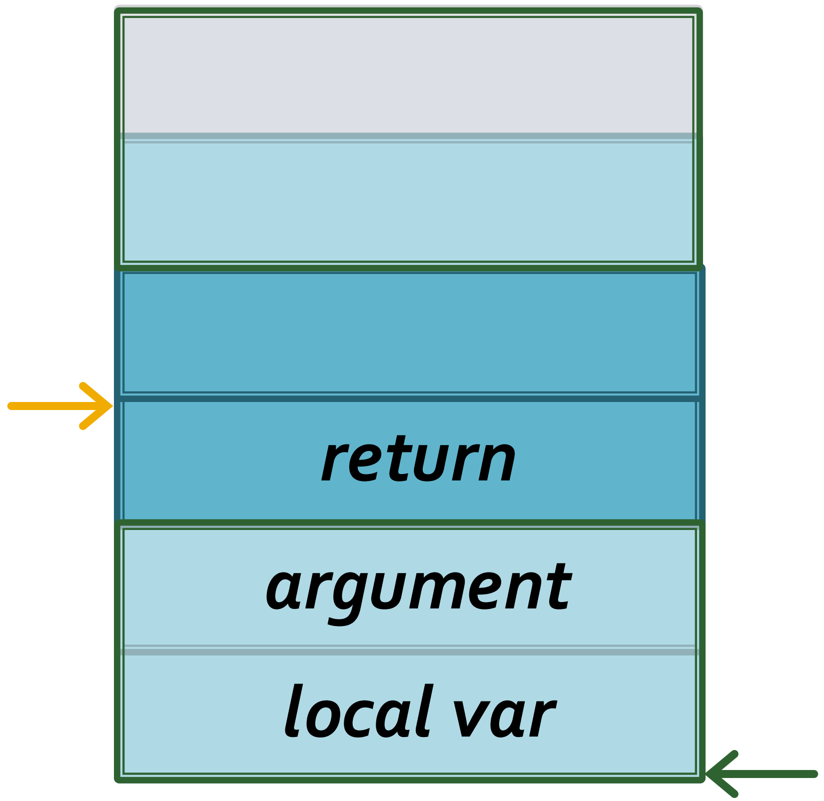


Return to libc

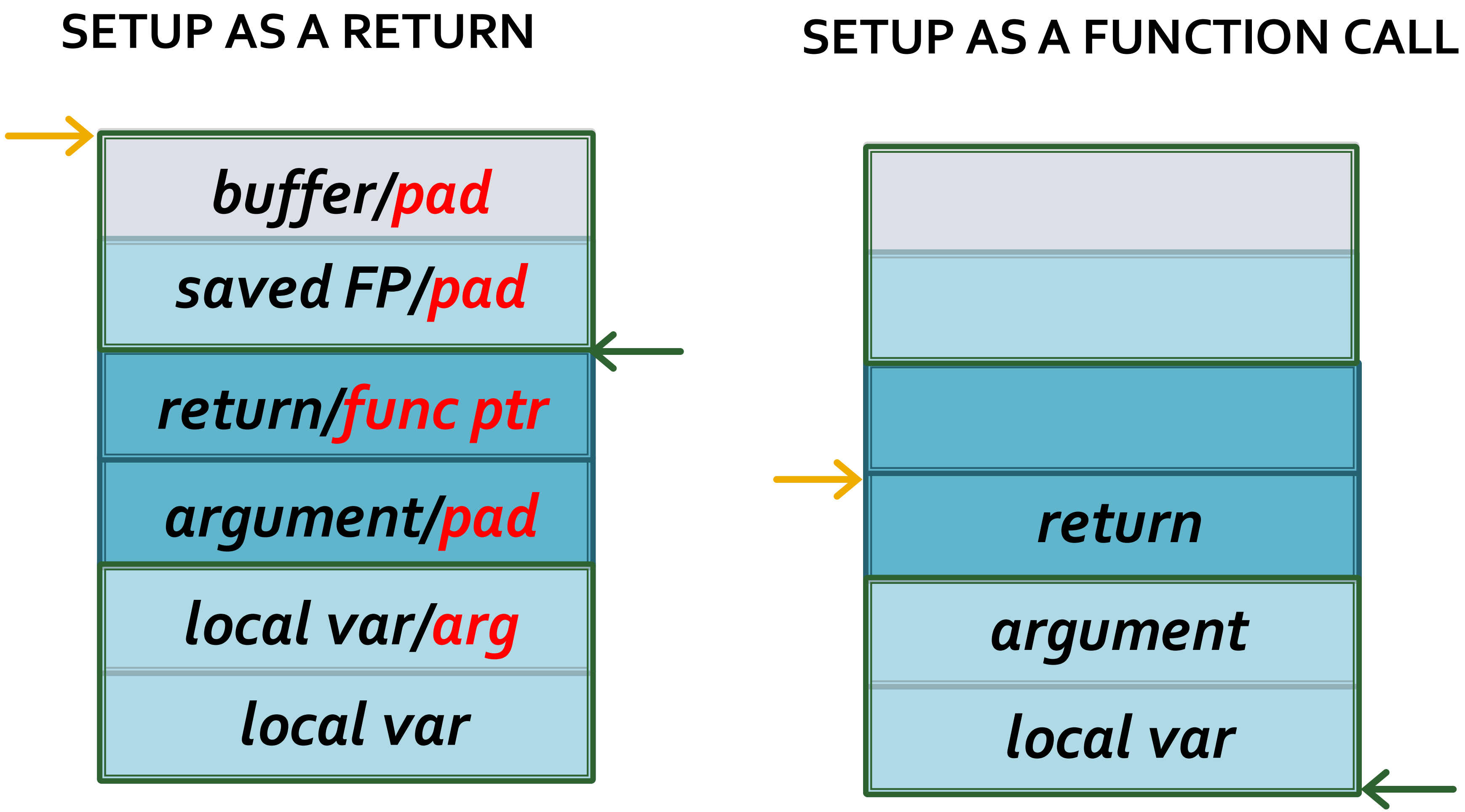
SETUP AS A RETURN



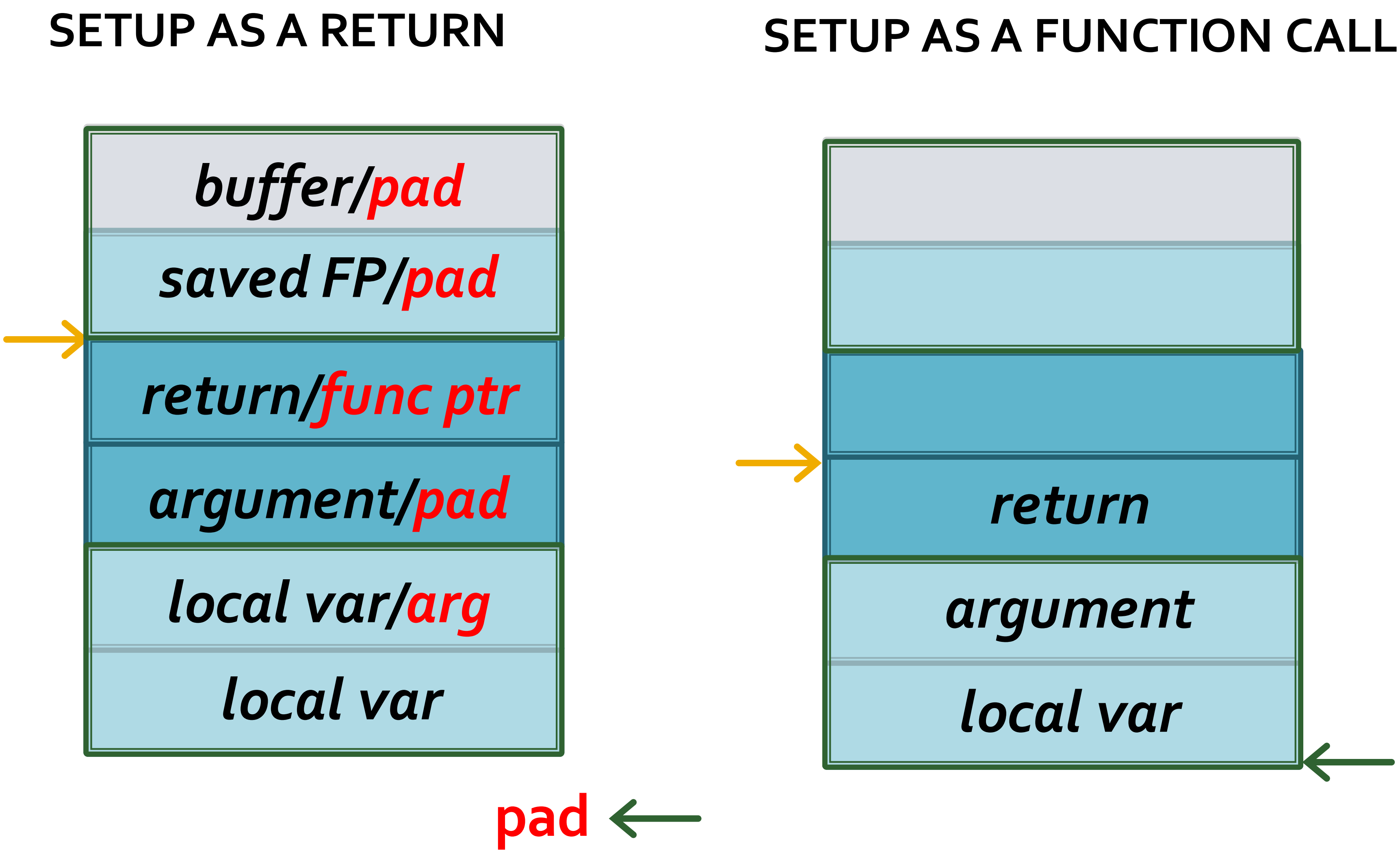
SETUP AS A FUNCTION CALL



Return to libc

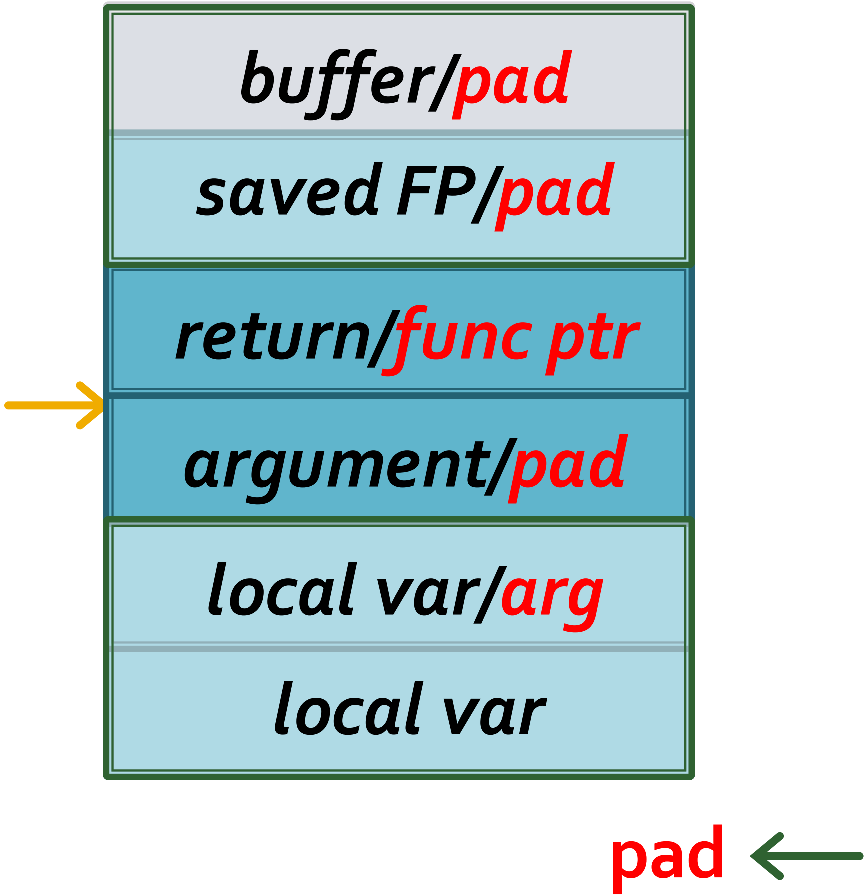


Return to libc

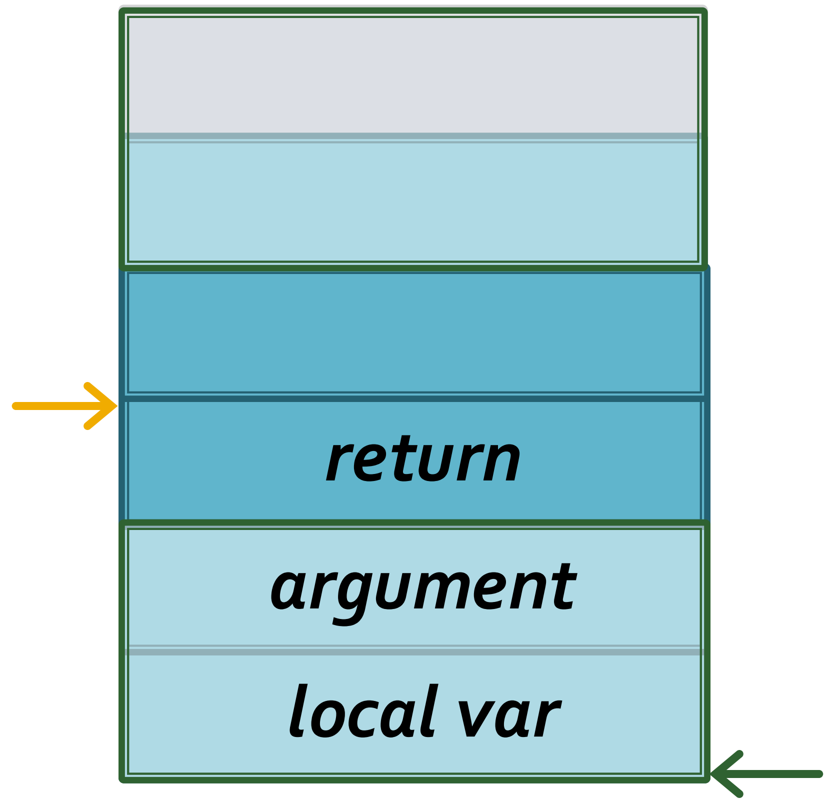


Return to libc

SETUP AS A RETURN

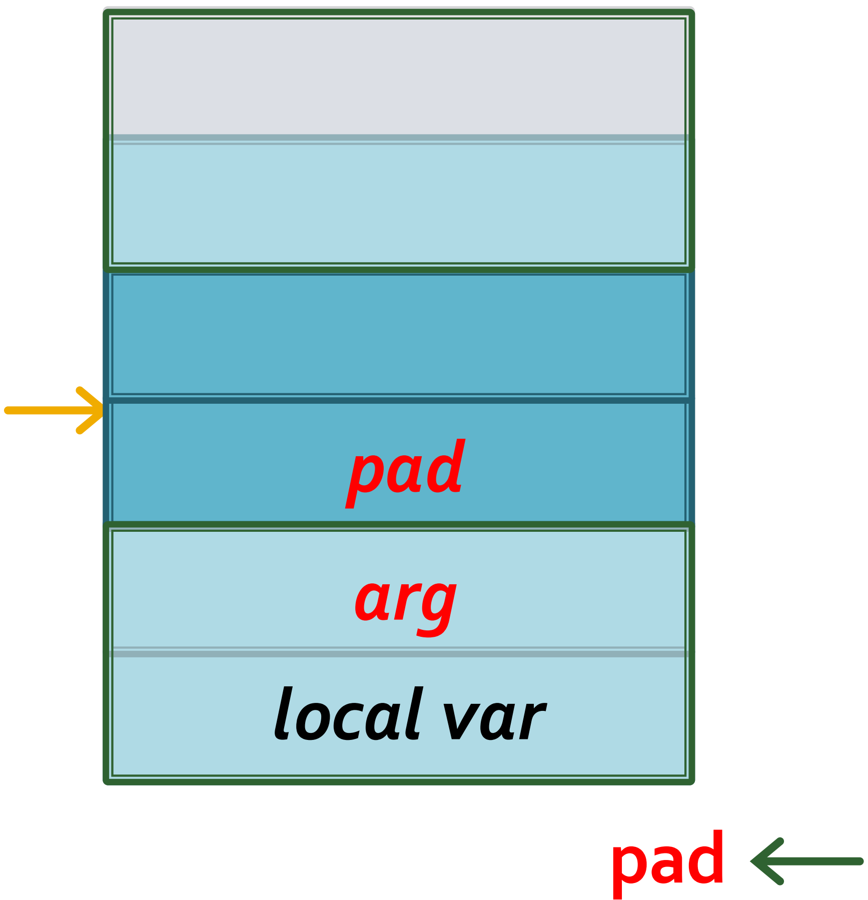


SETUP AS A FUNCTION CALL

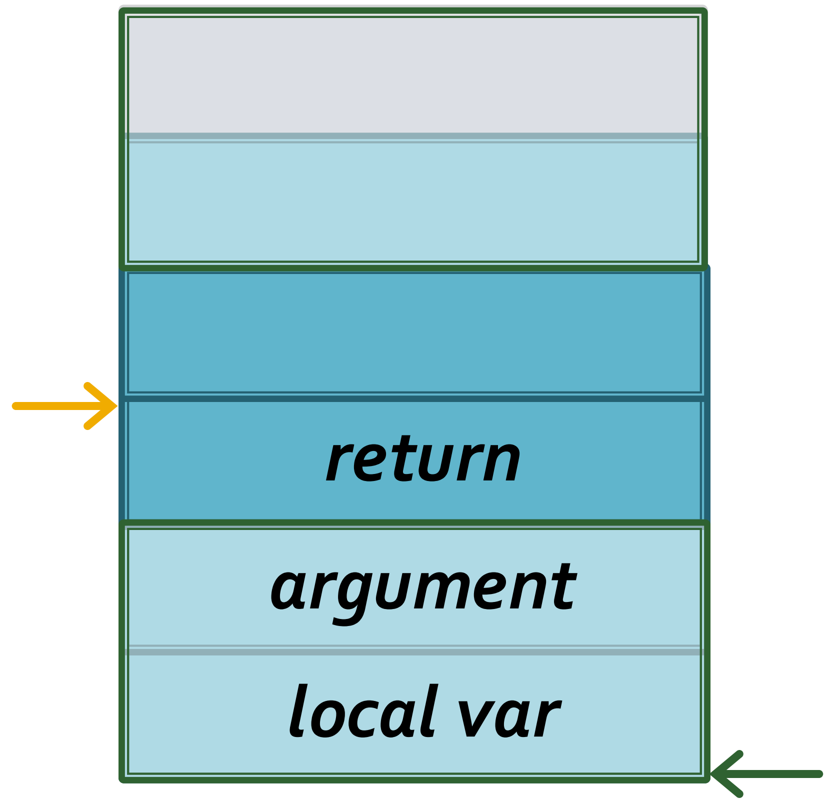


Return to libc

SETUP AS A RETURN

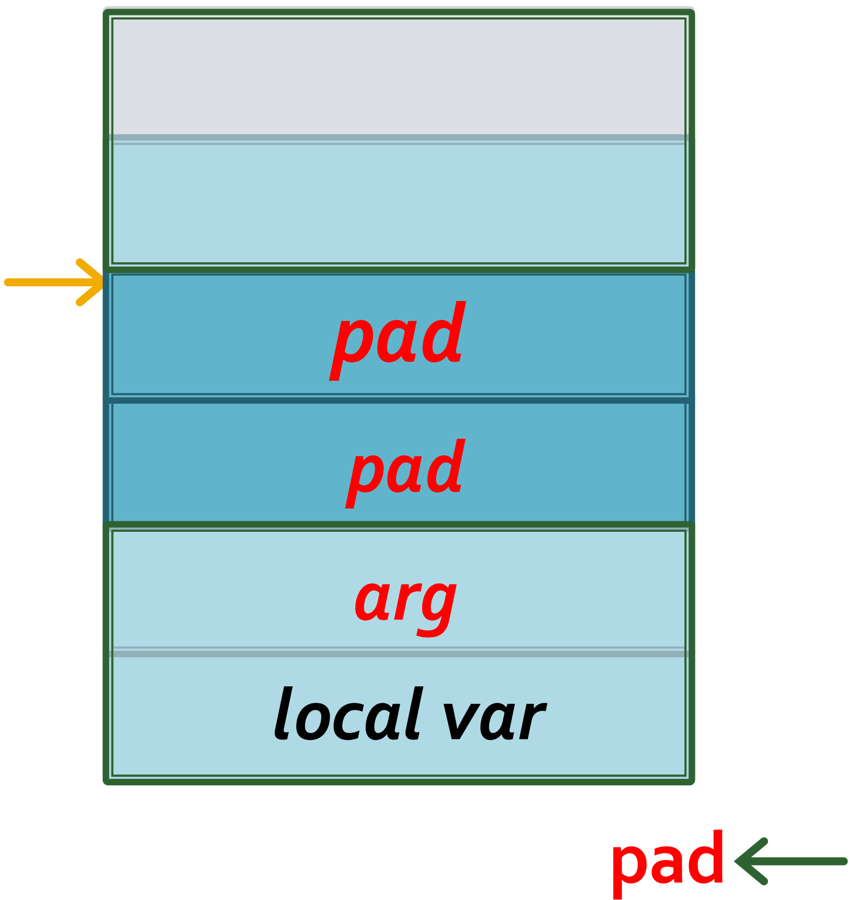


SETUP AS A FUNCTION CALL

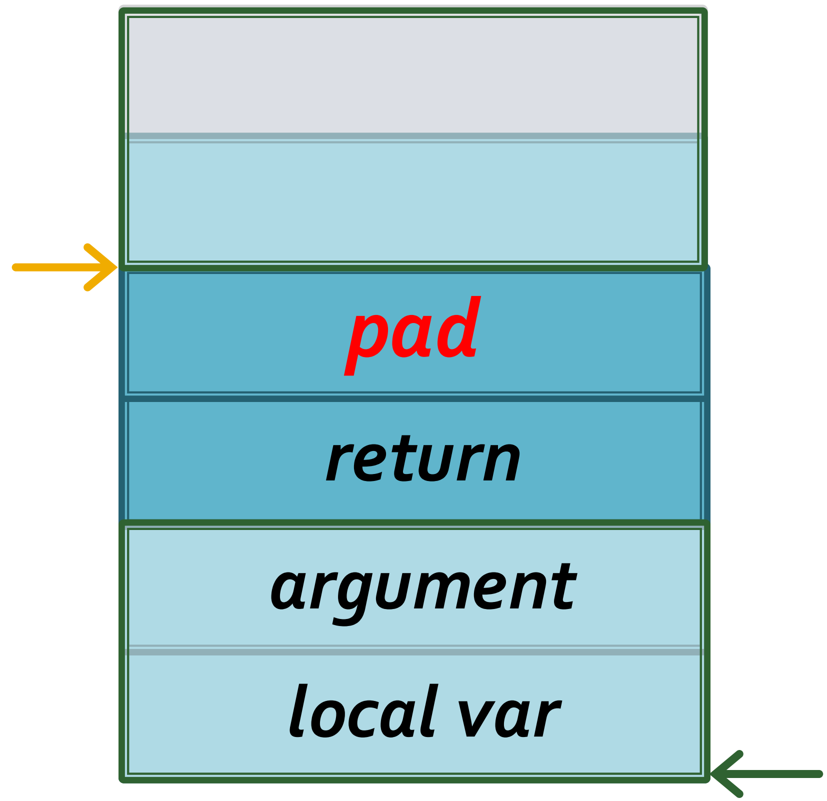


Return to libc

SETUP AS A RETURN

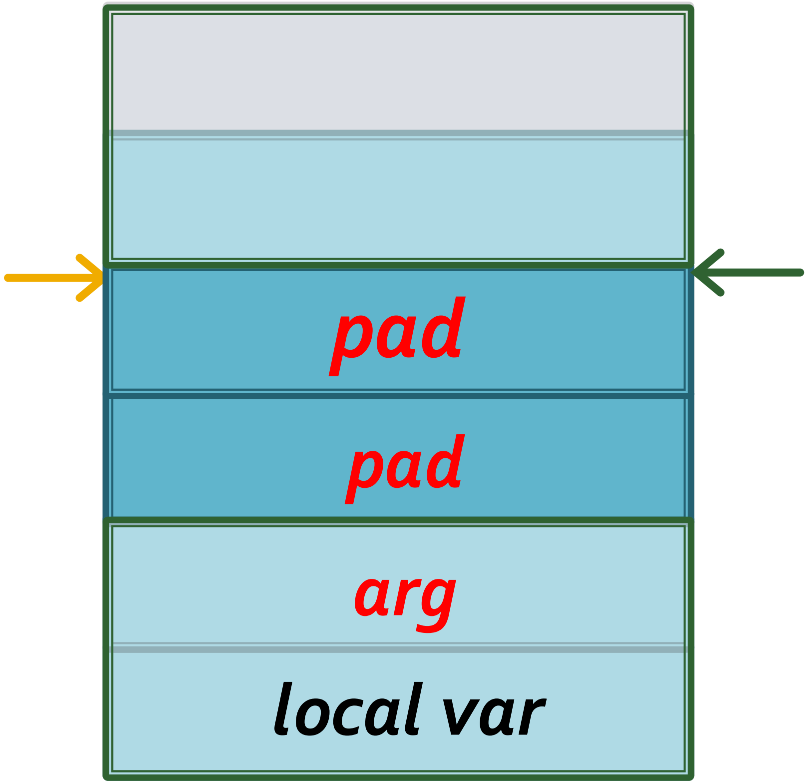


SETUP AS A FUNCTION CALL

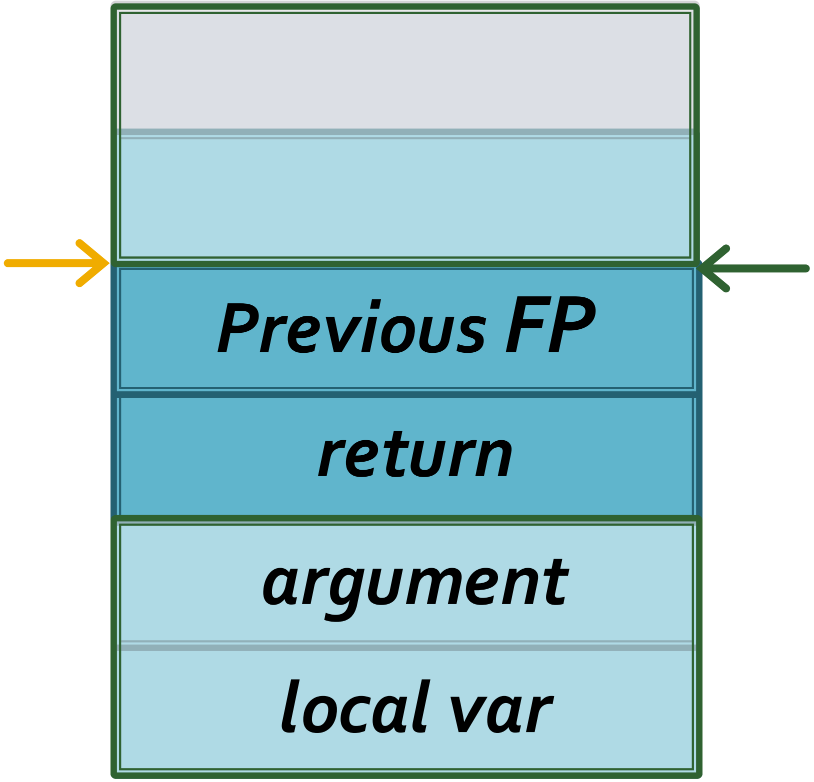


Return to libc

SETUP AS A RETURN

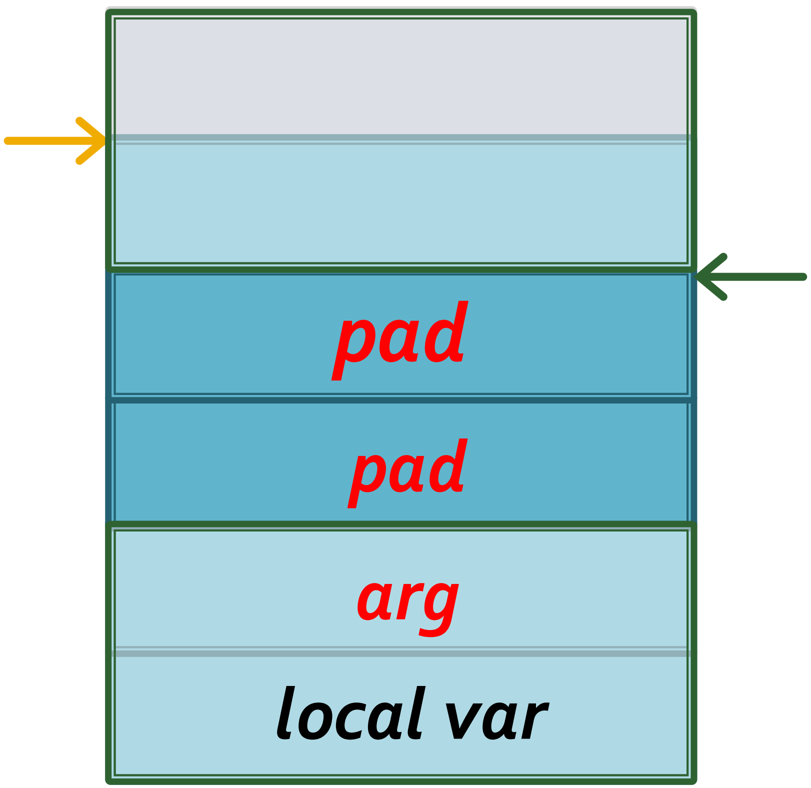


SETUP AS A FUNCTION CALL

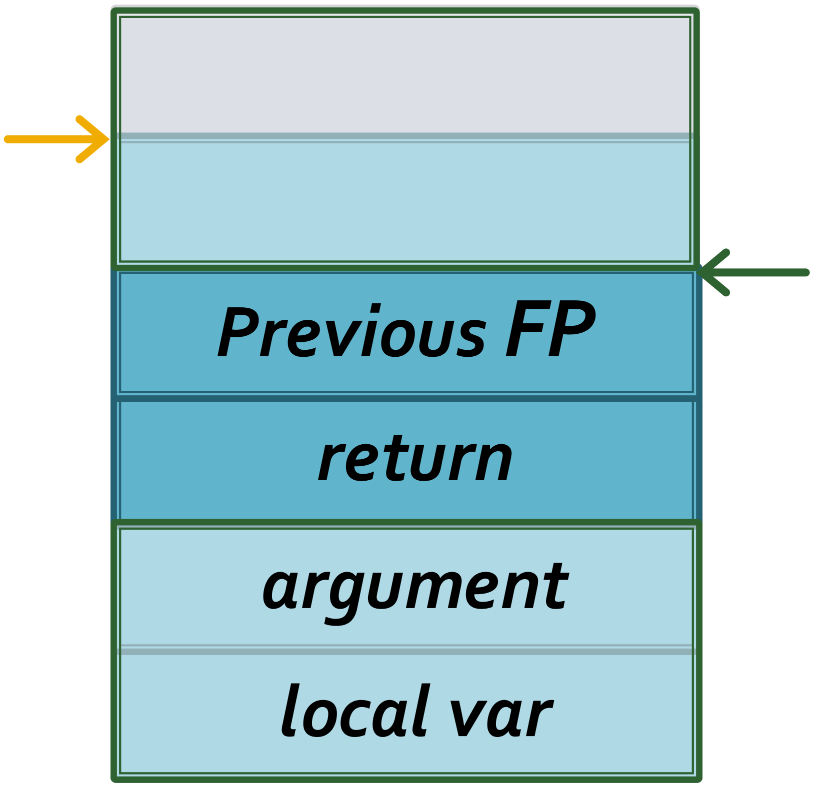


Return to libc

SETUP AS A RETURN



SETUP AS A FUNCTION CALL



Return to libc

Invoke any function that exists in the binary
execv() is a popular one

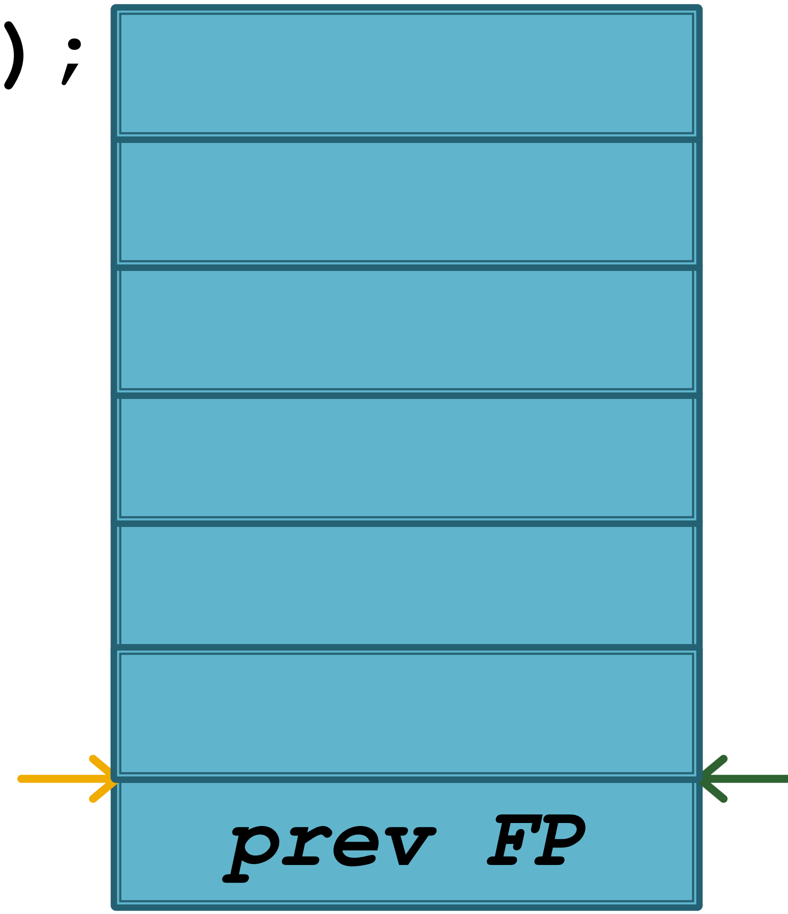
The **execv**(), **execvp**(), and **execvpe**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers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

Return to libc

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

Return to libc

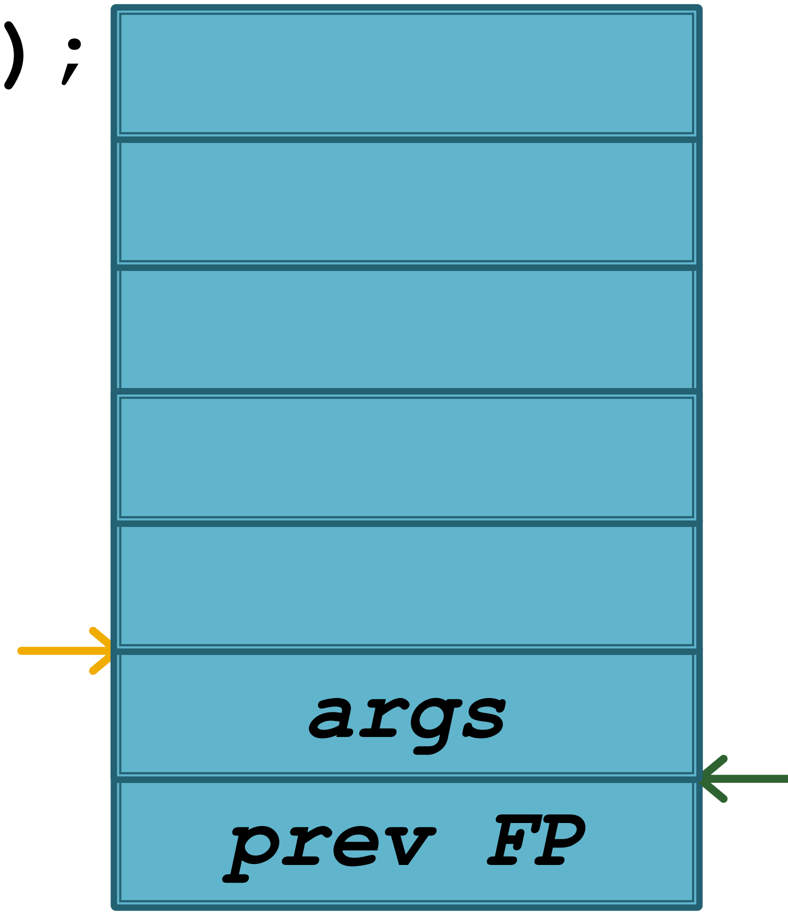
```
execv("/bin/ls", args);
```




```
Text:
path_str: "/bin/ls"
```

Return to libc

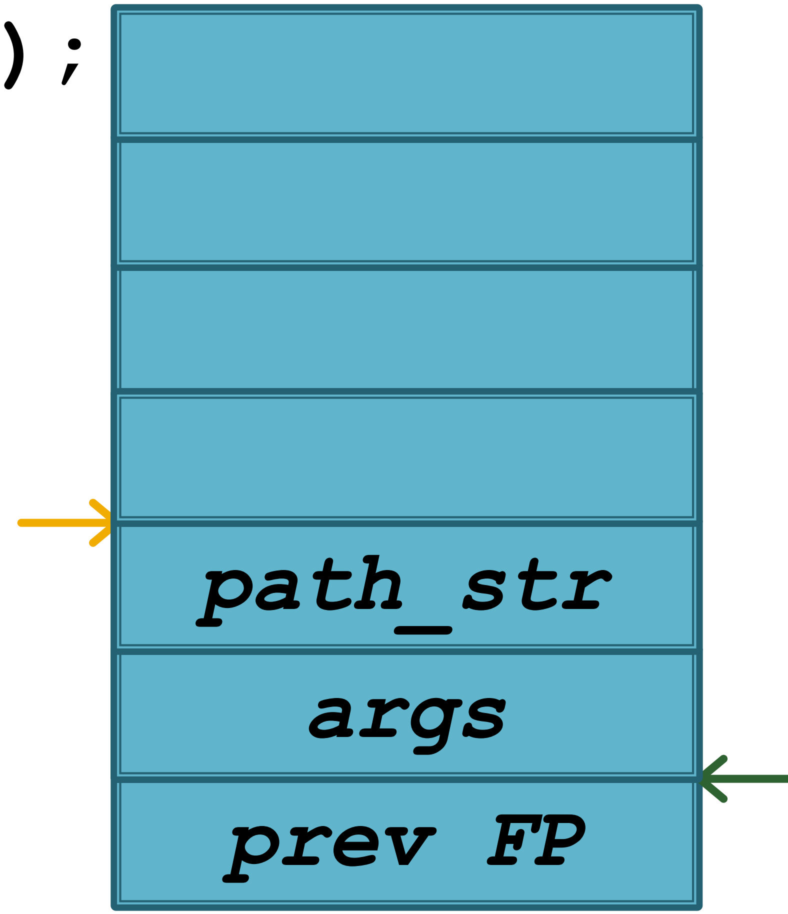
```
execv("/bin/ls", args);
```




```
Text:
path_str: “/bin/ls”
```

Return to libc

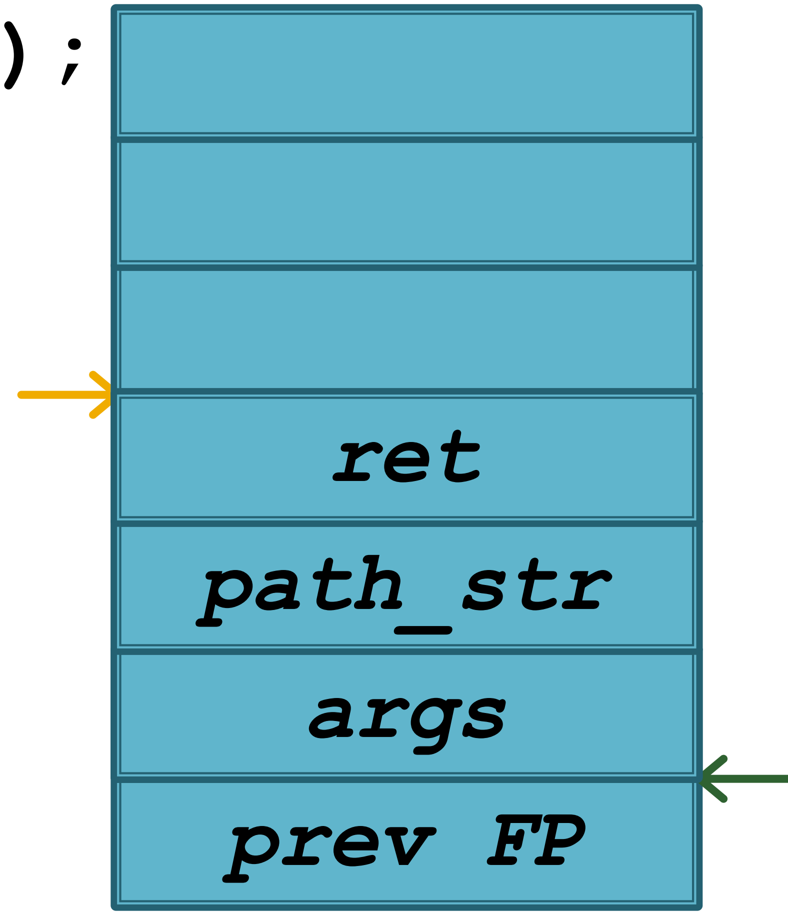
```
execv("/bin/ls", args);
```




```
Text:
path_str: "/bin/ls"
```

Return to libc

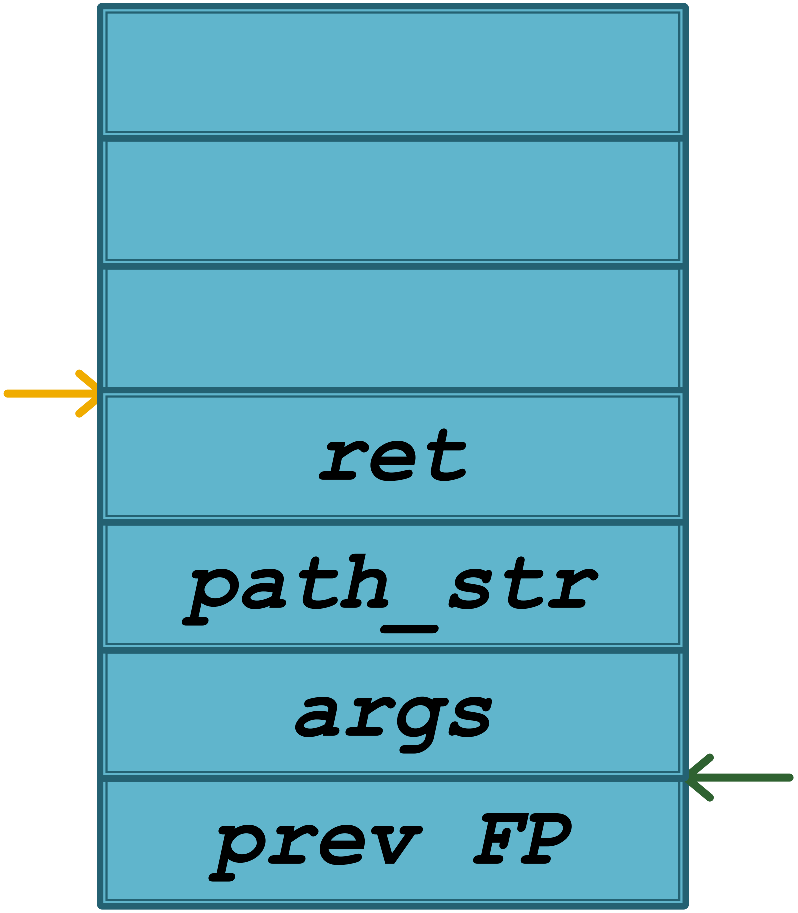
```
execv("/bin/ls", args);
```



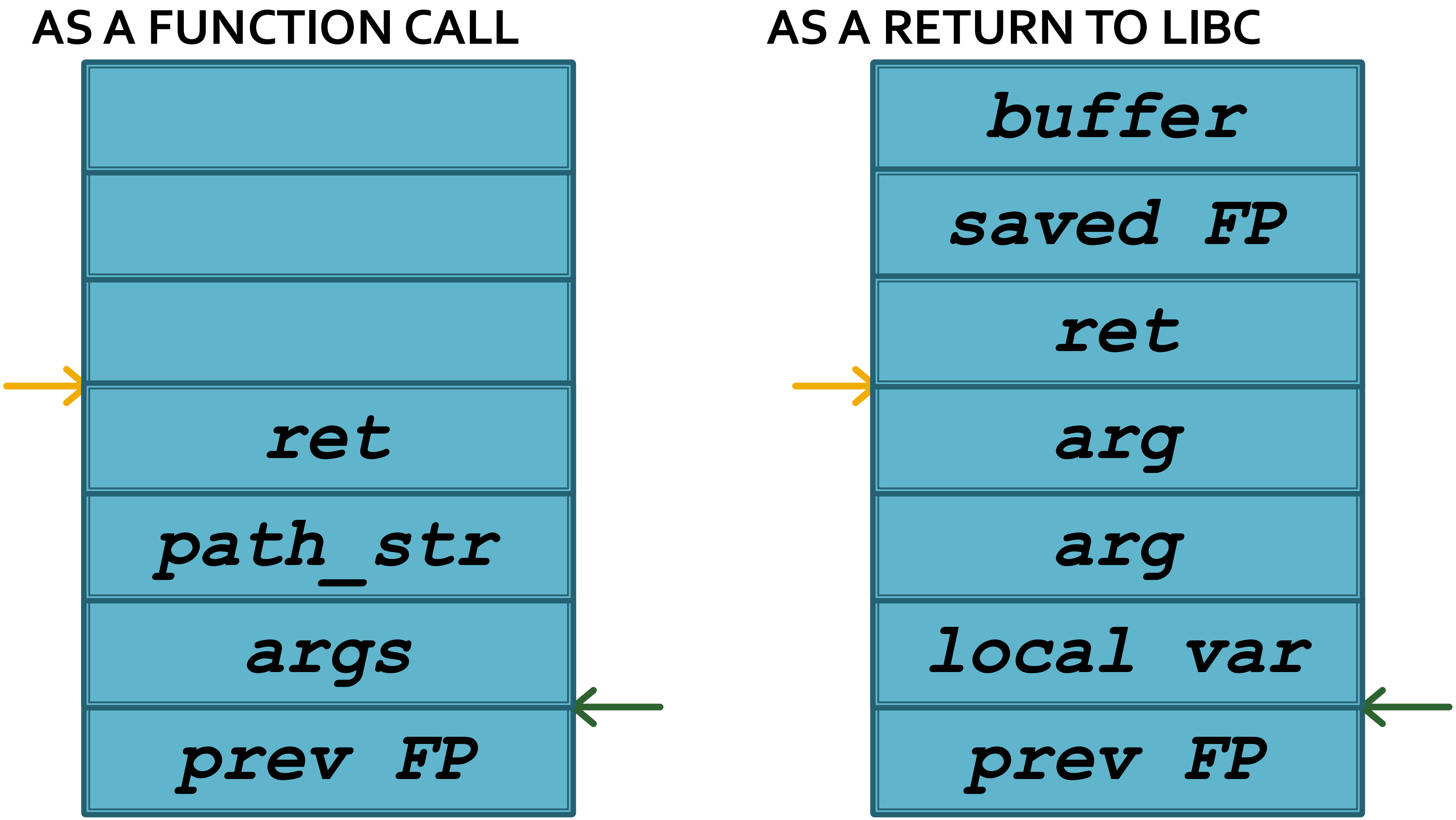

```
Text:
path_str: "/bin/ls"
```

Return to libc

AS A FUNCTION CALL

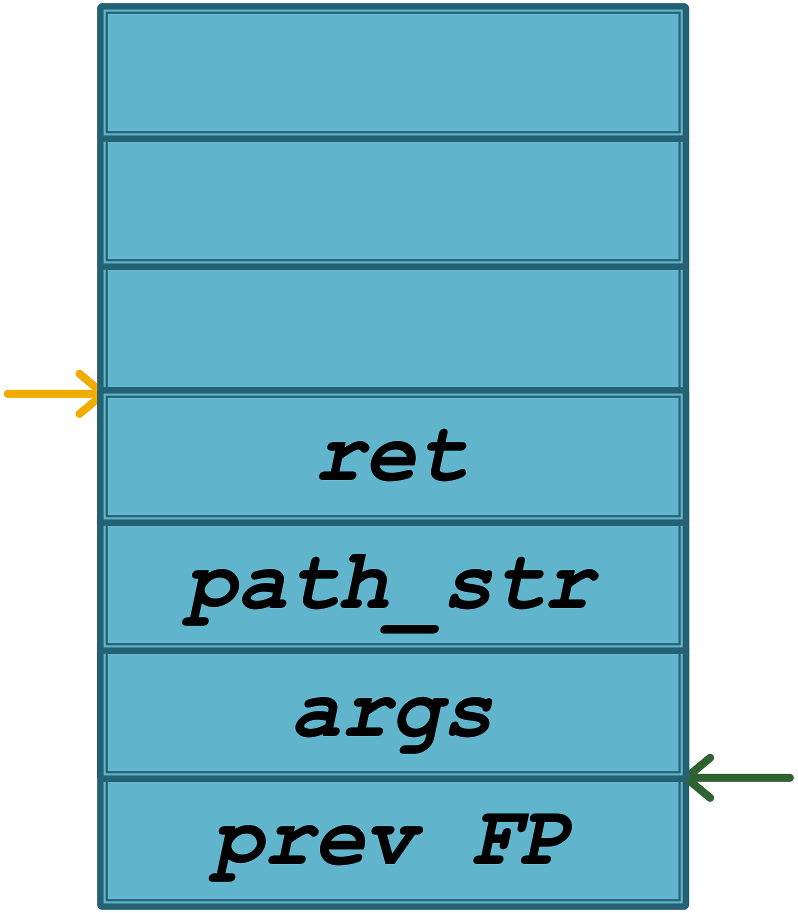


Return to libc

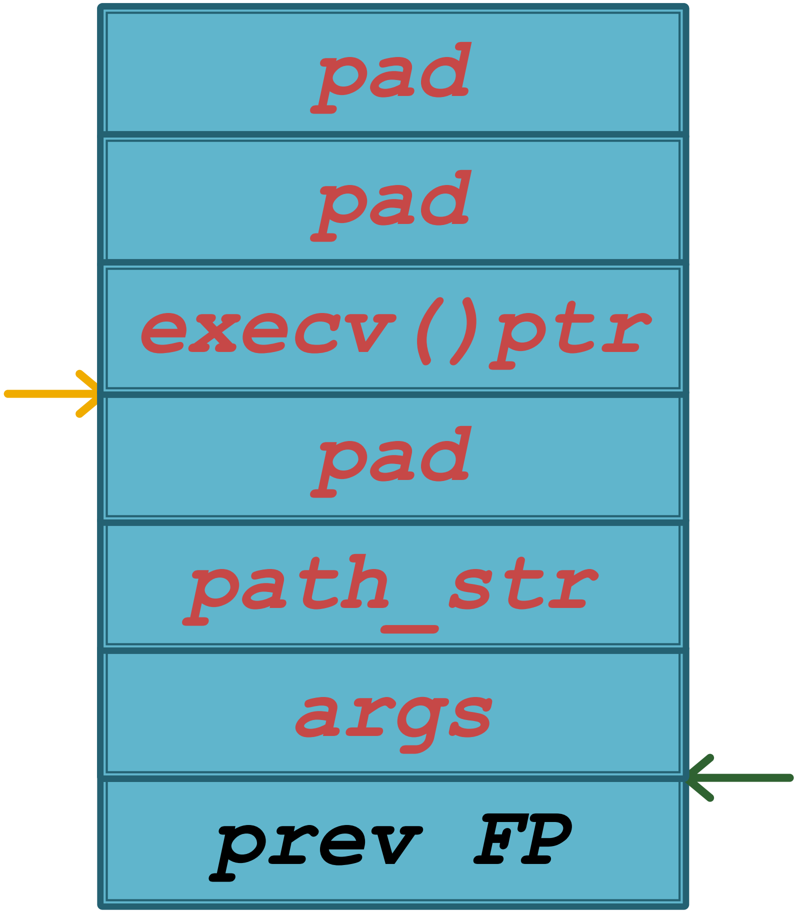


Return to libc

AS A FUNCTION CALL

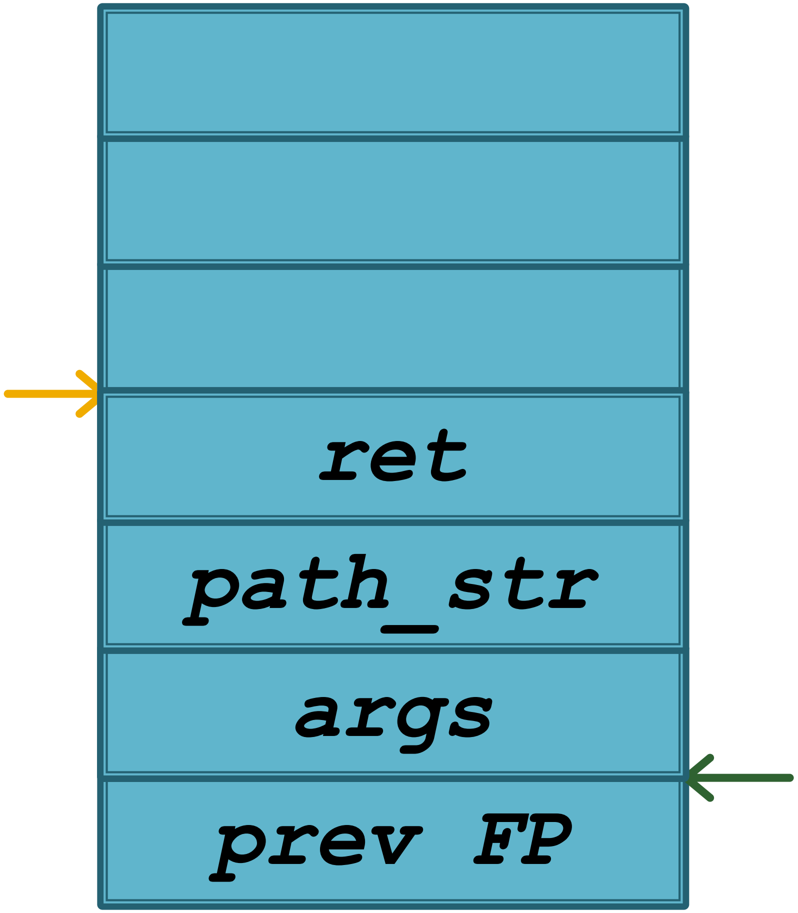


AS A RETURN TO LIBC

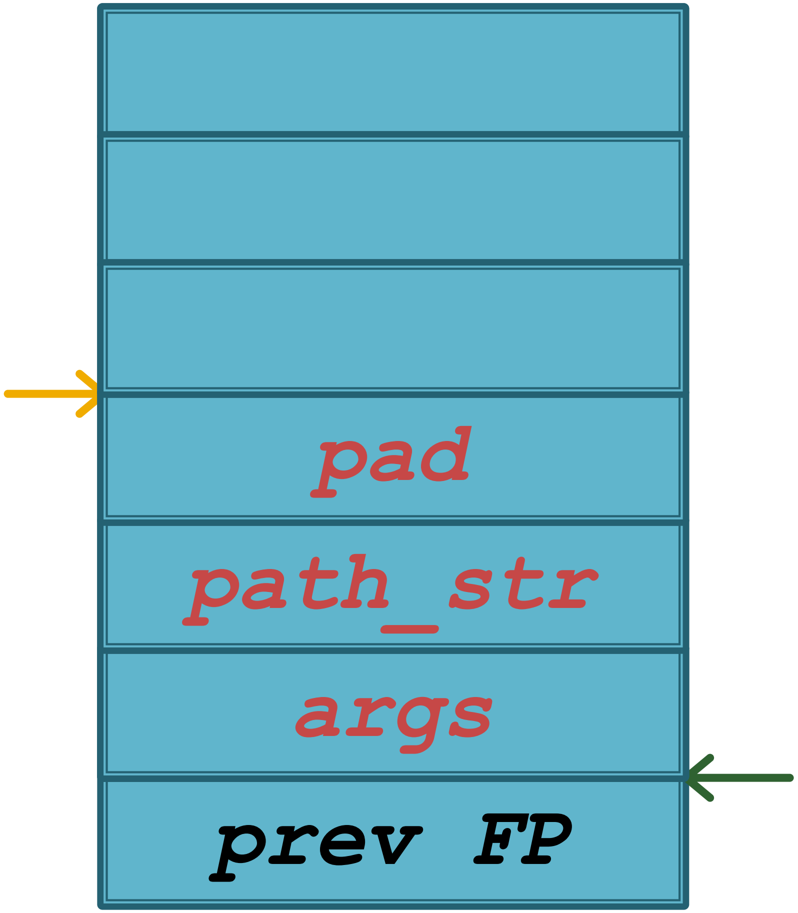


Return to libc

AS A FUNCTION CALL

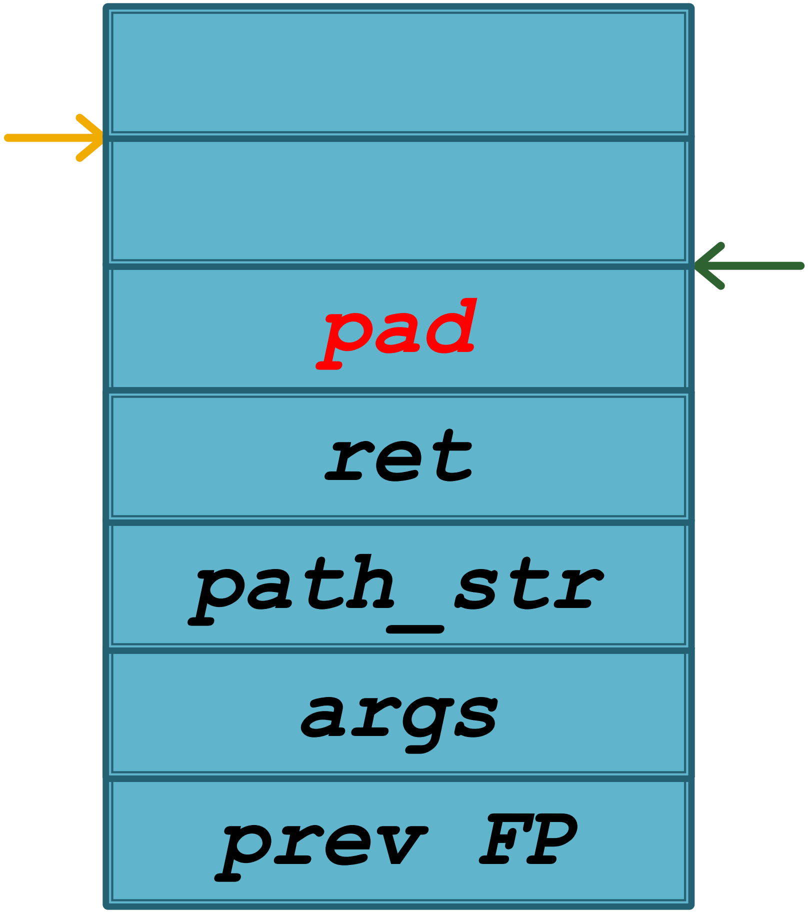


AS A RETURN TO LIBC

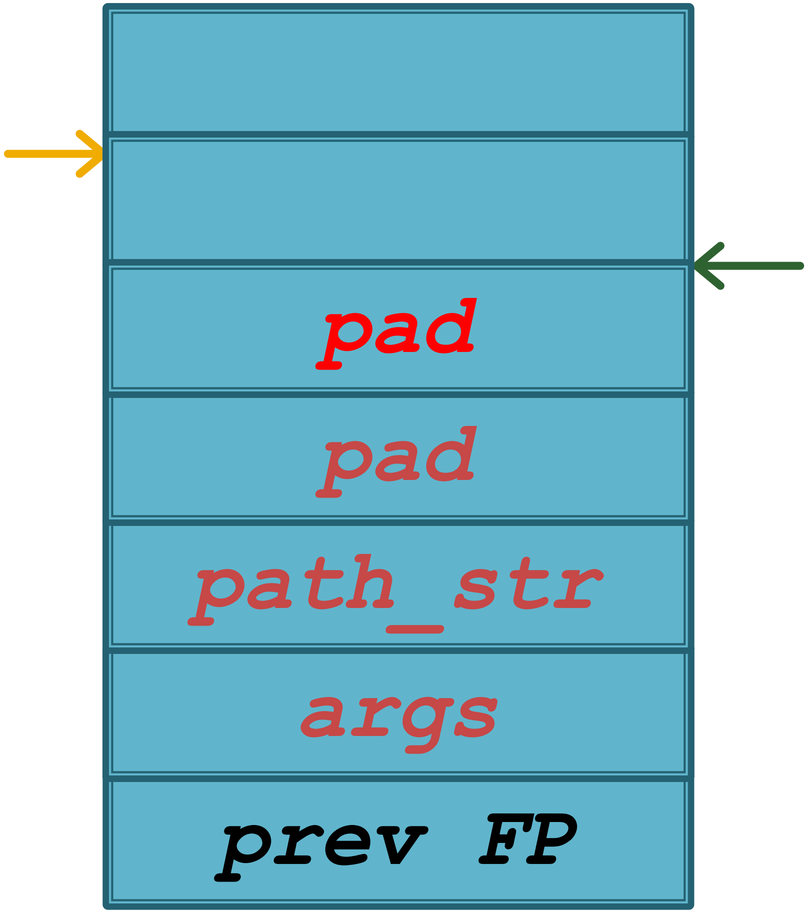


Return to libc

AS A FUNCTION CALL



AS A RETURN TO LIBC



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

Problem:

They are calling potentially evil functions

Solution:

Remove functions we don’t need!

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

Stack Canaries

Problem:

They keep overwriting return addresses!

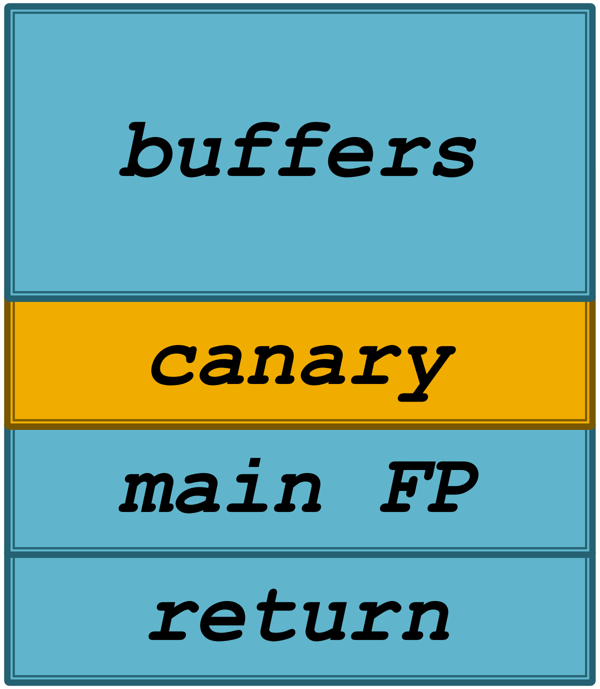
Solution:

Protect the return address!

Keep a canary in the coal mine!

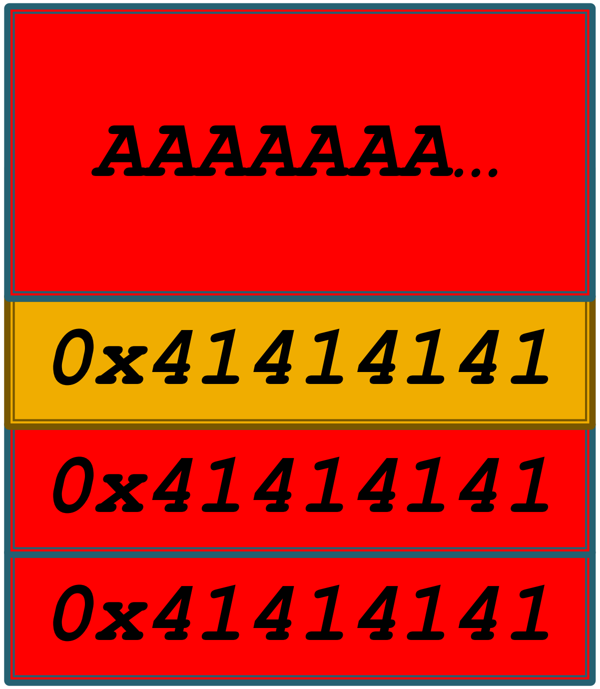
Stack Canaries

```
# on function call:

canary = secret
```



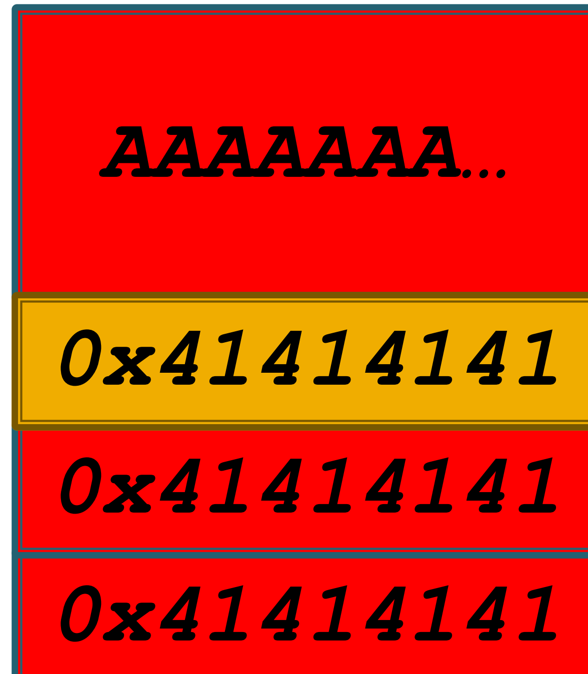
Stack Canaries

```
# vulnerability:

strcpy(buffer, str)
```



Stack Canaries

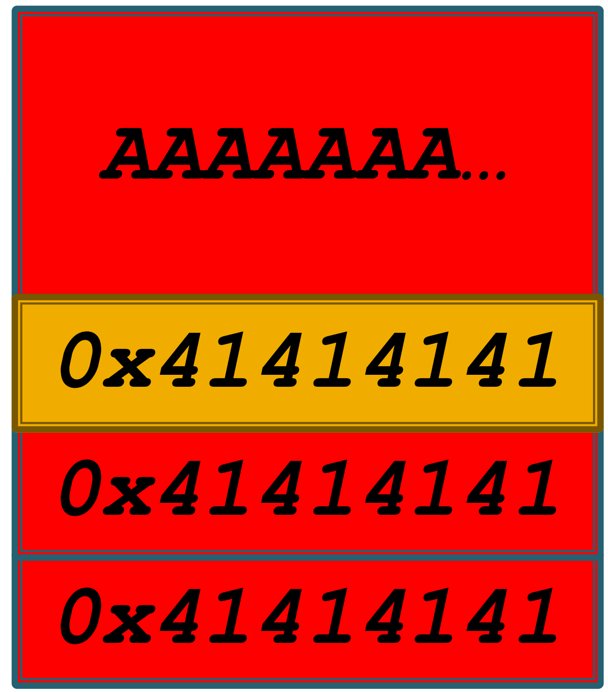
```
# on return:

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



Stack Canaries

*** stack smashing detected ***

```
# on return:

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



Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

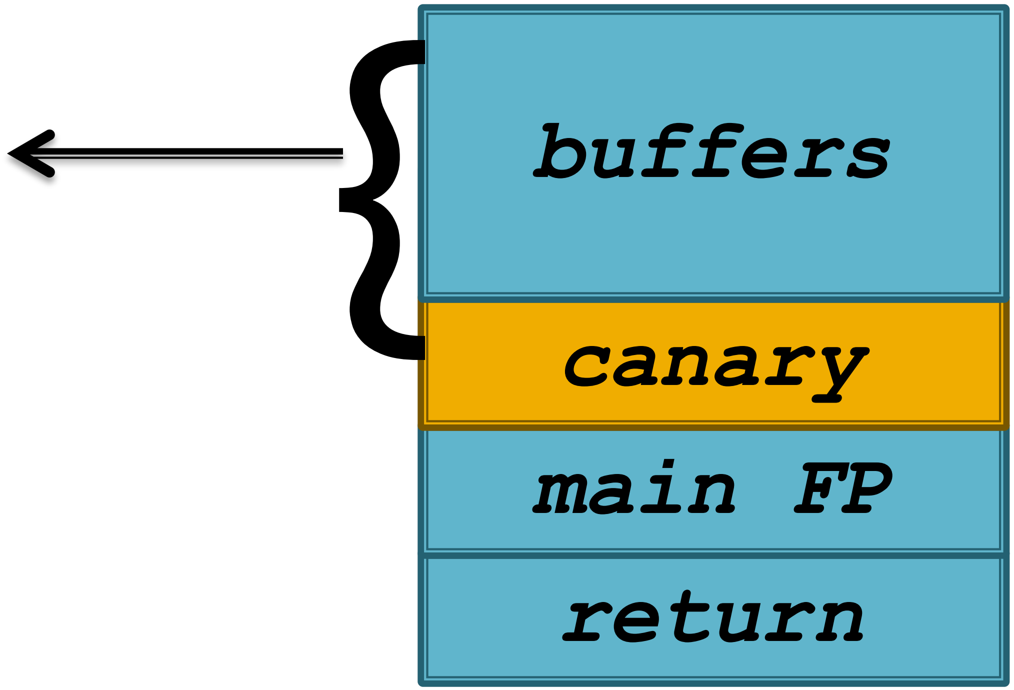
Buffer Over-read

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

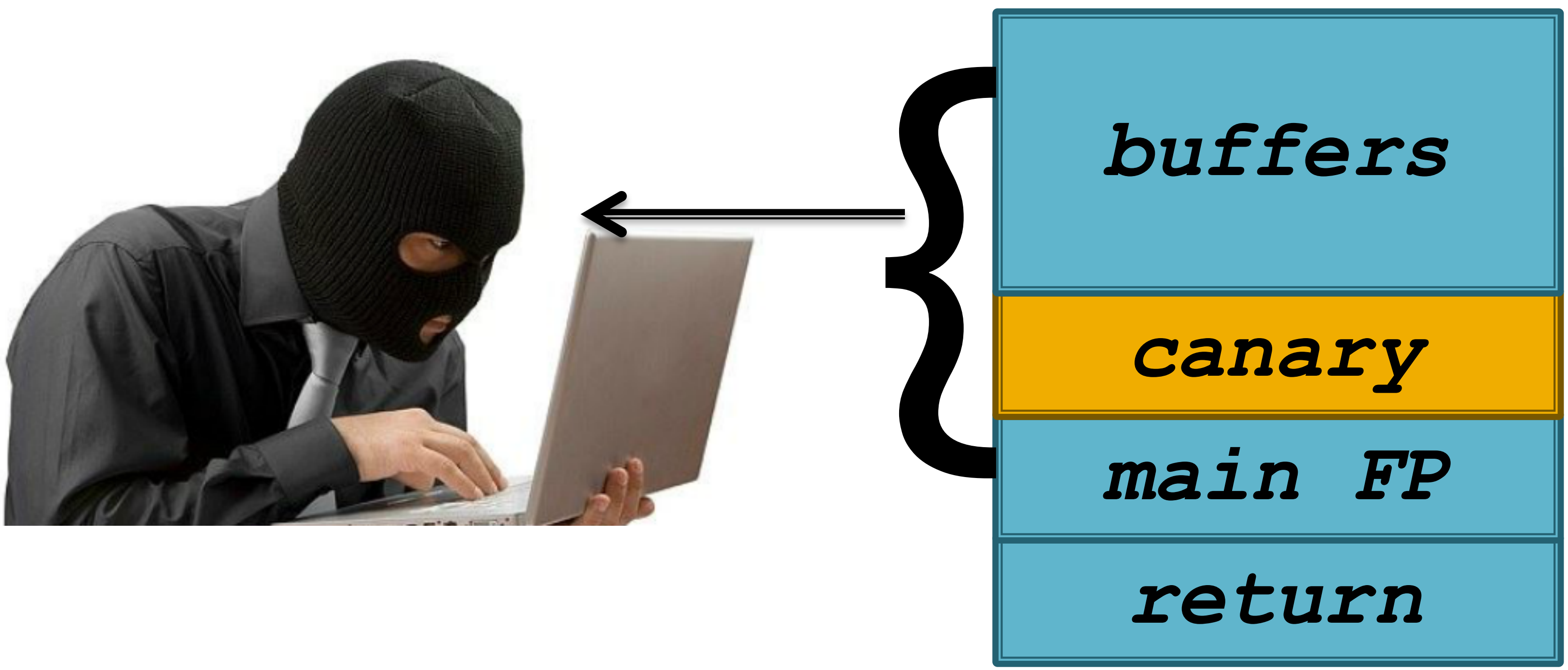
Buffer Over-read

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

Buffer Over-read



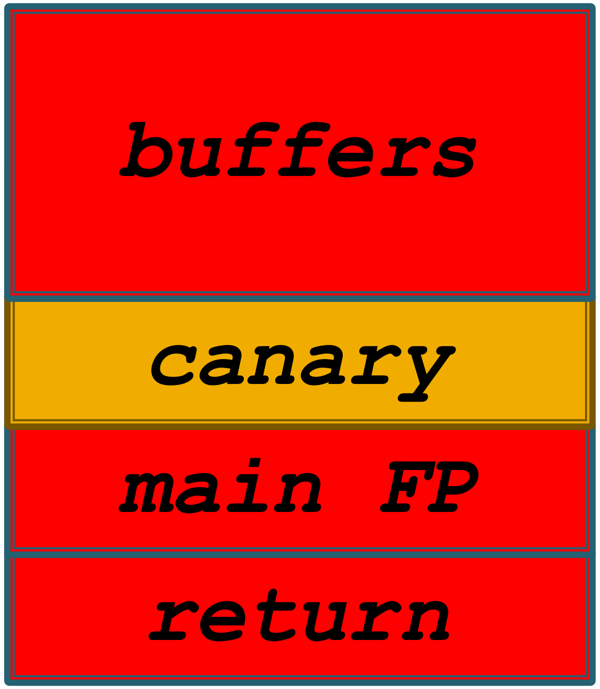
Buffer Over-read



Buffer Over-read

Buffer Overread

```
# on return:

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



Buffer Over-read

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

Integer overflow

Unsafe:

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

Use instead:

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

Integer Overflow

Problem:

Replacing `strcpy` with `str`**`n`**`cpy` is easy

Solution:

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

Can not be represented in 32 bits

Integer Overflow

len = 1,073,742,024 (~1 billion)

0x400000c8

len * 4 = 4,294,968,096 (~4 billion)

0x100000320

as uint32

len * 4 = 800

0x00000320

Integer overflow

size

200

buffer

```
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];
    }
}
```

Write

~1 billion

elements

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

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;
}
```

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;
}
```

Negative Number

Passes Signed Check

Treated as a very large number (unsigned integer)

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

ROP

Problem:

They took out functions that can launch shells

Solution:

Use the instructions that are still there

ROP

Return Oriented Programming

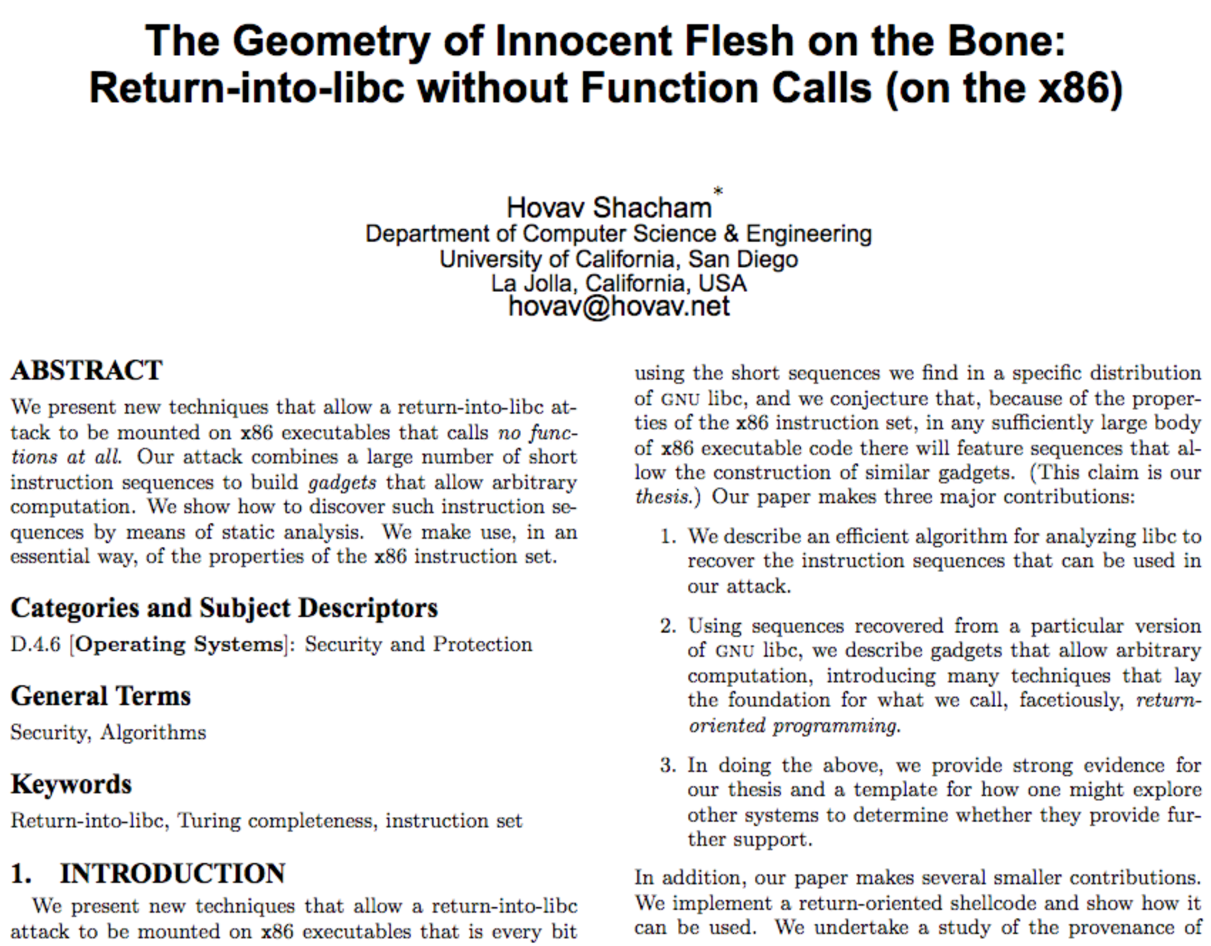
Return to libc without function calls

Arbitrary functionality via “gadgets”

- Turing complete

Worse on x86

ROP



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.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Algorithms

Keywords

Return-into-libc, Turing completeness, instruction set

1. INTRODUCTION

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit 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 our attack.

2. Using sequences recovered from a particular version of GNU libc, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call, facetiously, *return-oriented programming*.

3. In doing the above, we provide strong evidence for our thesis and a template for how one might explore other systems to determine whether they provide further support.

In addition, our paper makes several smaller contributions. We implement a return-oriented shellcode and show how it can be used. We undertake a study of the provenance of

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 **var = var - 10**

```
foo:
  push ebp
  mov esp, ebp
  mov eax, [ebp + 4]
  add eax, 10
  mov [eax], 0x00
  sub eax, 10
  leave
  ret
```

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

ROP Gadget

Small section of code

Contains a very small number of instructions

Ends in a `ret`

Not an existing function body

Don’t even have to be an existing ret

ROP

`0xc3` : **`ret`**

- Could be part of another instruction
- Could be part of an address

X86 uses "variable length instructions"

- The meaning of opcode bytes depends on where the instruction begins (where EIP points to)

Any `0xc3` byte is a valid ROP gadget return

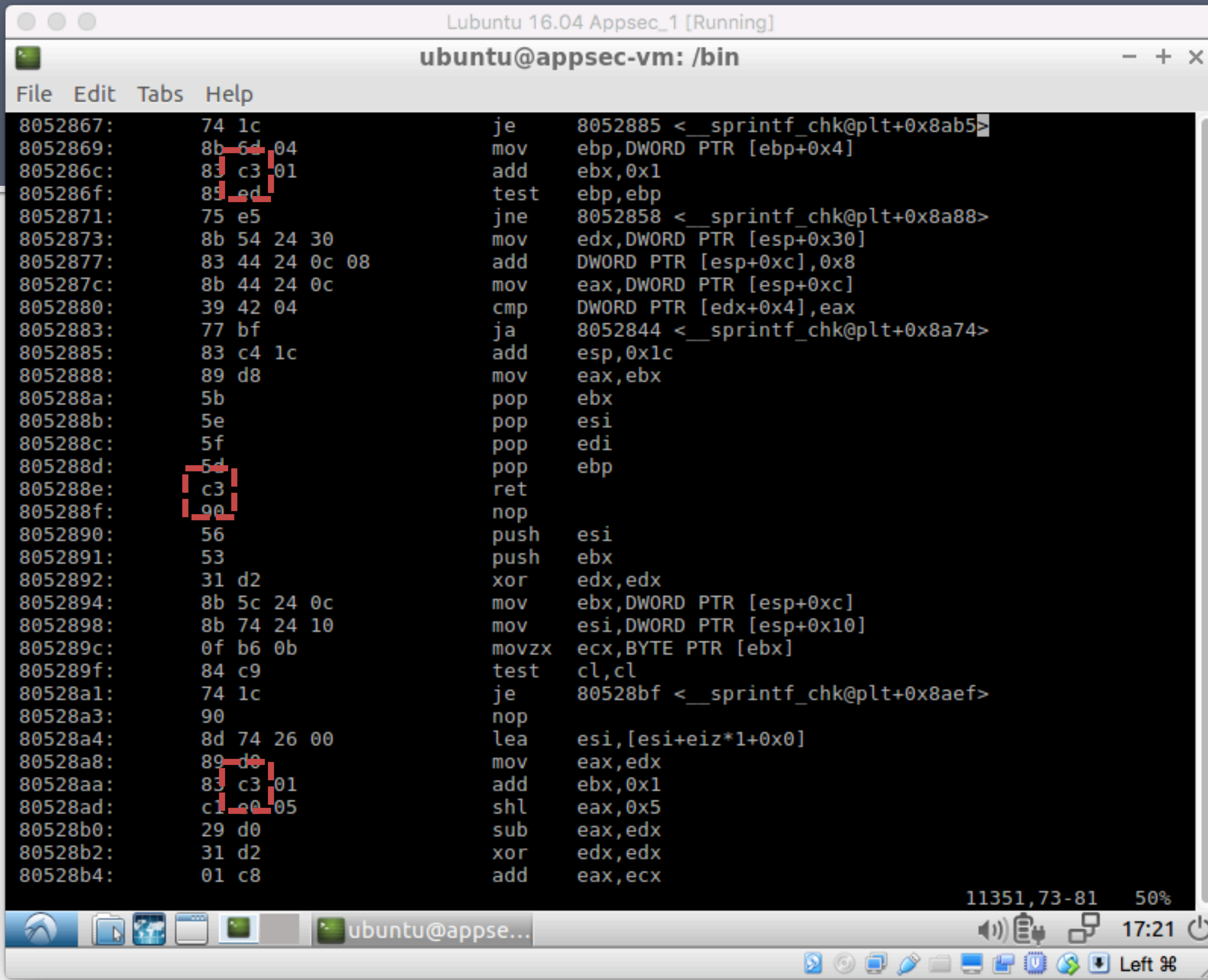
Lubuntu 16.04 Appsec_1 [Running]

ubuntu@appsec-vm: /bin

File Edit Tabs Help

```
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%
```

ubuntu@appse... 17:21 Left ⌘


Lubuntu 16.04 Appsec_1 [Running]
ubuntu@appsec-vm: /bin
File Edit Tabs Help
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%
ubuntu@appse...
17:21
Left ⌘

ROP

Bytes in the Code Section:

`00 F7 C7 07 00 00 00 0f 95 45 c3`

Full Gadget:

ROP

EIP

`00 F7 C7 07 00 00 00 0f 95 45 c3`

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

Full Gadget:

```
ret
```

ROP

EIP

`00 F7 C7 07 00 00 00 0f 95 45 c3`

`inc ebp` **`ret`**

Full Gadget:

```
inc ebp
ret
```

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

ret

inc ebp

xchg ebp, eax

Full Gadget:

```
xchg ebp, eax
inc ebp
ret
```

ROP

EIP

00 F7 C7 07 00 00 00 0f 95 45 c3

setnzb [ebp - 61]

Full Gadget:

setnzb [ebp-61]

<no return>

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

Full Gadget:

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

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

Full Gadget:

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

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

Full Gadget:

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

ROP

EIP

`00 F7 C7 07 00 00 00 0f 95 45 c3`

Full Gadget:

`<none - invalid instruction>`

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

ret

mov edi, 0x0F000000

inc ebp

xchg ebp, eax

Full Gadget:

```
mov edi, 0x0F000000
xchg ebp, eax
inc ebp
ret
```

ROP

EIP

00 F7 C7 07 00 00 00 0f 95 45 c3

`test edi, 0x00000007`

`setnzb [ebp - 61]`

Full Gadget:

```
test edi, 0x00000007
setnzb [ebp-61]
<no return>
```

ROP

EIP

```
00 F7 C7 07 00 00 00 0f 95 45 c3
```

```
add bh, dh
mov edi, 0x0F000000
xchg ebp, eax
inc ebp
ret
```

Full Gadget:

```
add bh, dh
mov edi, 0x0F000000
xchg ebp, eax
inc ebp
ret
```

ROP-Chains

Gadget1:

```
mov eax, 0x10; ret
```

Gadget2:

```
add eax, ebp; ret
```

Gadget3:

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

Gadget4:

```
mov ebp, esp; ret
```

ROP-Chains

Gadget1:

```
mov eax, 0x10; ret
```

Gadget2:

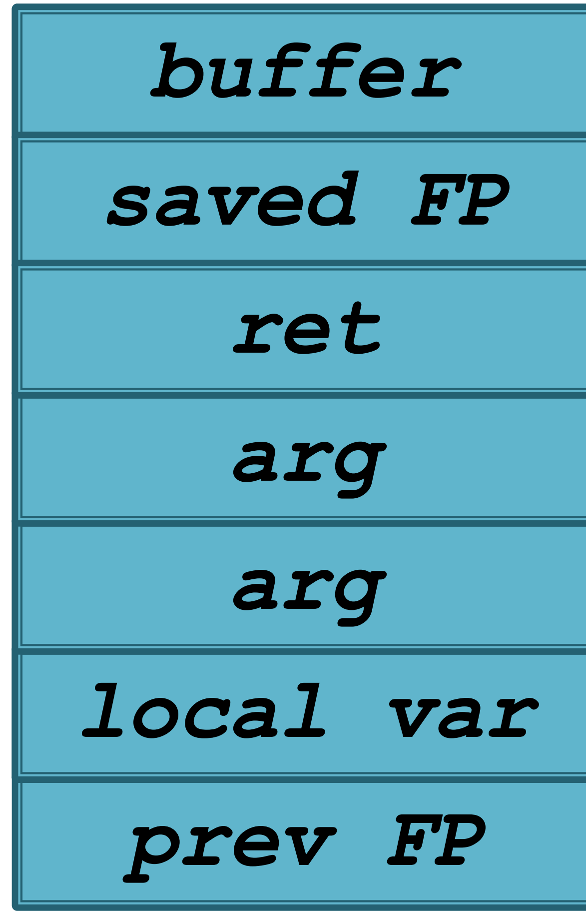
```
add eax, ebp; ret
```

Gadget3:

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

Gadget4:

```
mov ebp, esp; ret
```



ROP-Chains

Gadget1:

```
mov eax, 0x10; ret
```

Gadget2:

```
add eax, ebp; ret
```

Gadget3:

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

Gadget4:

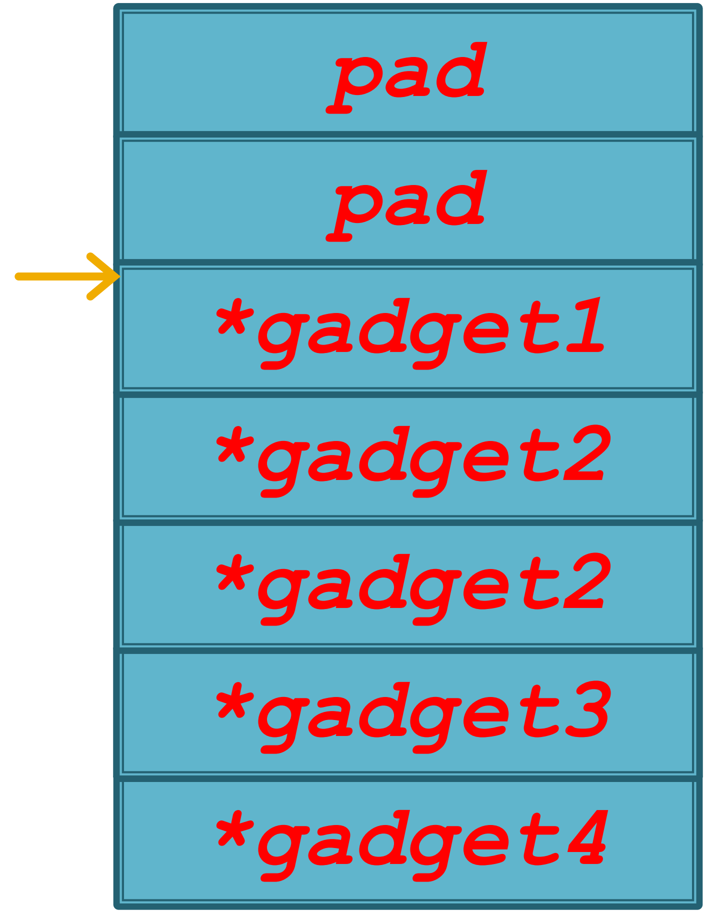
```
mov ebp, esp; ret
```

pad
pad
**gadget1*
**gadget2*
**gadget2*
**gadget3*
**gadget4*

ROP-Chains

ROP Chain:

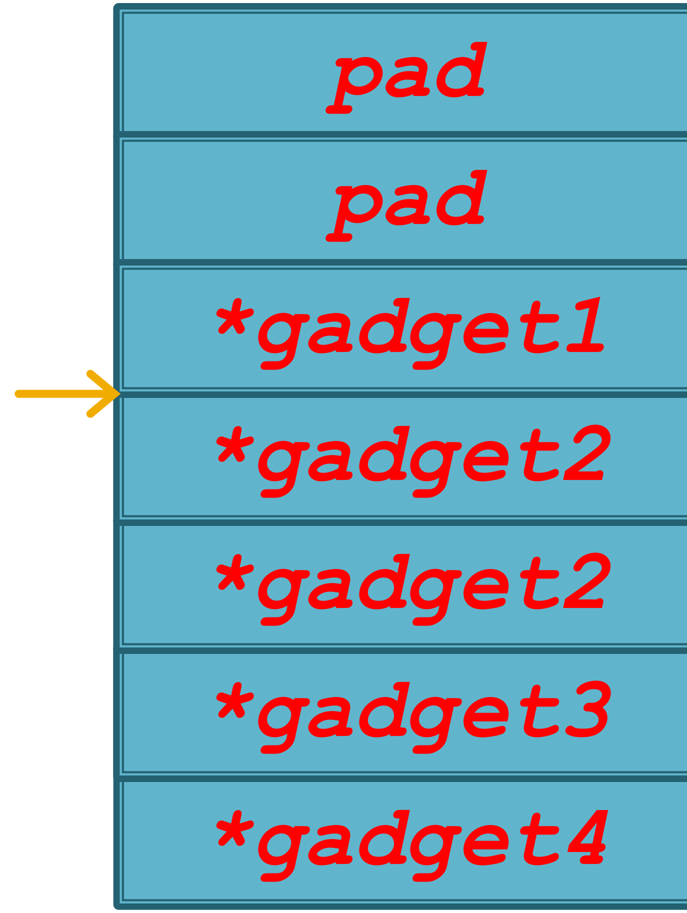
```
mov eax, 0x10
add eax, ebp
add eax, ebp
mov [eax+8], eax
mov ebp, esp
ret
```



ROP-Chains

ROP Chain:

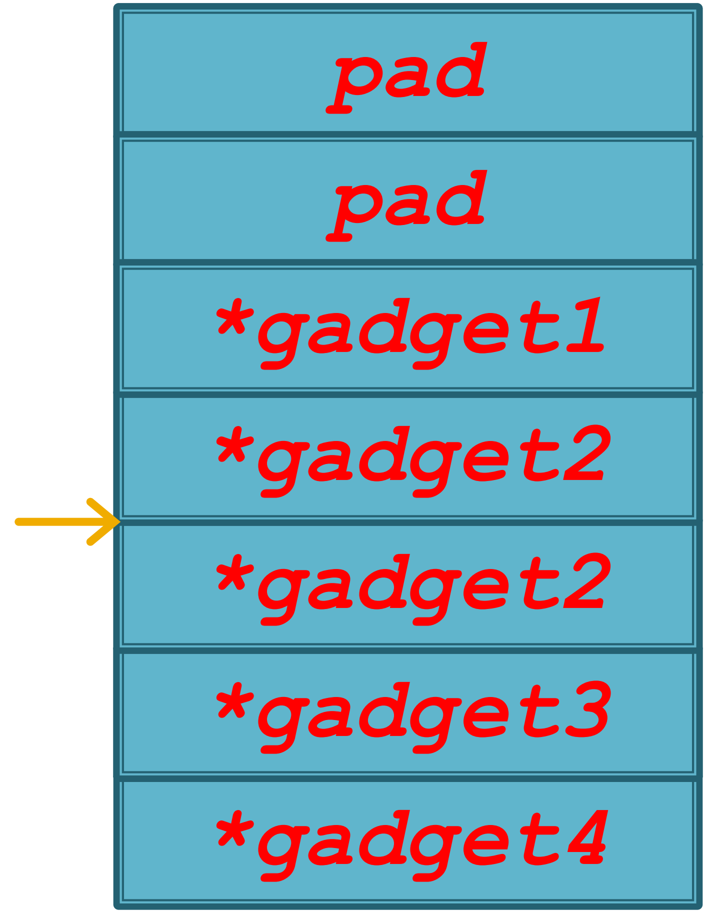
```
mov eax, 0x10
add eax, ebp
add eax, ebp
mov [eax+8], eax
mov ebp, esp
ret
```



ROP-Chains

ROP Chain:

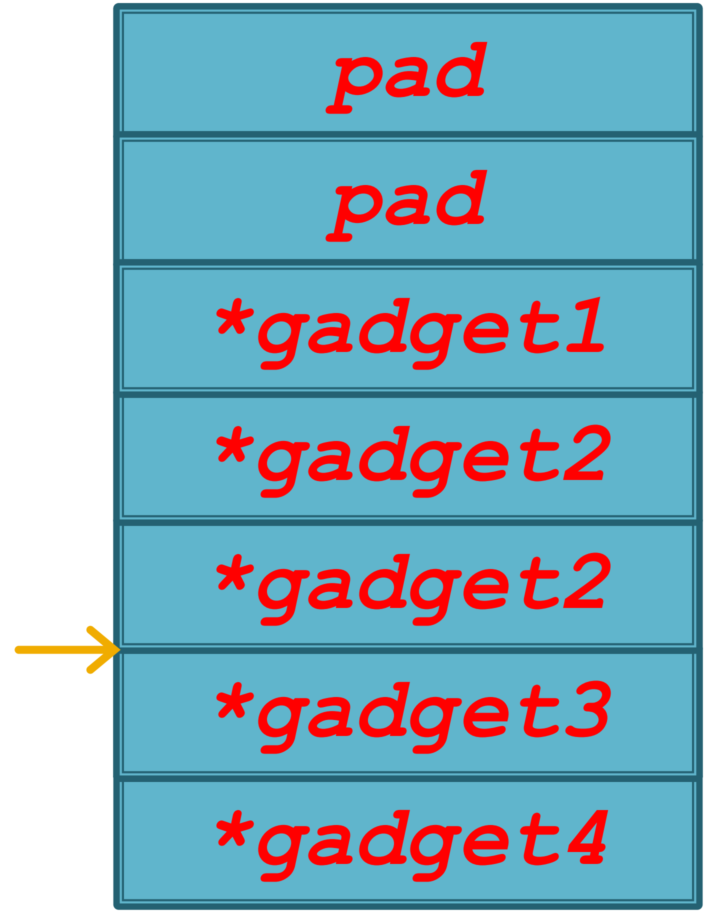
```
mov eax, 0x10
add eax, ebp
add eax, ebp
mov [eax+8], eax
mov ebp, esp
ret
```



ROP-Chains

ROP Chain:

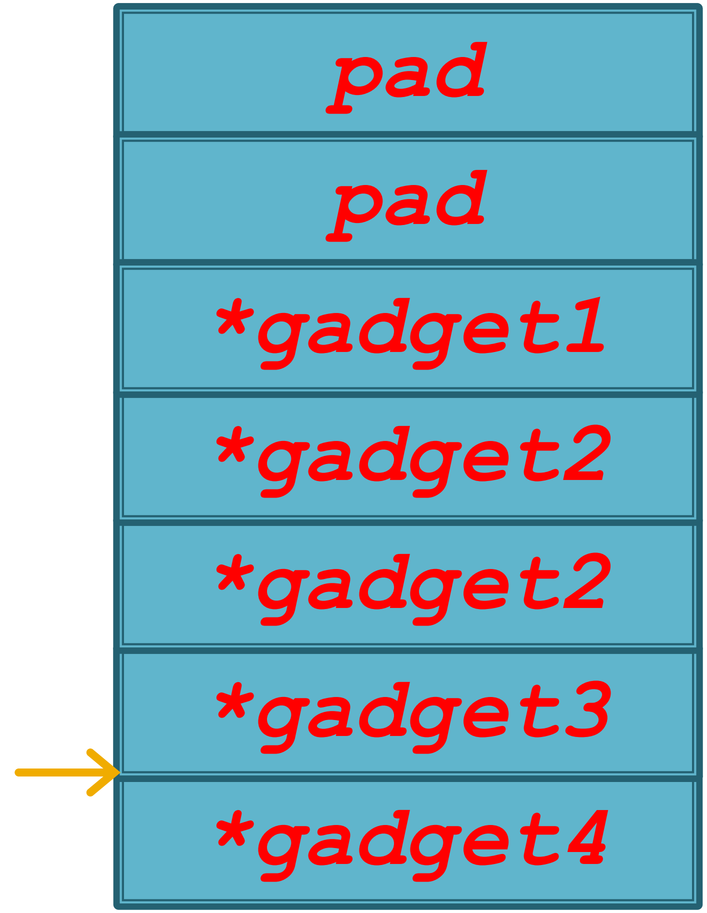
```
mov eax, 0x10
add eax, ebp
add eax, ebp
mov [eax+8], eax
mov ebp, esp
ret
```



ROP-Chains

ROP Chain:

```
mov eax, 0x10
add eax, ebp
add eax, ebp
mov [eax+8], eax
mov ebp, esp
ret
```



ROP

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.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Algorithms

Keywords

Return-into-libc, Turing completeness, instruction set

1. INTRODUCTION

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit 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 our attack.

2. Using sequences recovered from a particular version of GNU libc, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call, facetiously, *return-oriented programming*.

3. In doing the above, we provide strong evidence for our thesis and a template for how one might explore other systems to determine whether they provide further support.

In addition, our paper makes several smaller contributions. We implement a return-oriented shellcode and show how it can be used. We undertake a study of the provenance of

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

ASLR

Problem:

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

Solution:

Move around where the code lives

ASLR

Address Space Layout Randomization

Make it extremely hard to predict references

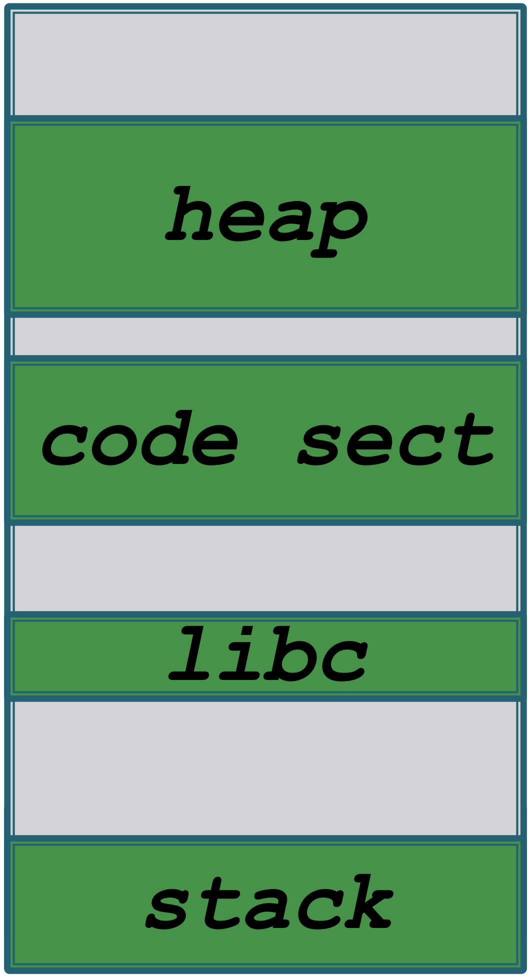
Requires many changes to compilation and/or loading

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

<Details are out-of-scope>

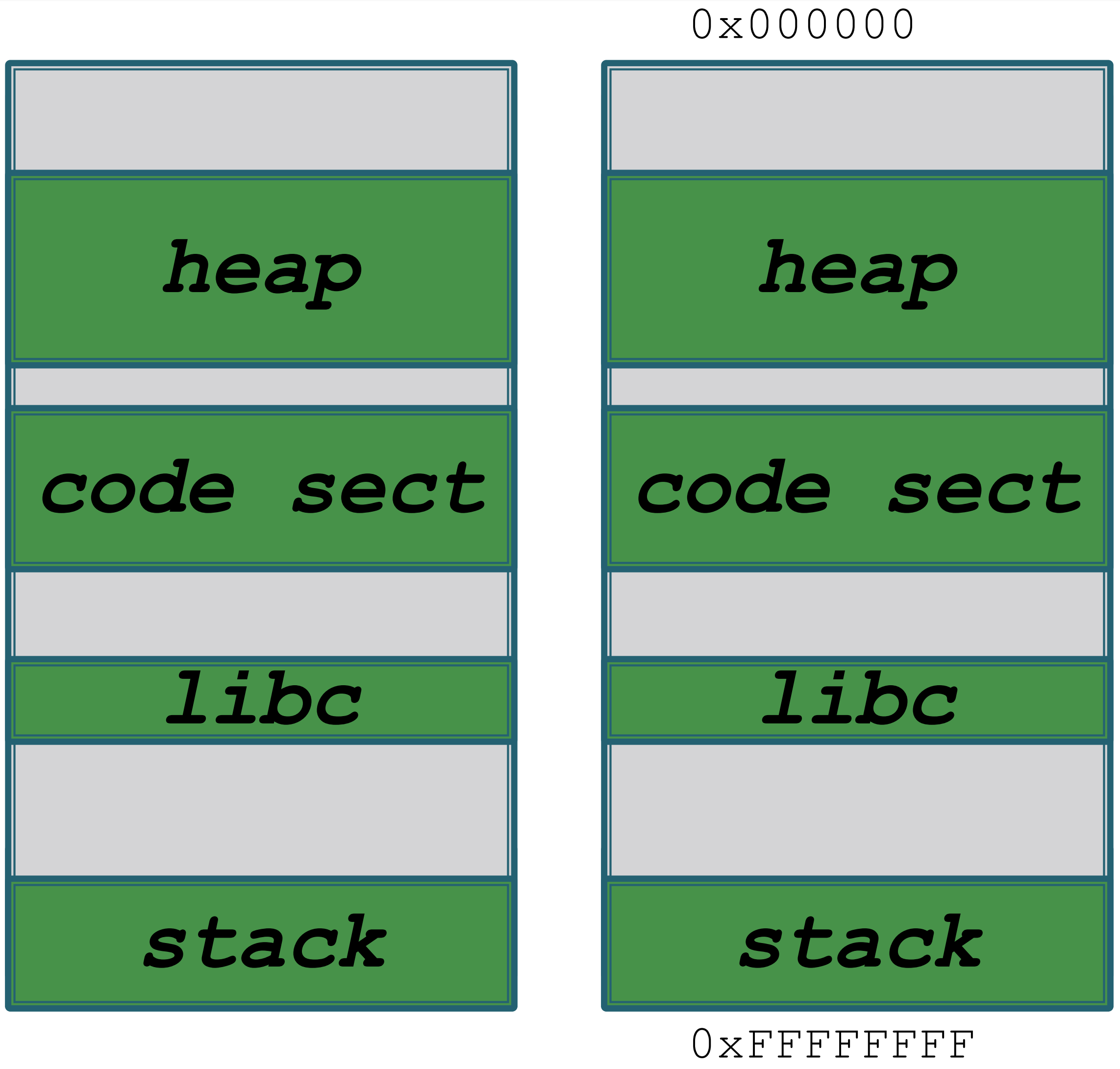
Memory Layout (no ASLR)

0x000000

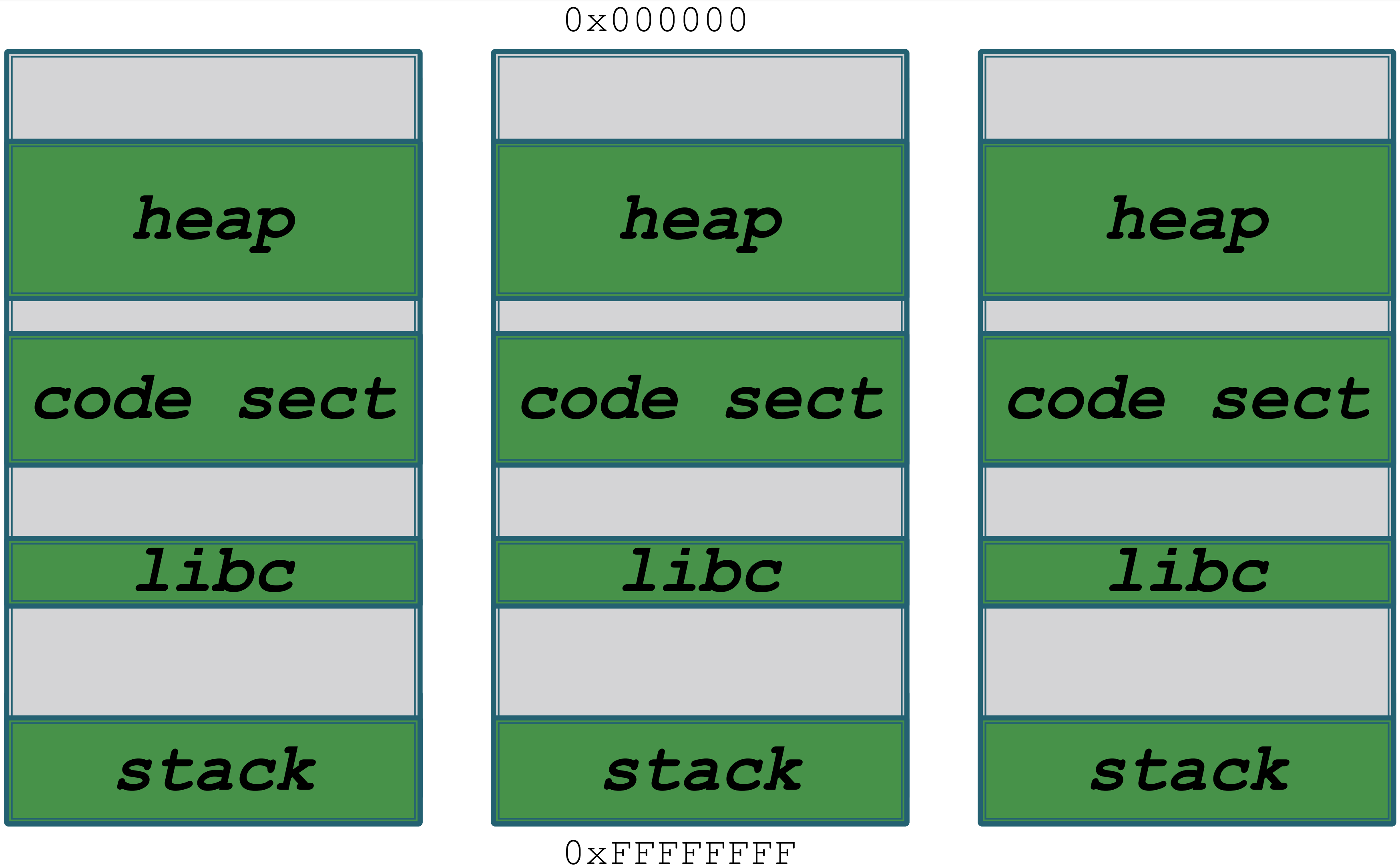


0xFFFFFFFF

Memory Layout (no ASLR)

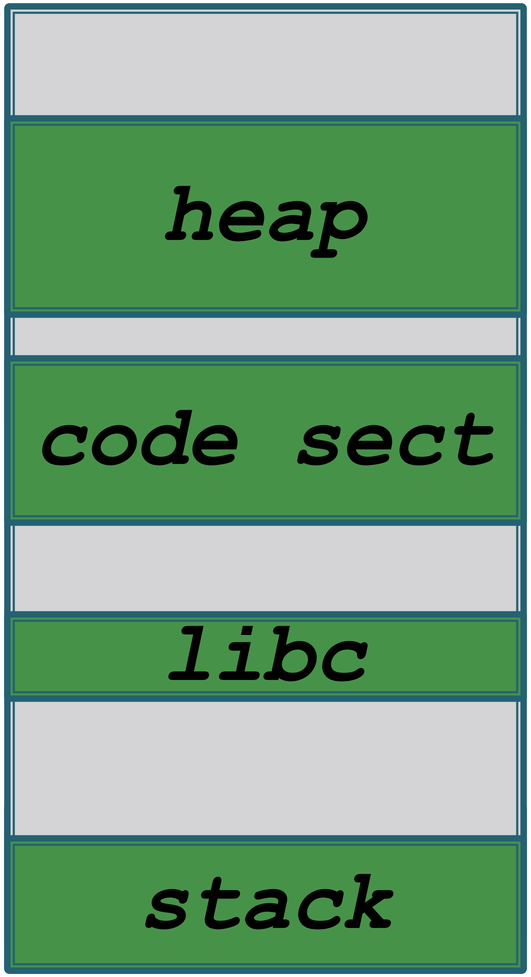


Memory Layout (no ASLR)



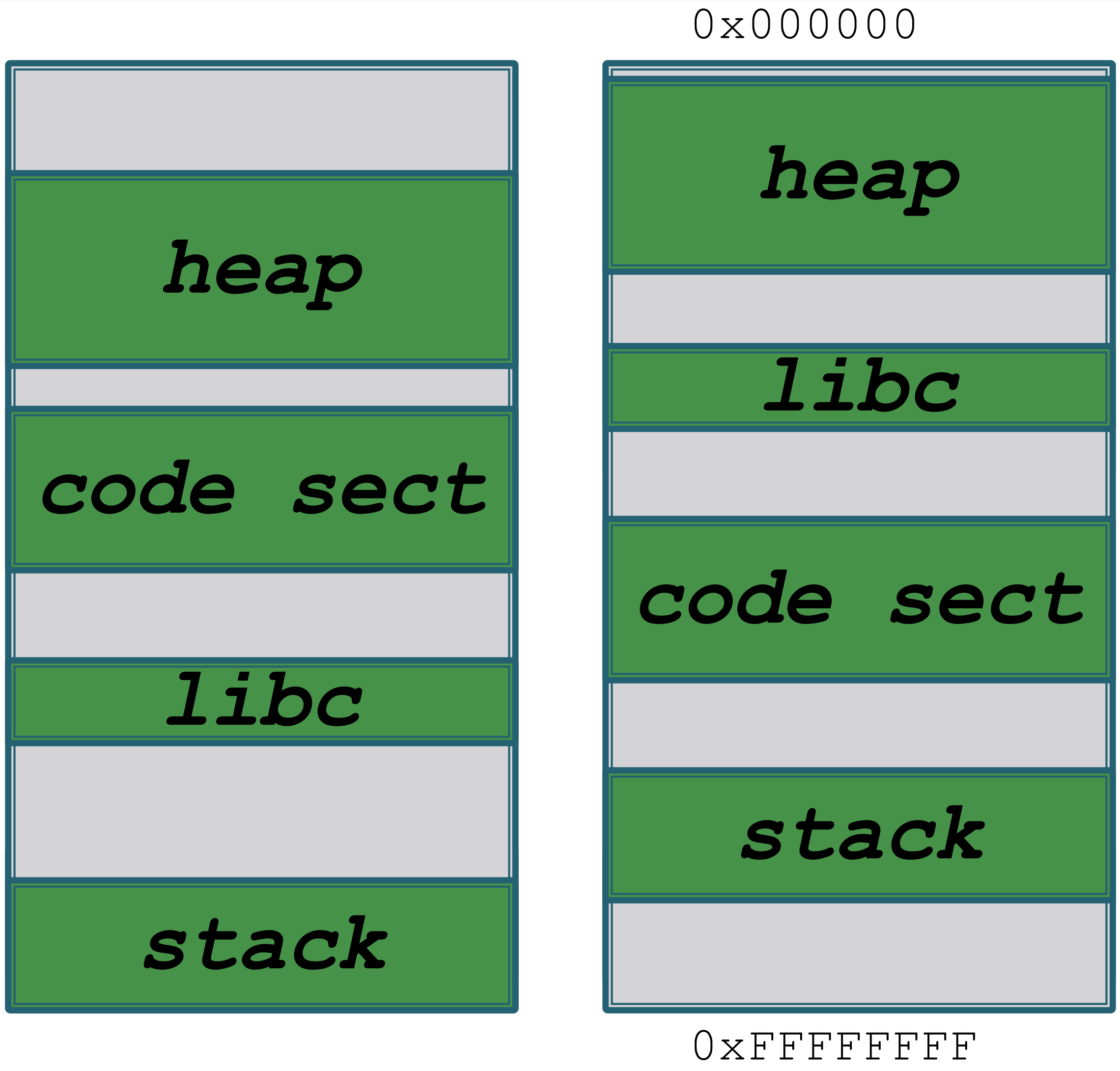
Memory Layout (with ASLR)

0x000000

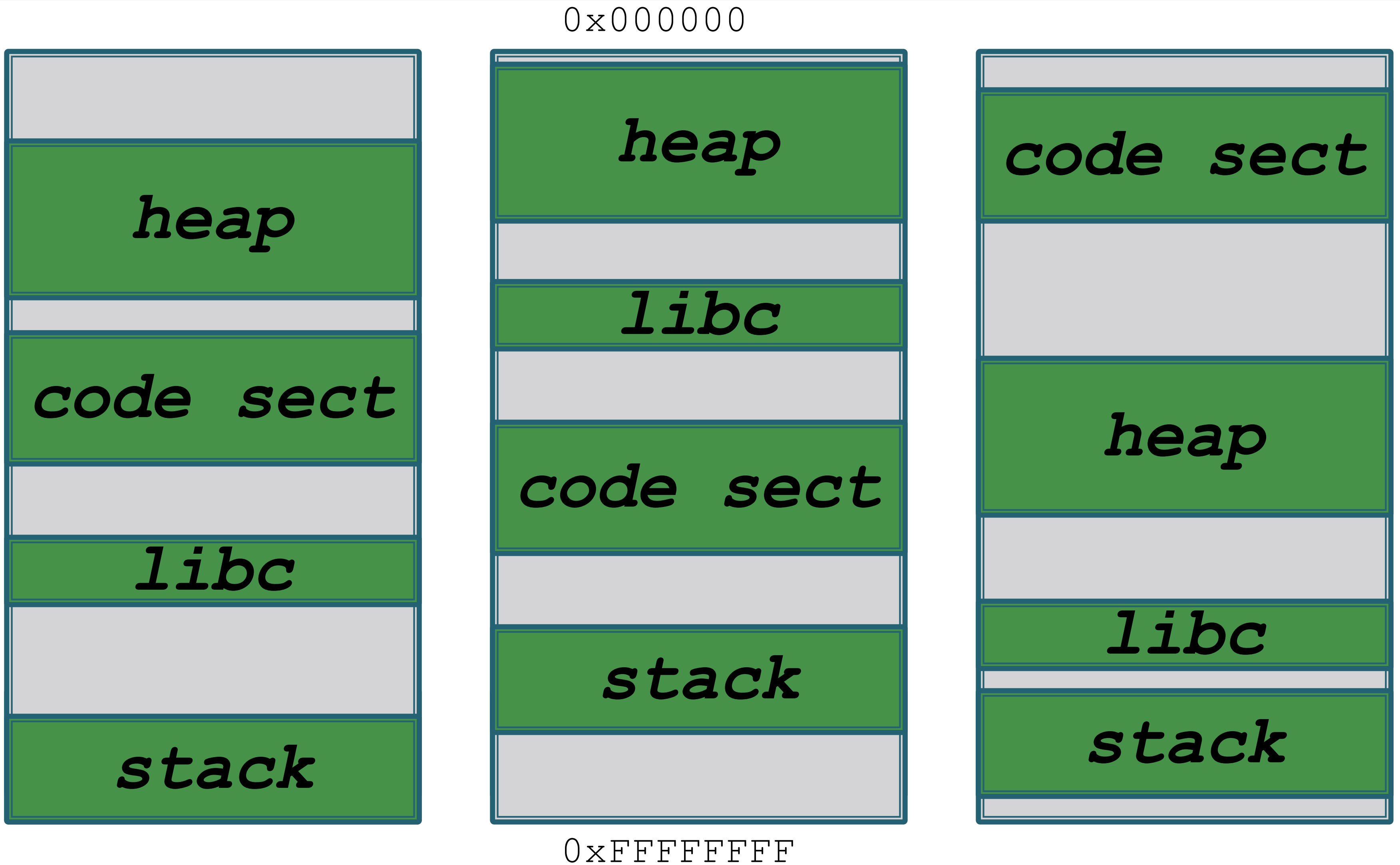


0xFFFFFFFF

Memory Layout (with ASLR)



Memory Layout (with ASLR)



ASLR

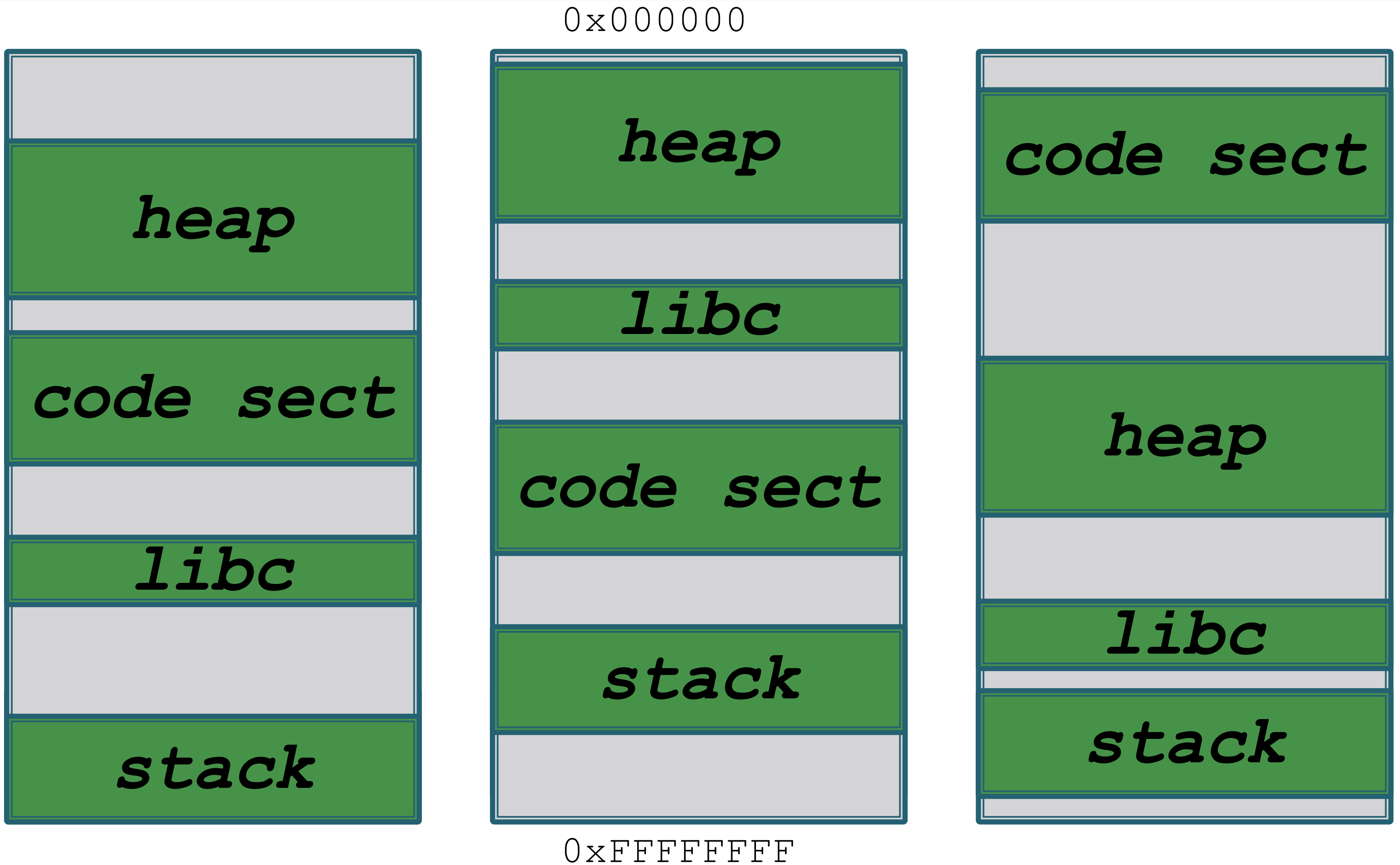
Everything must be relocatable to be effective

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 things within the chunks.

Memory Layout (with ASLR)



ASLR

How can we defeat ASLR?

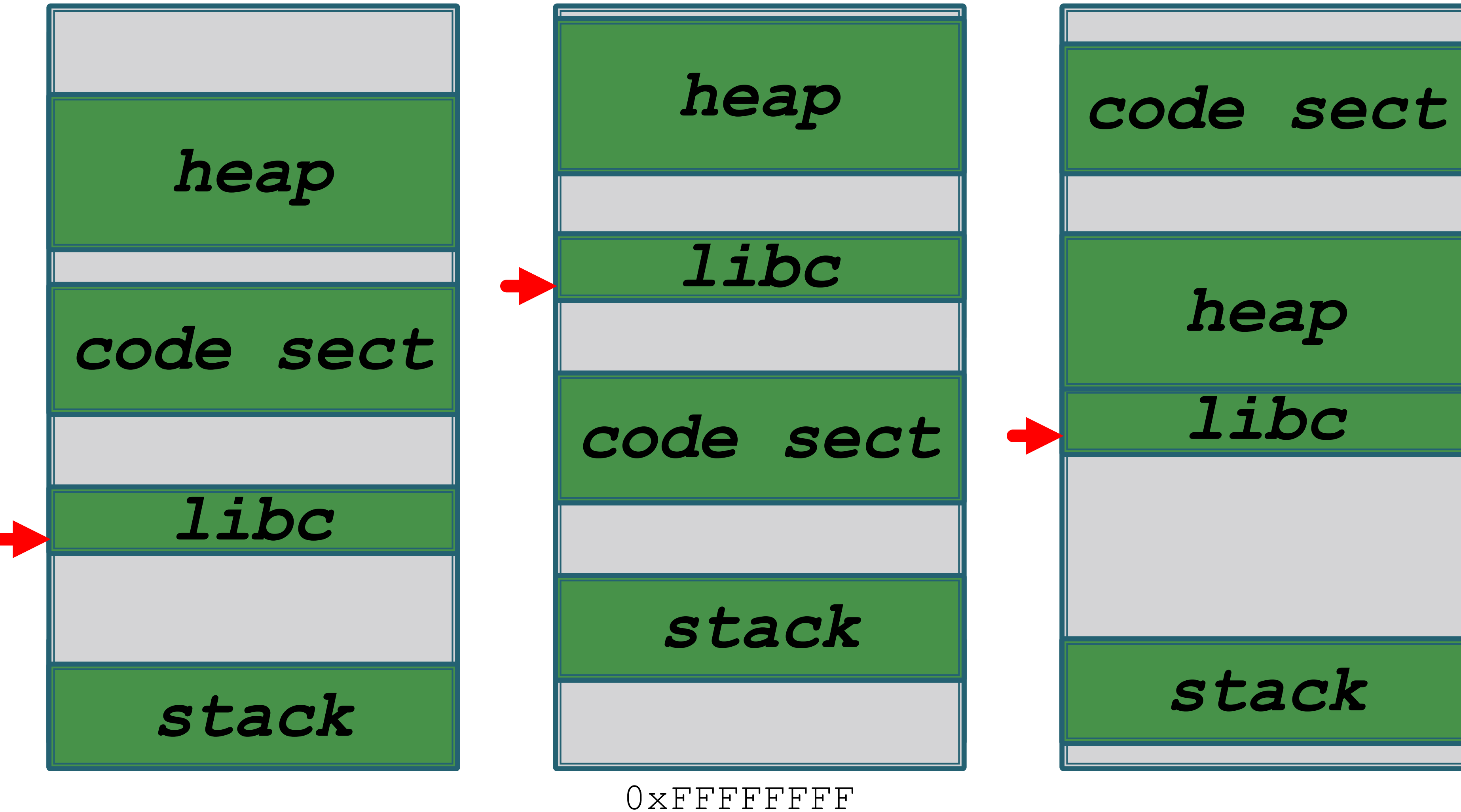
Hint:

All of libc is at a single offset.

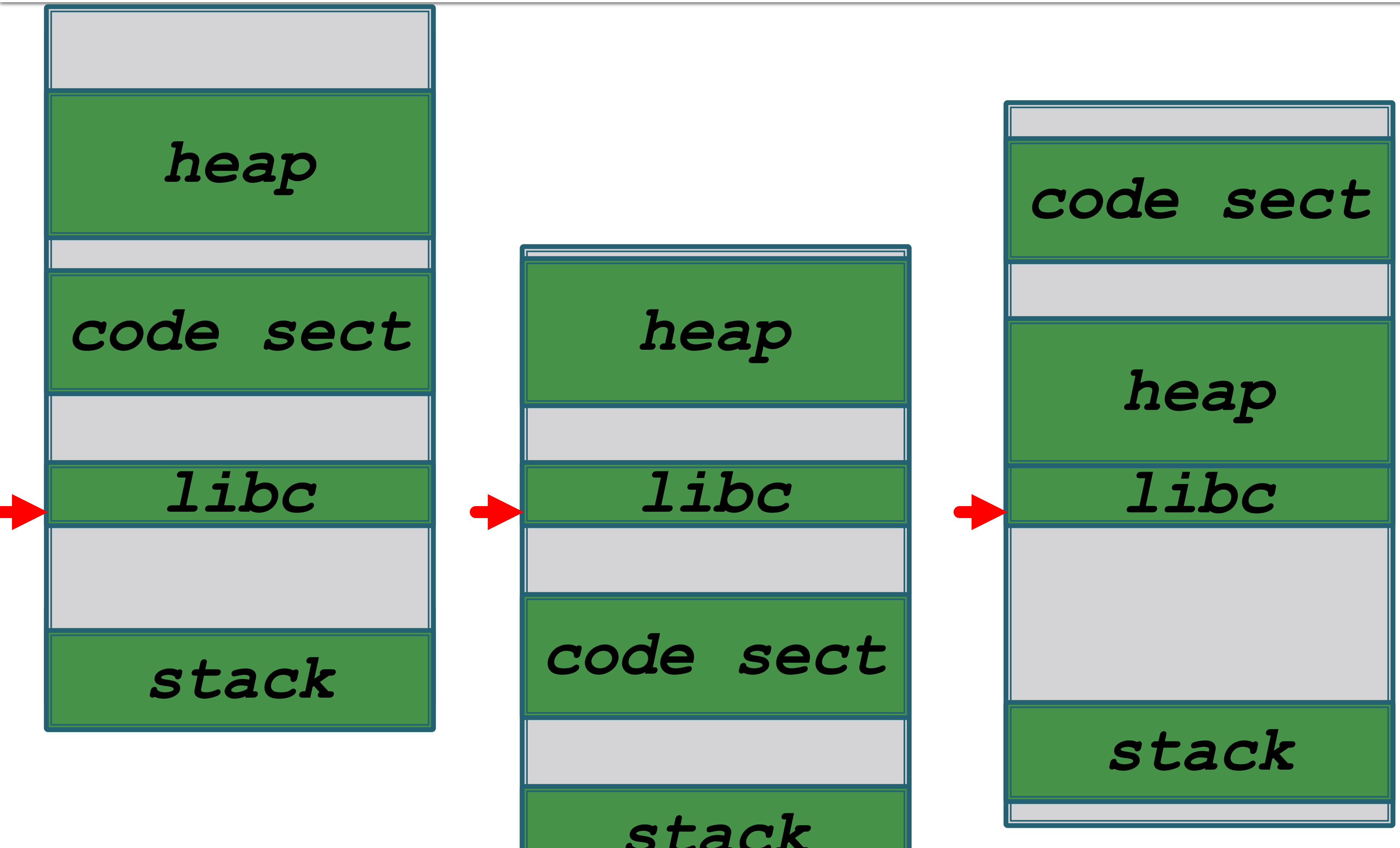
Over-read a single pointer in libc!

0xFFFFFFFF

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 its chunk

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing **Automated Testing**

Toolbox of Exploitation Techniques

Automated Testing

Problem:

Vulnerabilities are hard to find by hand

(and attacks use them ☹)

(and attacks use them ☺)

Solution:

Automate the process!

Automated Testing

Finding vulnerabilities manually is very hard

If source is available:

- Pure-size of possible locations 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 free'd?

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

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

Toolbox of Exploitation Techniques

- Every vulnerability is different
  - Some are not exploitable at all
- Sometime 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?

Data-only attacks

Hypothetical function:

Delete a user from a website.

Username from input field on website.

    Needs to be "canonicalized"

Return 0 on success.

```
int delete_account(char* username,
   int length, VOID* creds);
```

Data-only attacks

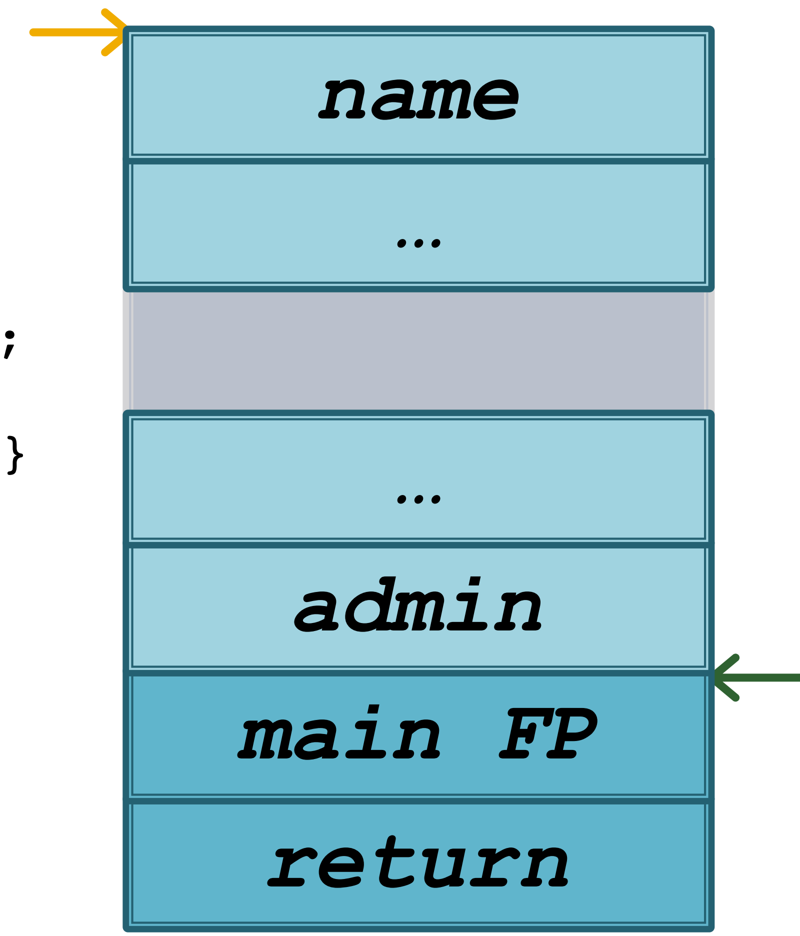
```
int delete_account(char* username,
  int length, VOID* creds) {
    int admin;
    char name[100];
    admin = check_admin(creds);
    strncpy(name, username, length);
    canonicalize_username(name);
    if (admin) {delete_user(name);}
    return (admin > 0);
}
```

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);
}
```

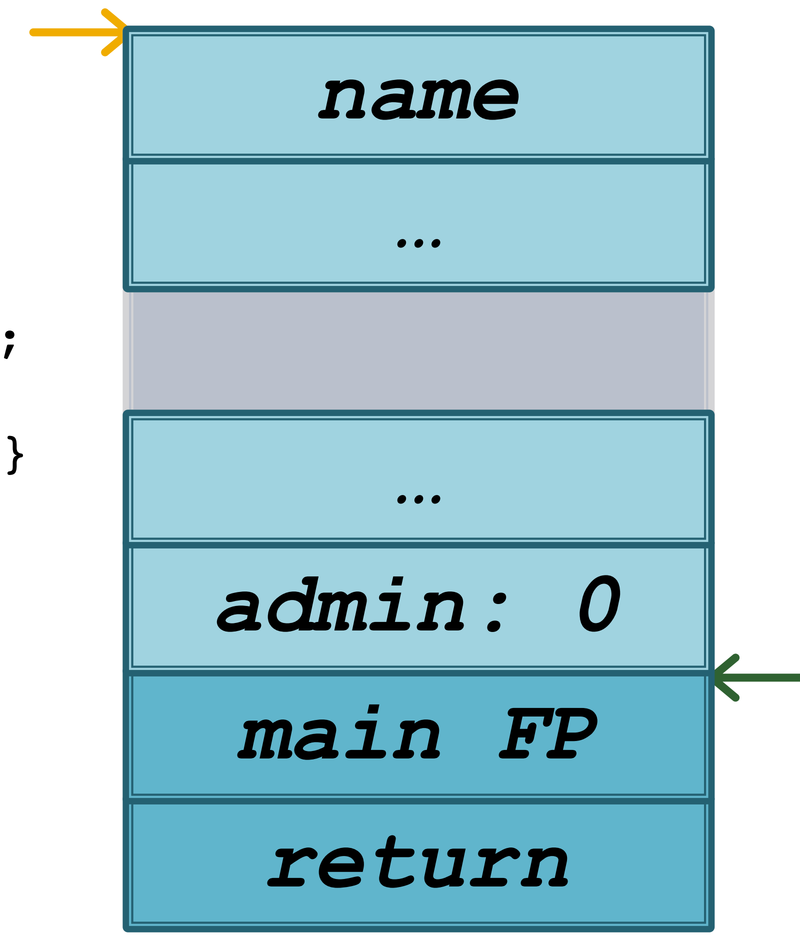
Data-only attacks

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



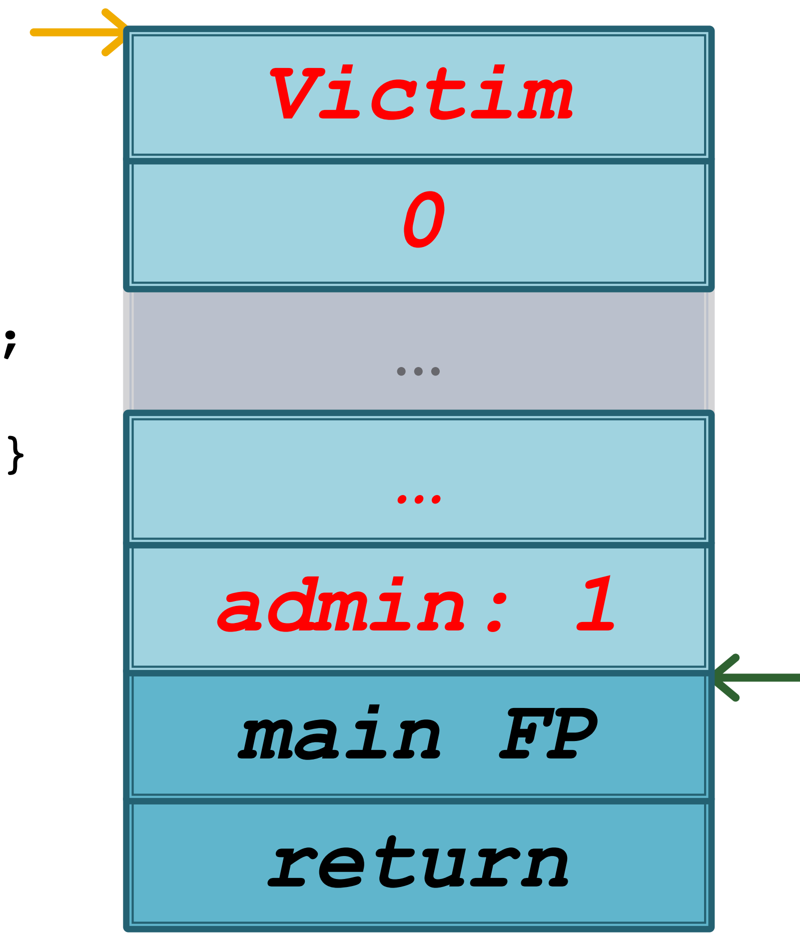
Data-only attacks

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



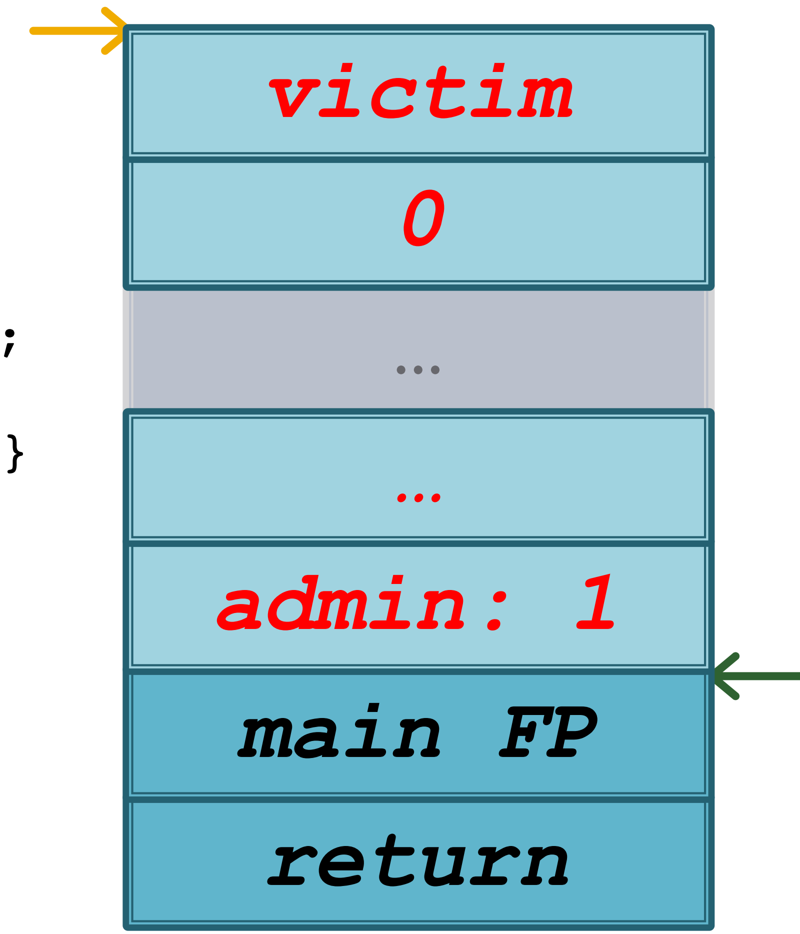
Data-only attacks

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



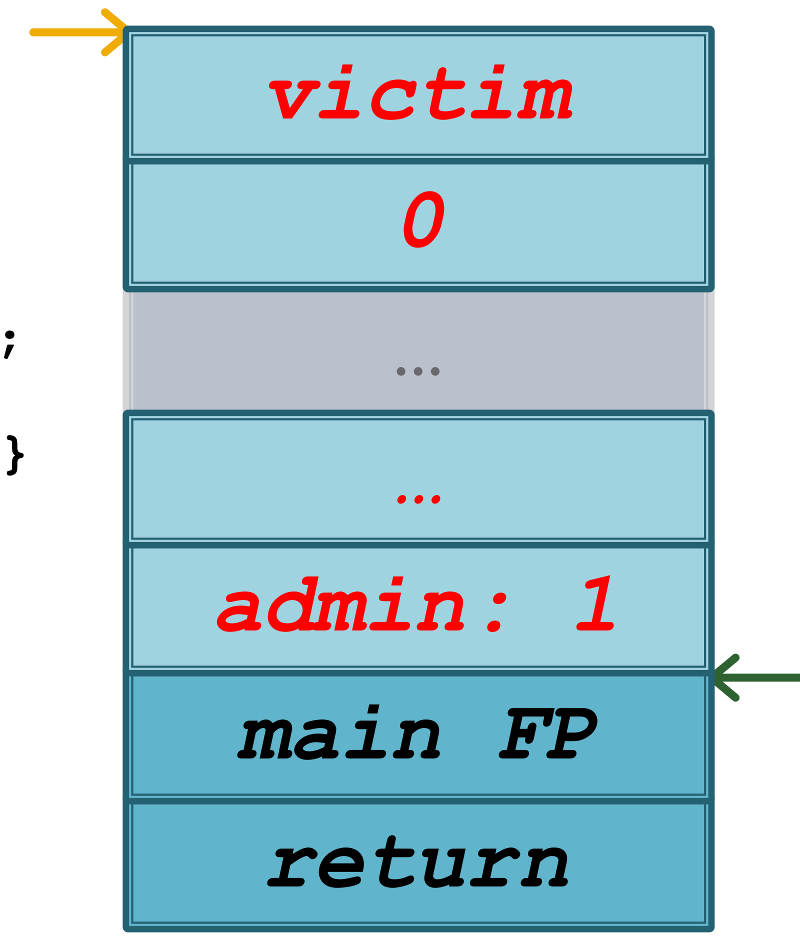
Data-only attacks

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



Data-only attacks

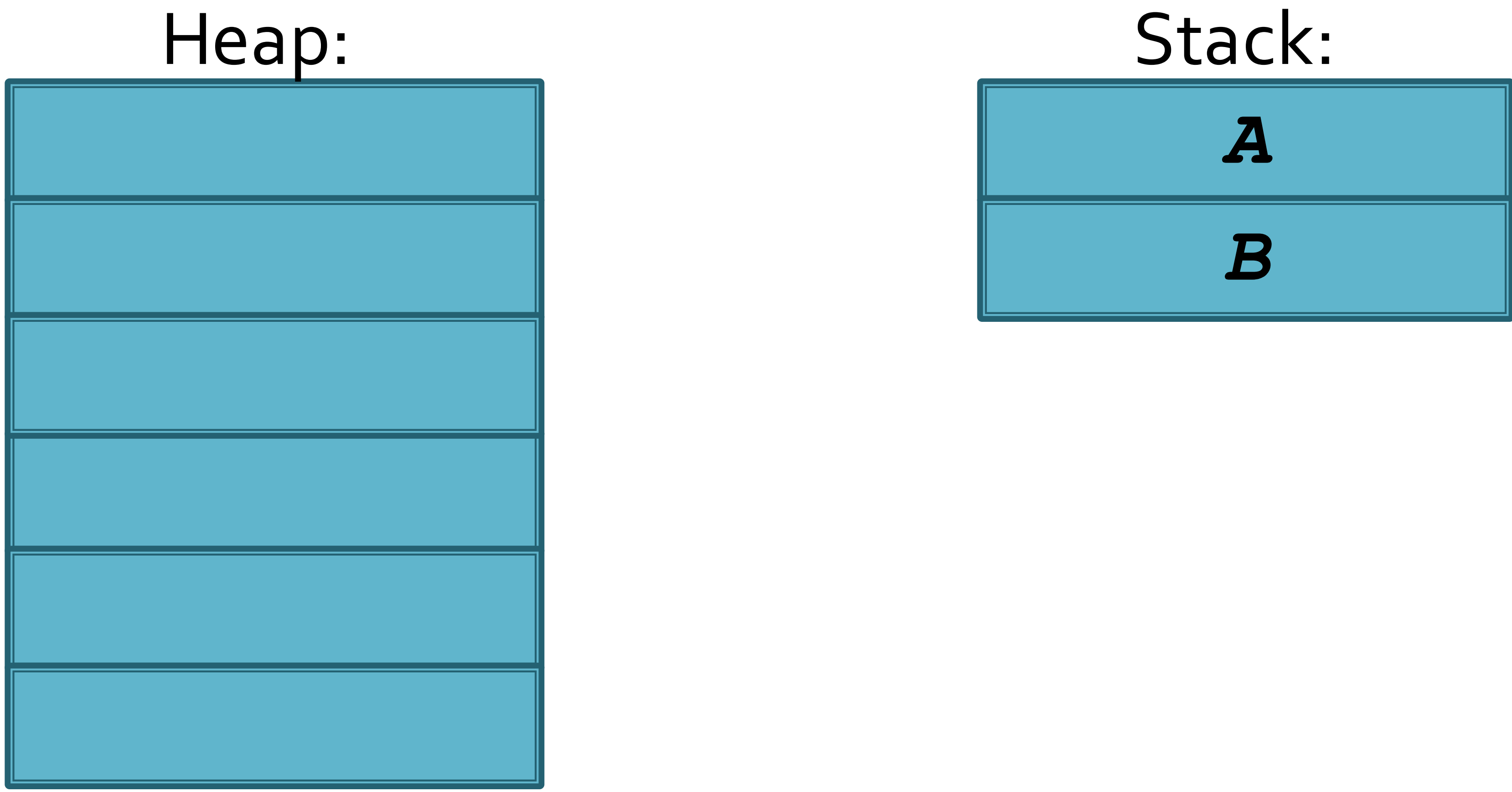
```
int delete_account(char* username,
  int length, VOID* creds) {
      int admin;
      char name[100];
      admin = check_admin(creds);
      strcpy(name, username, length);
      canonicalize_username(name);
      if (admin) {delete_user(name);}
      return (admin > 0);
}
```



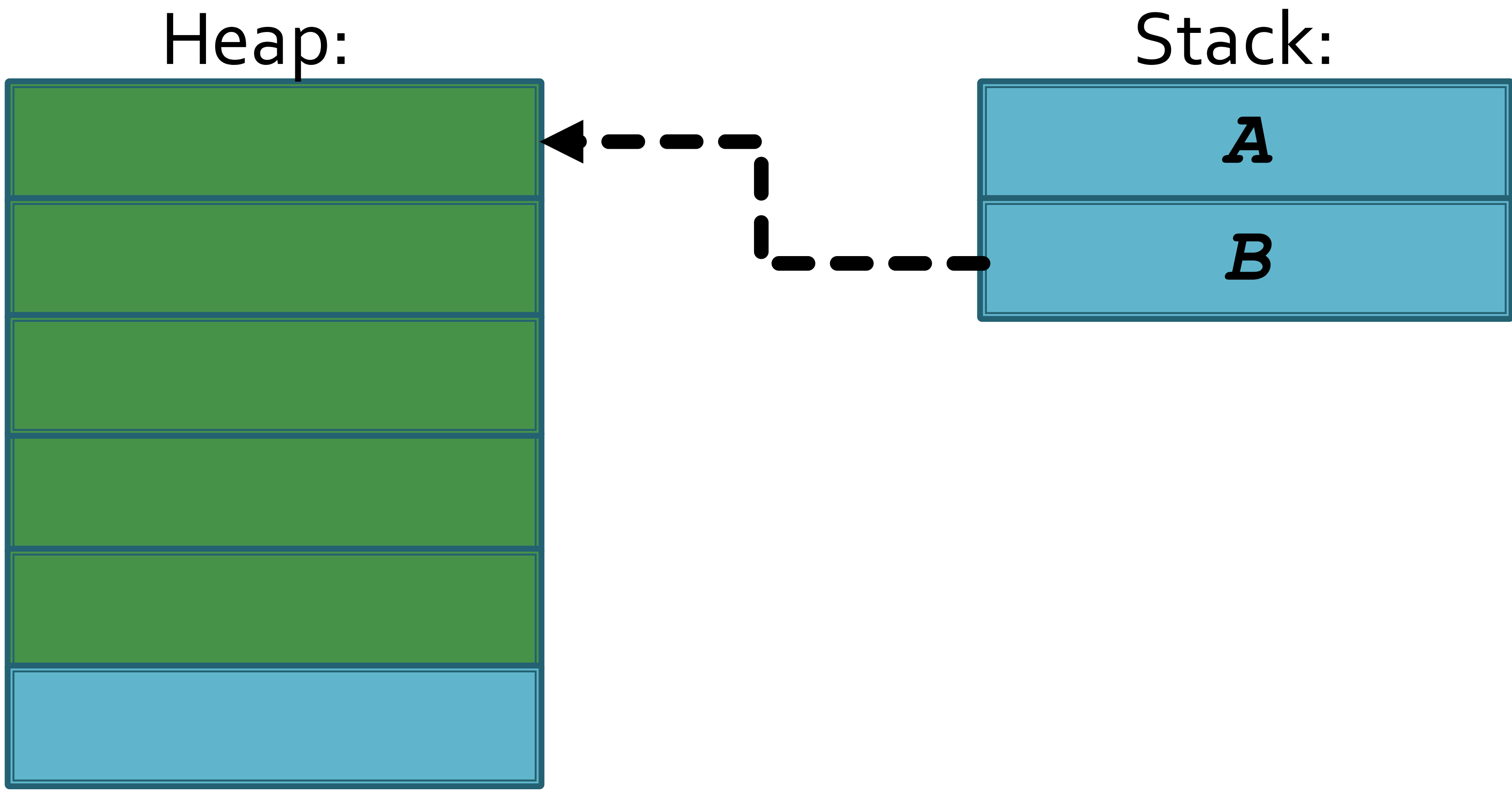
Use After Free

- Common in multi-threaded programs that share variables
  - Though can exist in single threaded programs
- Sometimes caused by a race condition

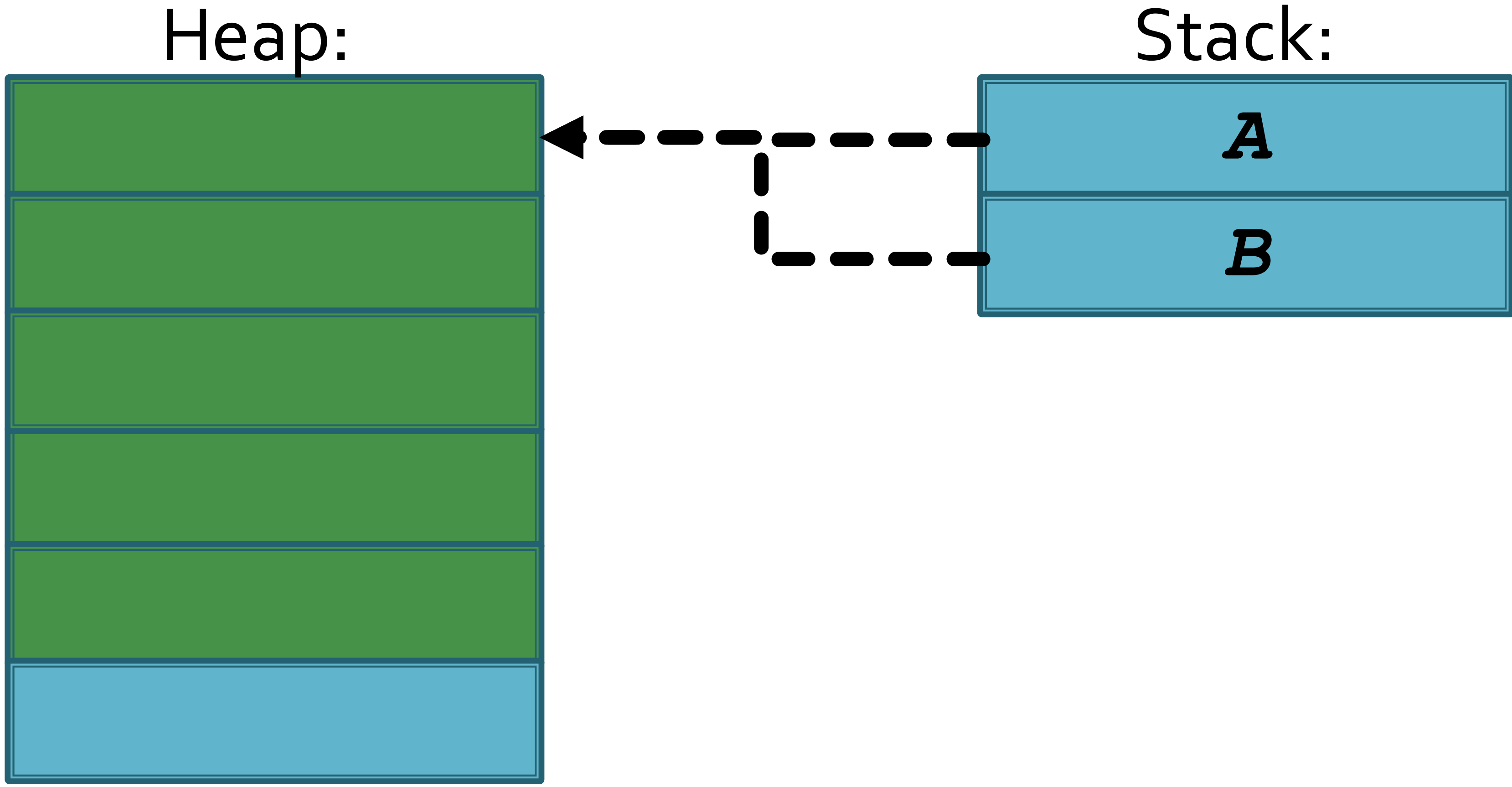
Use After Free



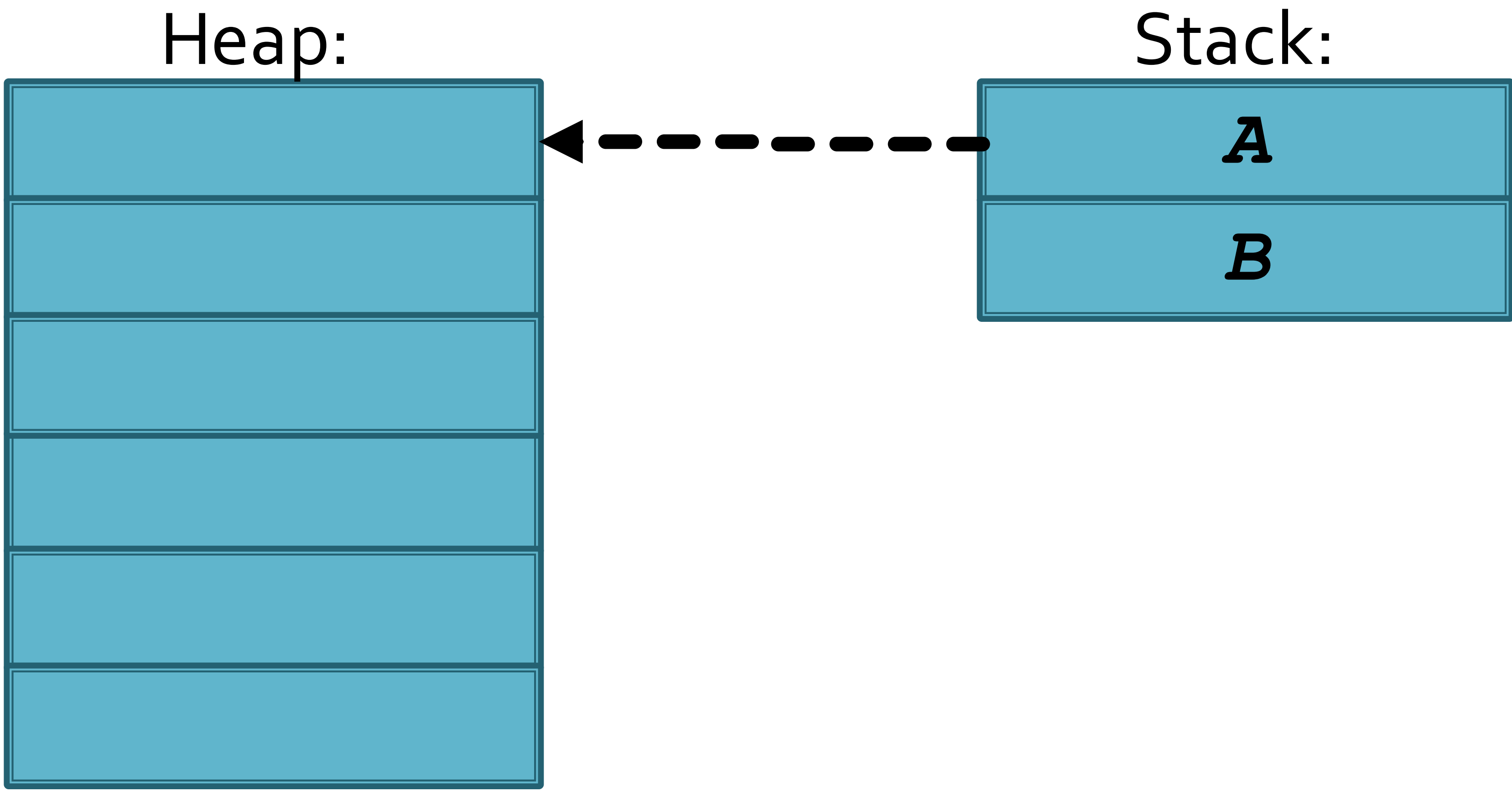
Use After Free



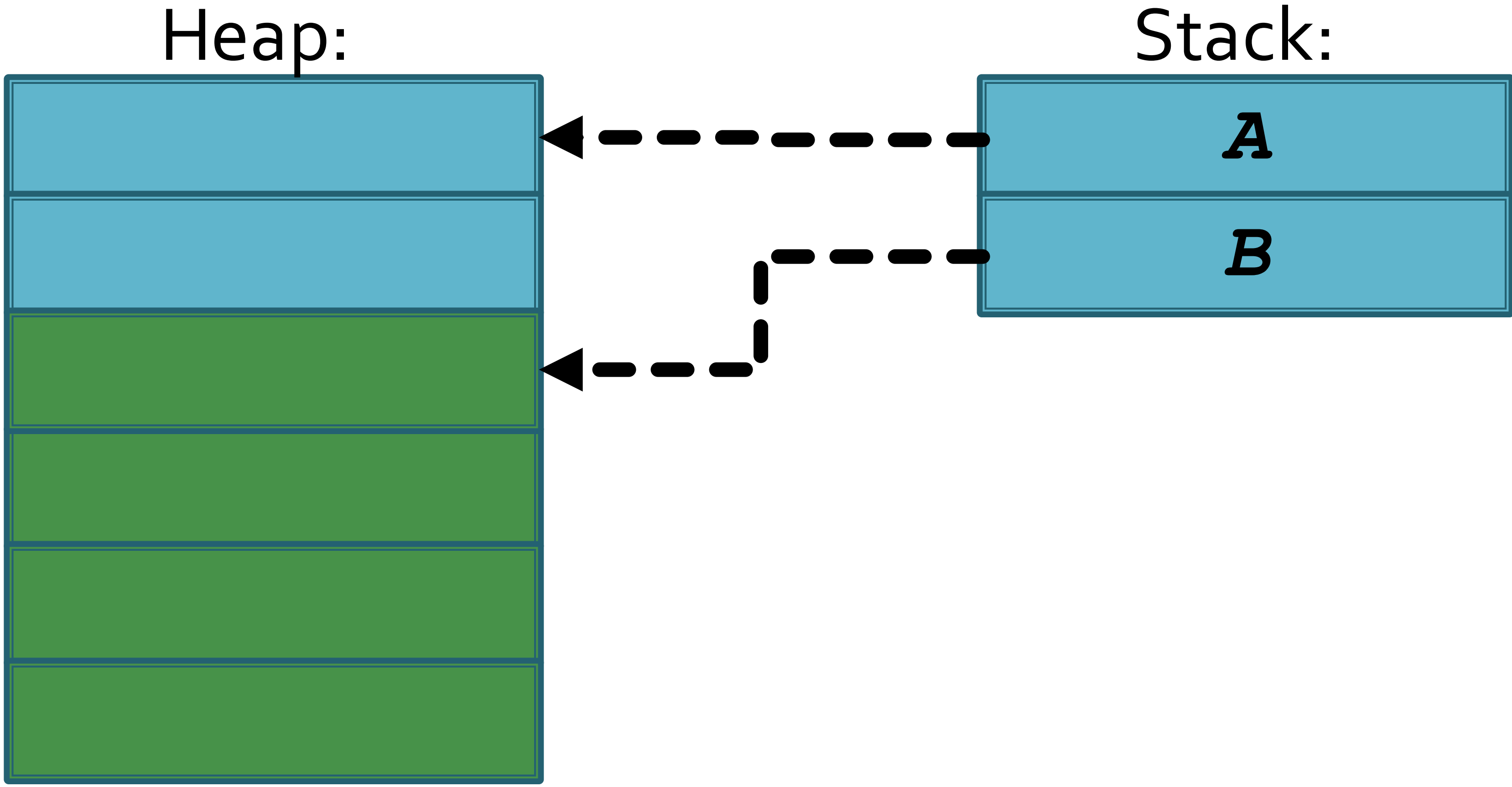
Use After Free



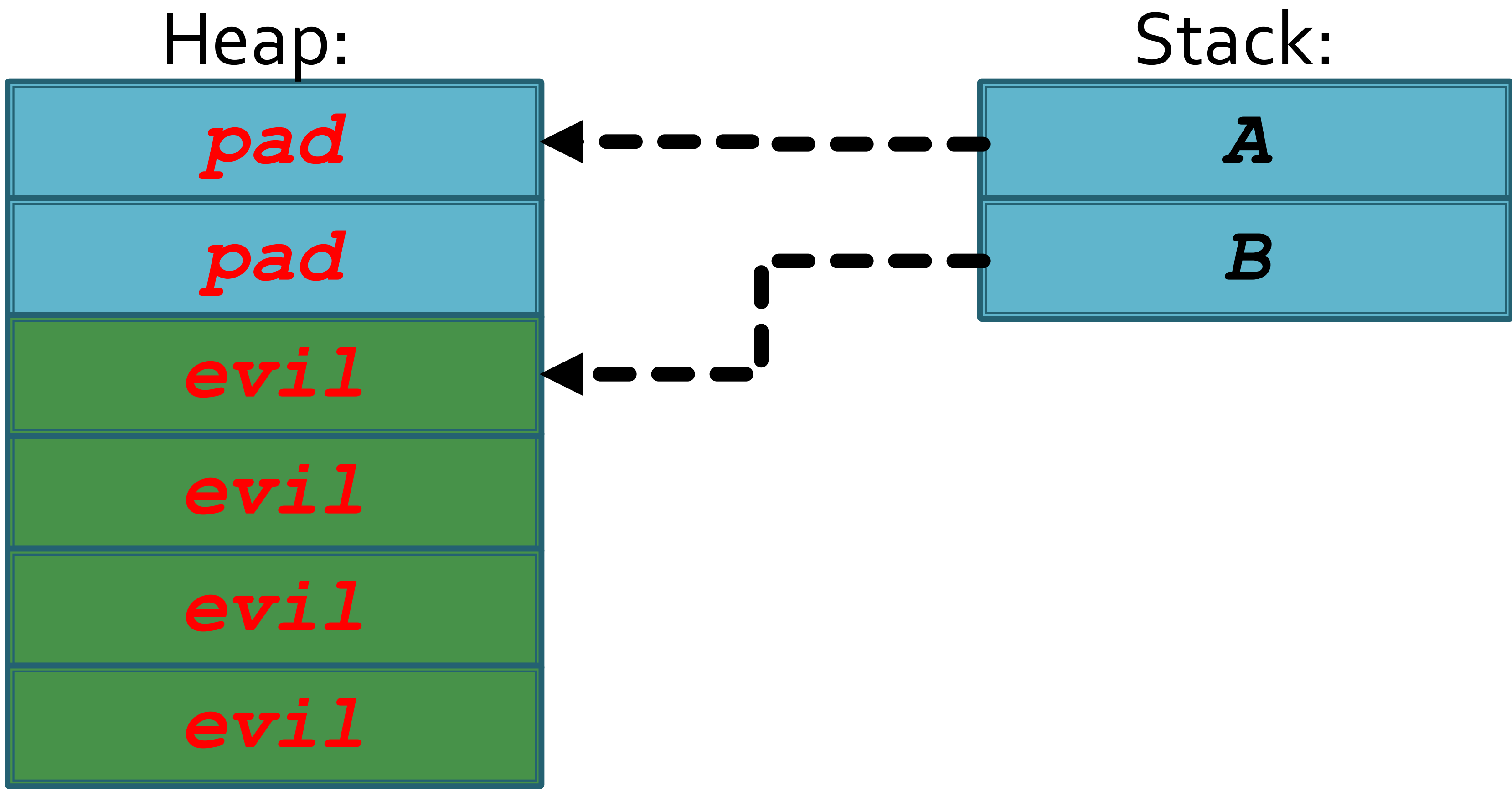
Use After Free



Use After Free



Use After Free



SEH Exploitation

Structured Exception Handling

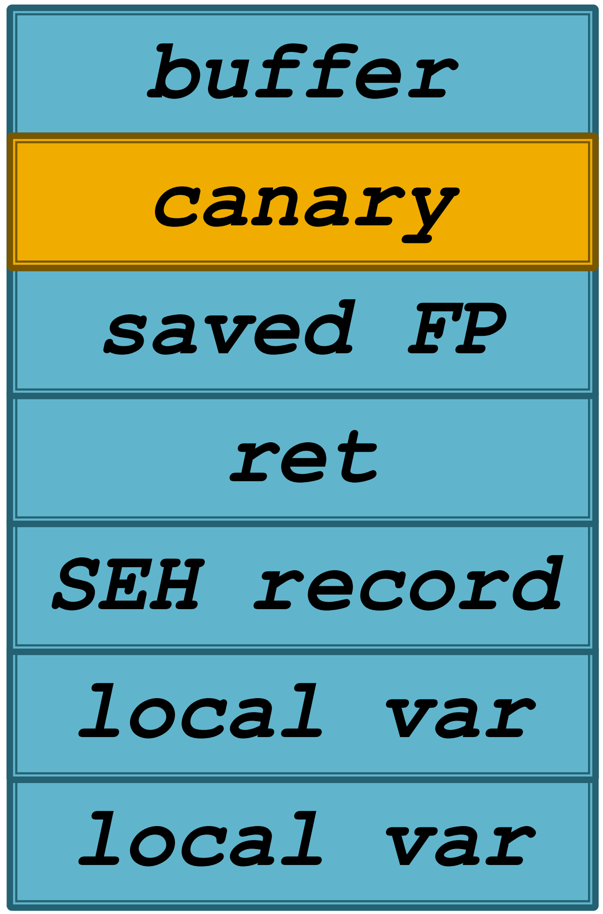
Redirect control flow via the exception hander address *not* the return address

Need a POP-POP-RET ROP Chain

Requires triggering a recoverable exception

  Like realizing that the canary is wrong

SEH Exploitation



SEH Exploitation

Format String Vulnerability

Attack programmer's lack of sanitization

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

Format String Vulnerability

Attack programmer's lack of sanitization

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

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

Format String Vulnerability

Attack programmer's lack of sanitization

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

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

Pops a values off of the stack unexpectedly

Heap Fung Shui

Abuse the heap's memory allocation algorithm

Allocate memory in specific sequences or sizes to influence the address of other allocated memory spaces

Use to increase chances of success

Heap-Spray

Inject data into the application's memory space many times to increase the chances of finding it

Commonly used for web browser exploitation

Less precise than Heap Fung Shui

Egg Hunting

Where vulnerability does not allow enough space for full payload

Pre-load malicious shellcode via heap spraying or simply a controlled write

Use a “finder” in the constrained exploit to find the pre-loaded shellcode and begin execution

References/Acknowledgements

- Aleph One's "Smashing the Stack for Fun and Profit" http://insecure.org/stf/smashstack.html
- Paul Makowski's "Smashing the Stack in 2011" http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/
- Blexim's "Basic Integer Overflows" http://www.phrack.org/issues.html?issue=60&id=10
- Return-to-libc demo http://www.securitytube.net/video/258

- Thank you prior slide authors!