

Control Hijacking (day 1)

EECS 388: Introduction to Computer Security

Ben VanderSloot

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

Control Hijacking

We aren't talking about:

Simple passwords that can be guessed

The password checker is functioning correctly

Tricking someone into installing malware

The malware is functioning correctly

KRACK Attack on WPA2

The implementation matches the protocol, and executes it correctly

Control Hijacking

Definition:

Binary exploitation is the process of subverting a compiled application such that it violates some trust boundary in a way that is advantageous to you, the attacker. In this module we are going to focus on memory corruption.

Trail of Bits CTF Field Guide
(Creative Commons Attribution Share Alike 4.0)

Control Hijacking



A Top-Shelf iPhone Hack Now Goes for \$1.5 Million

SUBSCRIBE

BUSINESS

CULTURE

DESIGN

GEAR

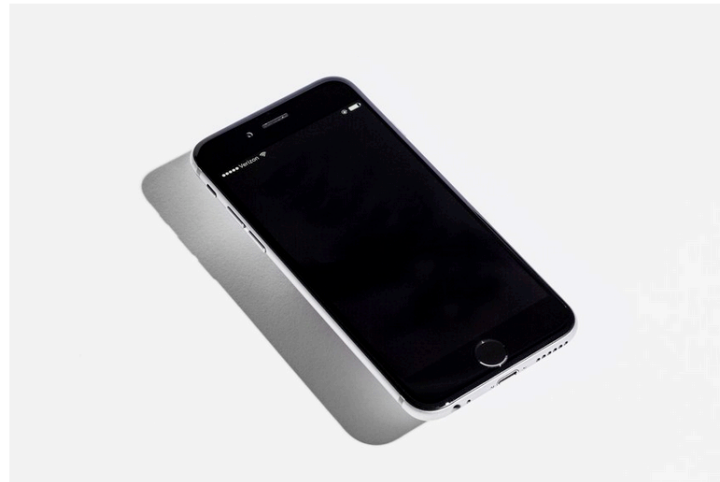
SCIENCE

SECURITY

TRANSPORTATION

LILY HAY NEWMAN SECURITY 09.29.16 6:00 PM

A TOP-SHELF IPHONE HACK NOW GOES FOR \$1.5 MILLION



MOST POPULAR



SECURITY
Inside the Cyberattack
That Shocked the US
Government
1 DAY



AUTONOMOUS VEHICLES
Tesla's Self-Driving Car
Plan Seems Insane, But It
Just Might Work
11 HOURS



SECURITY
What We Know About
Friday's Massive East
Coast Internet Outage
10.21.16

→ MORE STORIES

Control Hijacking



Got a tip? [Let us know.](#)

[News](#) [Video](#) [Events](#) [Crunchbase](#)

Follow Us [f](#) [in](#) [g+](#) [t](#) [v](#) [p](#) [r](#)

[Message Us](#)

[Search](#)



DISRUPT BERLIN Time is running out on Early Bird savings for Disrupt Berlin. [Save 30% off tickets today](#) ▶

[ukraine](#)
[cyber attack](#)
[cybersecurity](#)
[ransomware](#)
[Hack](#)

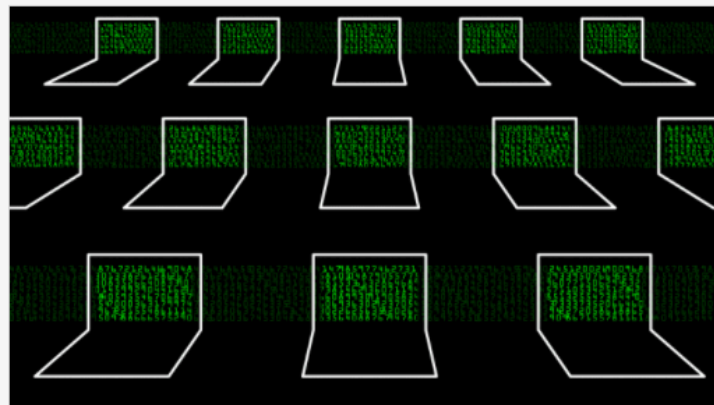
[Popular Posts](#)

A massive new ransomware attack is spreading around the globe

Posted Jun 27, 2017 by [Taylor Hatmaker](#) (@tayhatmaker)



[Next Story](#)



A month after the [WannaCry ransomware](#) attack paralyzed connected systems worldwide, a new threat appears to be spreading quickly.

As reports emerge, today's attack paints a picture of businesses and governments around the world held hostage by a second major wave of ransomware, a kind of software that hijacks computerized systems and demands payment, often in bitcoin, to unlock them.

NEWSLETTER SUBSCRIPTIONS

- ☐ **The Daily Crunch**
Get the top tech stories of the day delivered to your inbox
- ☐ **TC Weekly Roundup**
Get a weekly recap of the biggest tech stories
- ☐ **Crunchbase Daily**
The latest startup funding announcements

Enter your email

[SUBSCRIBE](#)

protected by reCAPTCHA
[Privacy](#) [Terms](#)



[SEE ALL NEWSLETTERS](#) >>

Outline

CPU

- Registers

- Instructions

Call stack

- Stack in assembly

- Stack frames

- Technical details

Buffer overflows

Adapted from Aleph One's "Smashing the Stack for Fun and Profit"

CPU - Registers

Assembly Language

x86, AMD, MIPS, PowerPC

General Purpose Registers

EAX, EBX, ECX, EDX, EDI, ESI

Special Purpose Registers:

EIP: Instruction Pointer

ESP: Stack Pointer

EBP: Frame/Base Pointer

CPU - Instructions

Move a value to a register

```
mov eax, 0x34
```

Add a value to a register

```
add eax, 10
```

Change execution path

```
jmp 0x12345678      # don't return
```

```
call 0x12345678     # do return
```


Stack

```
Stack myStack = Stack();  
myStack.push(1);  
myStack.push(2);  
myStack.pop();           // returns 2
```

Stack

SP →

A diagram showing a stack pointer. The letters "SP" are on the left, followed by a yellow arrow pointing to the right. The arrow points to a horizontal line consisting of two parallel black lines, representing the stack boundary.

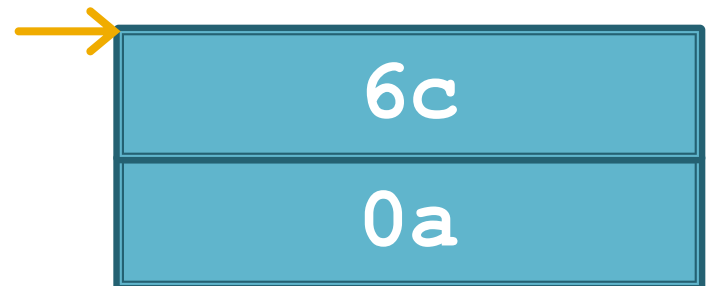
Stack

push 0x0a



Stack

```
push 0x0a  
push 0x6c
```

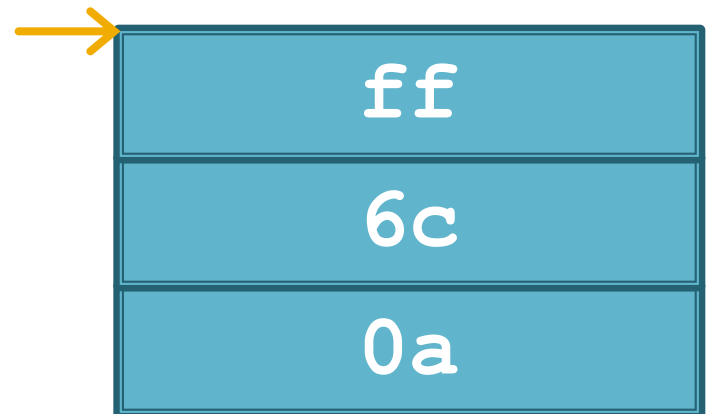


Stack

push 0x0a

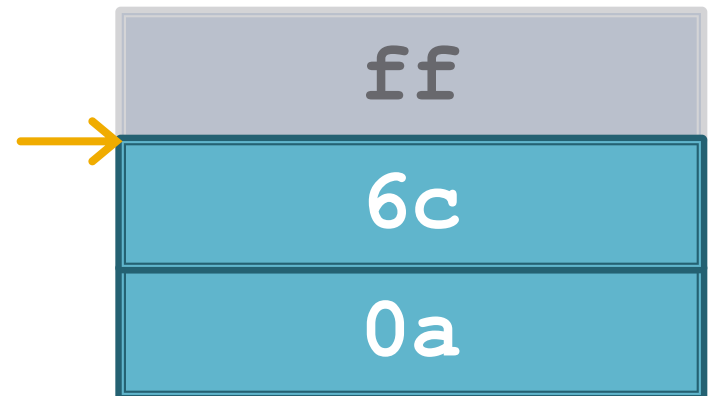
push 0x6c

push 0xff



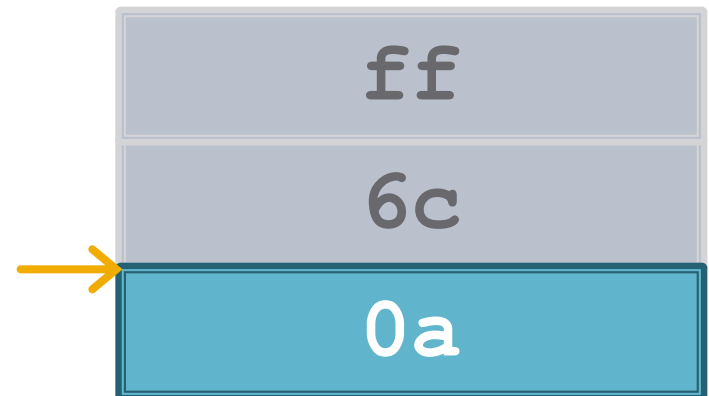
Stack

```
push 0x0a  
push 0x6c  
push 0xff  
pop  eax    #0xff
```



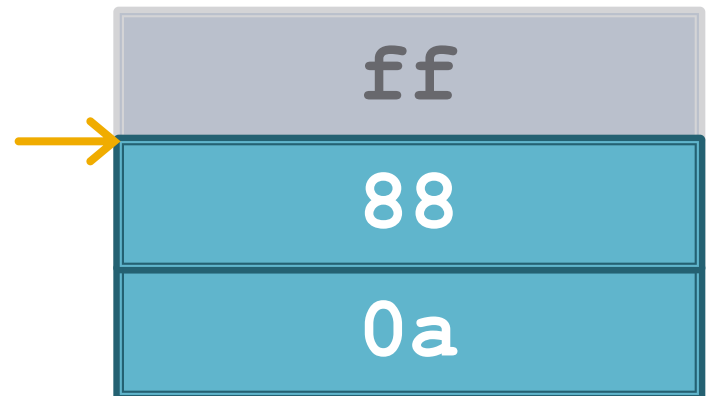
Stack

```
push 0x0a  
push 0x6c  
push 0xff  
pop  eax    #0xff  
pop  eax    #0x6c
```



Stack

```
push 0x0a
push 0x6c
push 0xff
pop  eax    #0xff
pop  eax    #0x6c
push 0x88
```

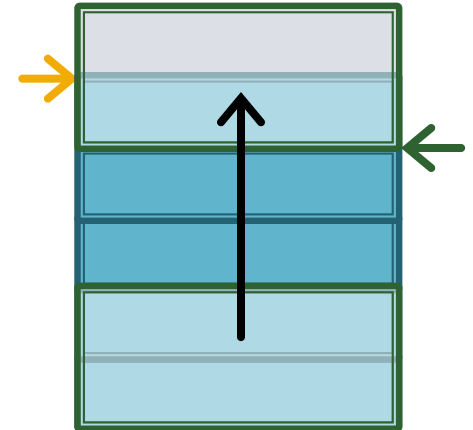


Stack frames

Starts at `0xffffffff`

Grows toward `0x00000000`
(x86 specific)

Low address `0x00`



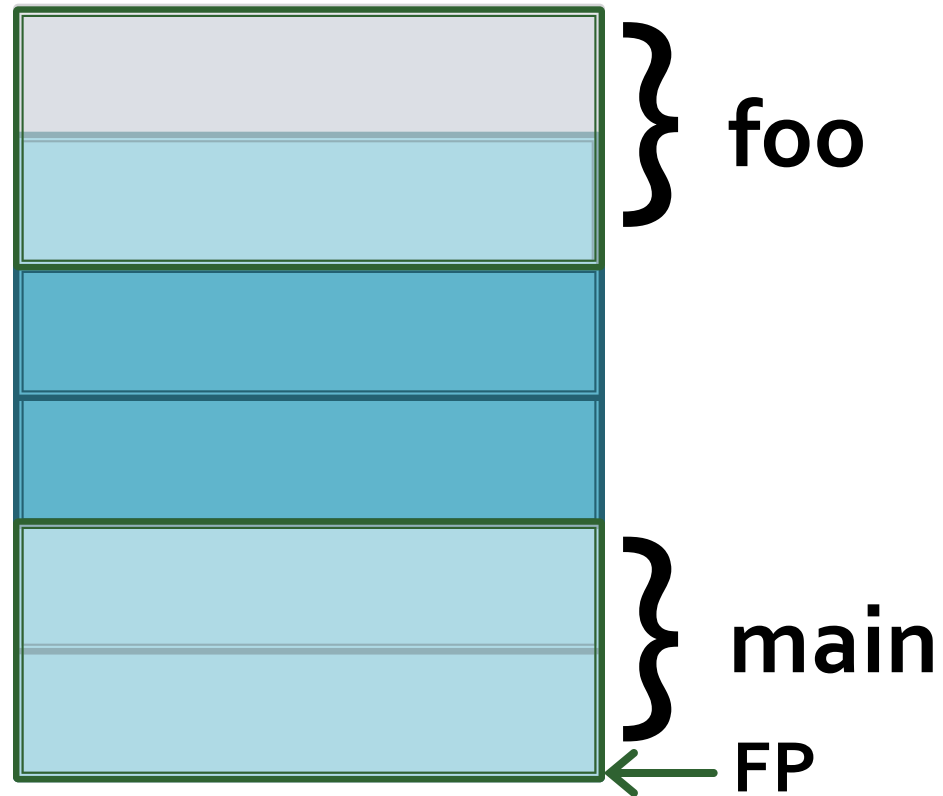
High address `0xff`

example.c

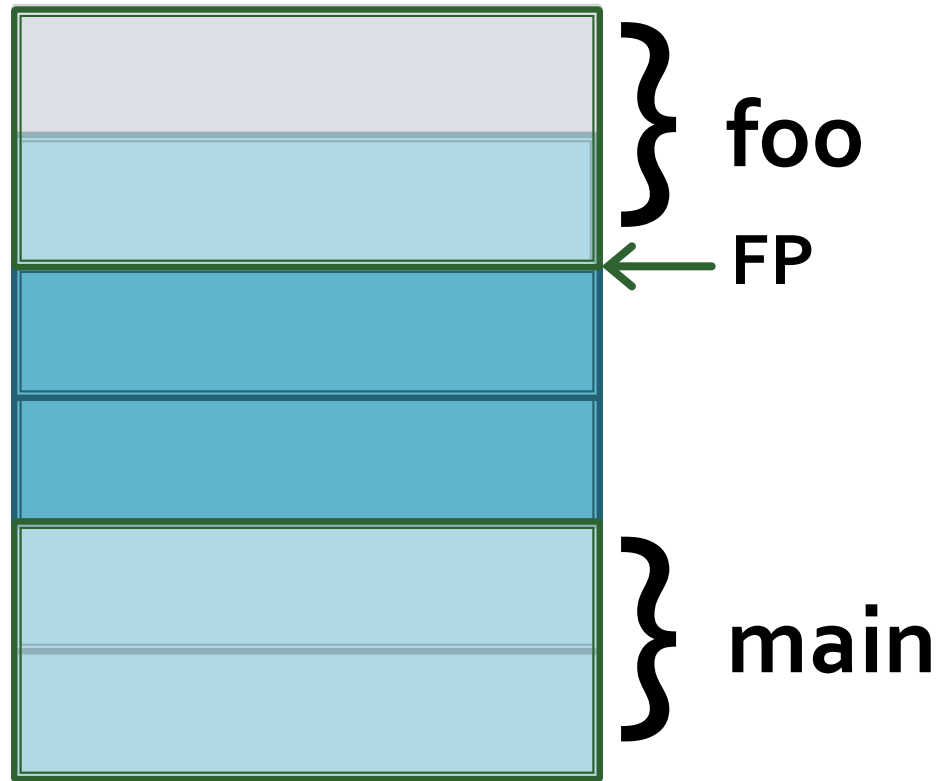
```
void foo(int a, int b) {  
    char buf1[16];  
}
```

```
int main() {  
    foo(3, 6);  
}
```

Stack frames



Stack frames



Stack frames

Starts at `0xffffffff`

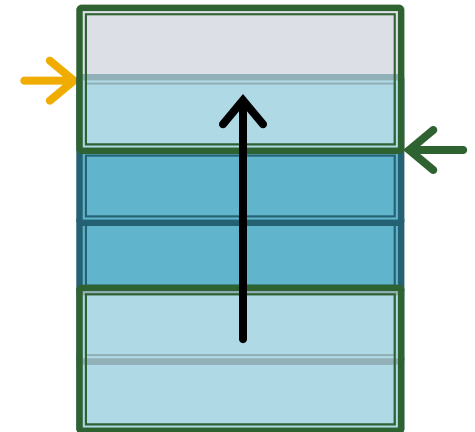
Grows toward `0x00000000`

X86 specific

Stack Pointer (ESP): →

Frame Pointer (EBP): ←

Low address `0x00`



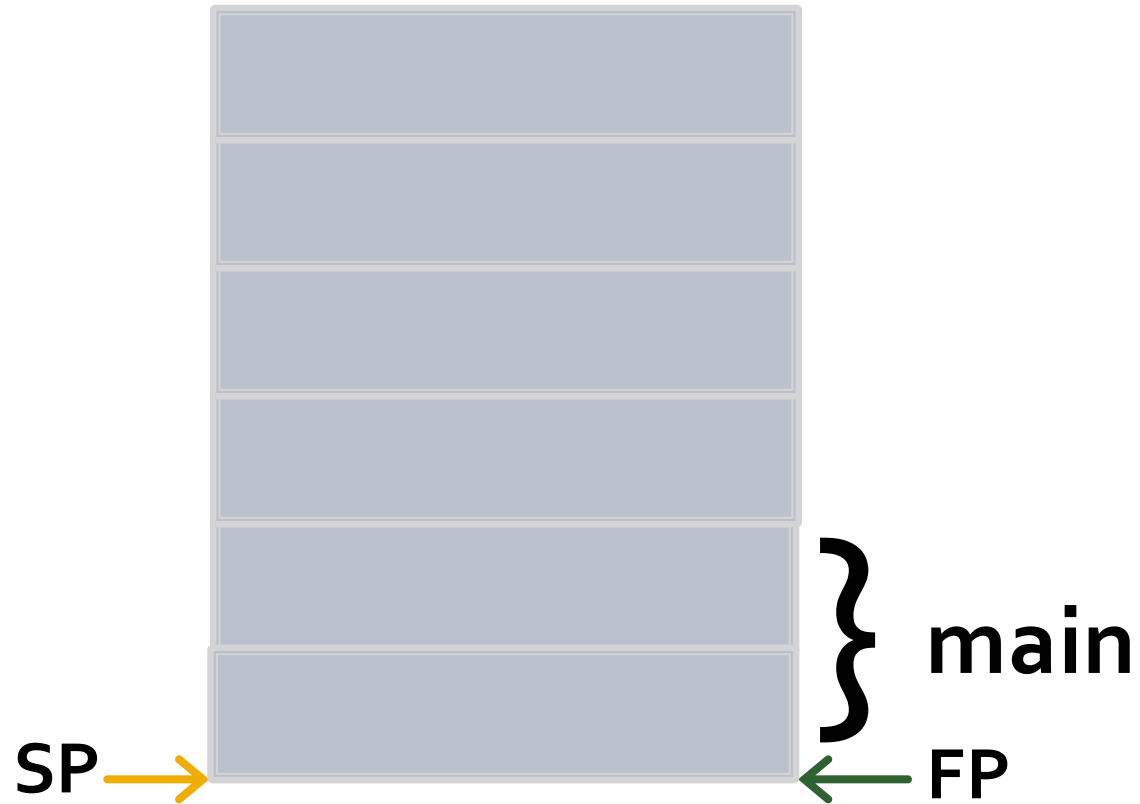
High address `0xff`

example.c

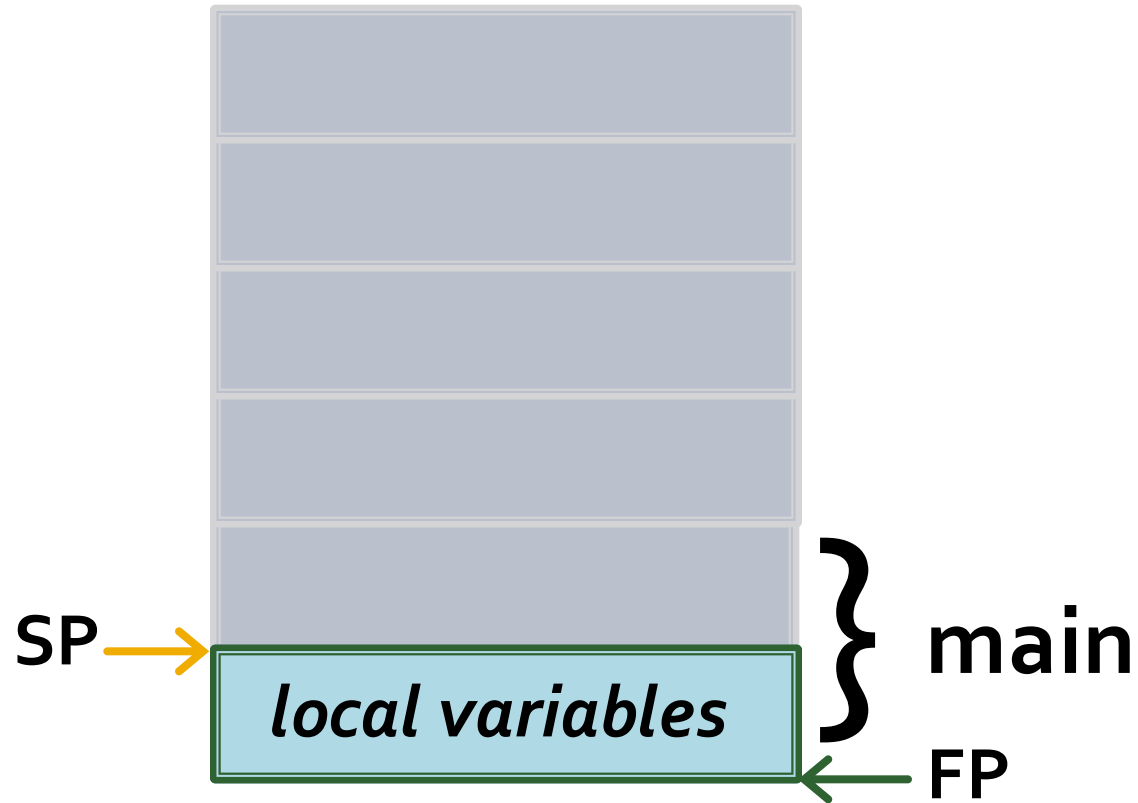
```
void foo(int a, int b) {  
    char buf1[16];  
}
```

```
int main() {  
    foo(3, 6);  
}
```

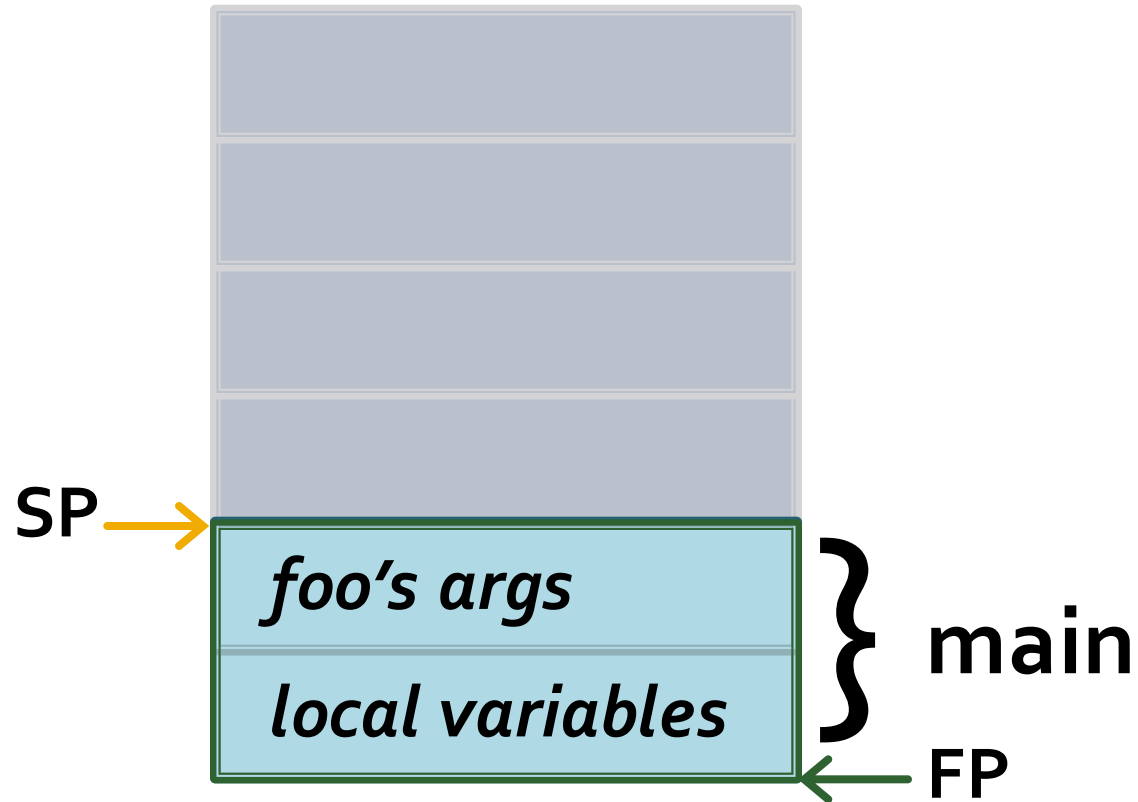
Stack frames



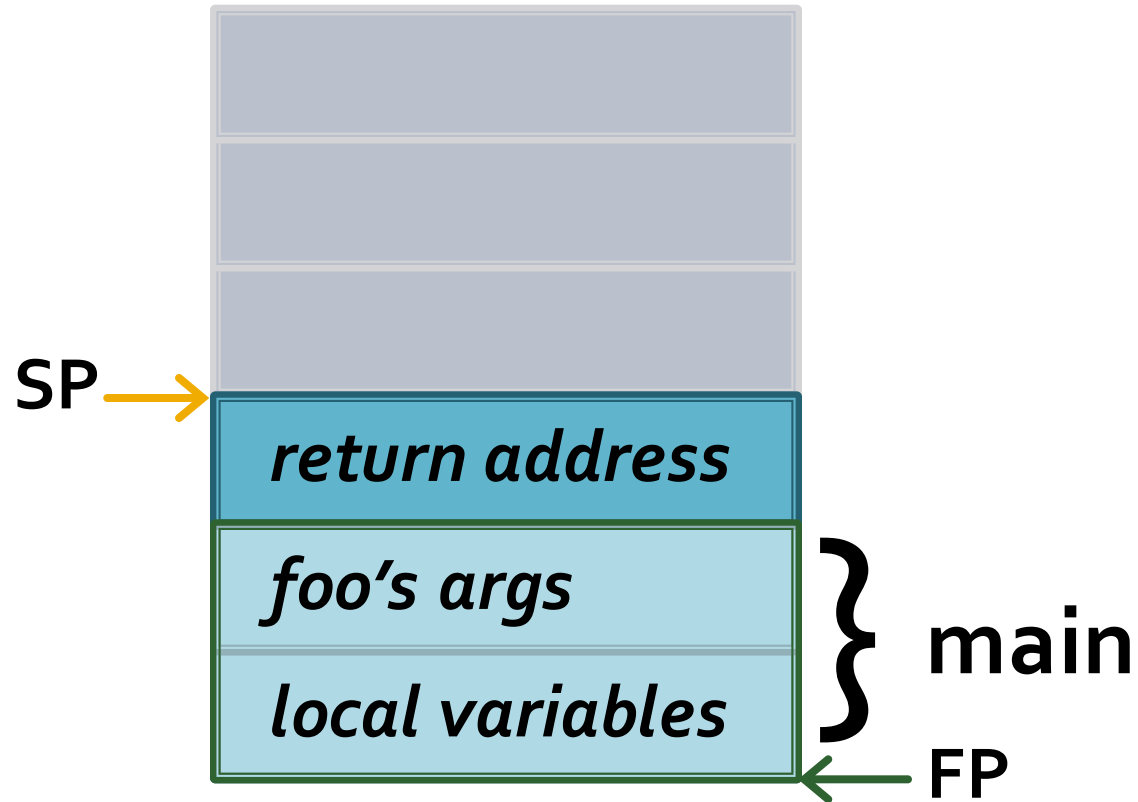
Stack frames



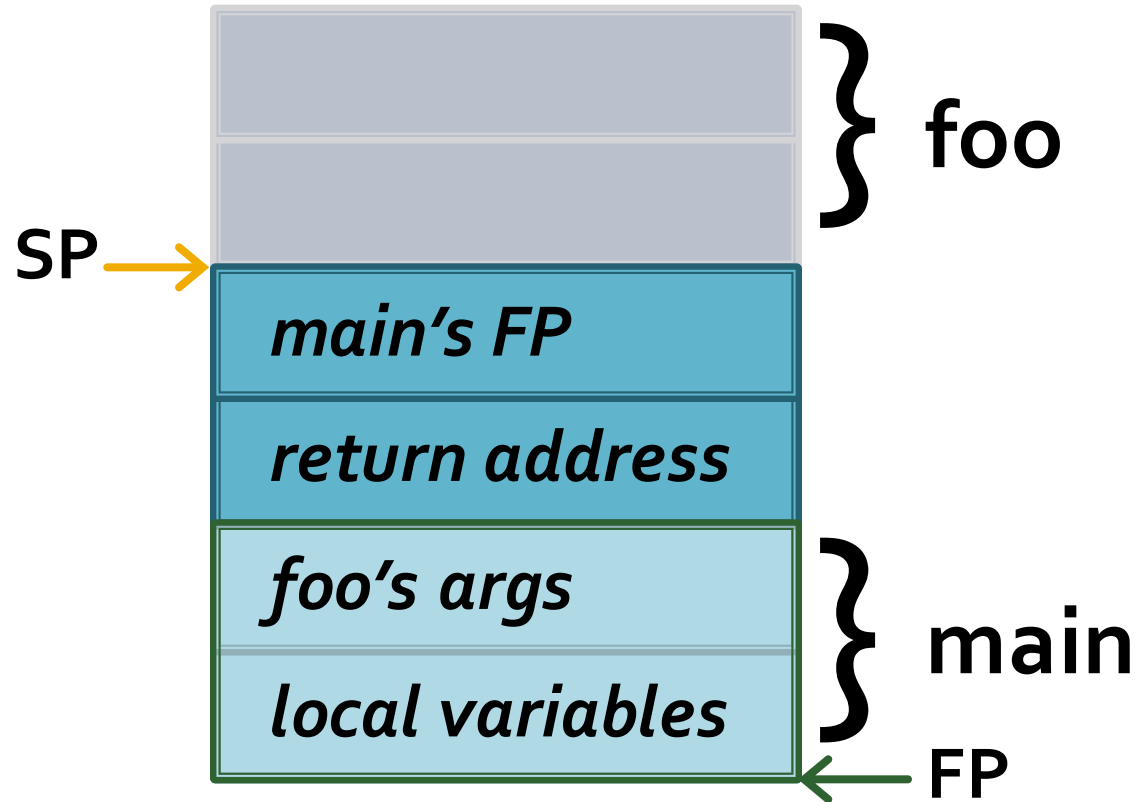
Stack frames



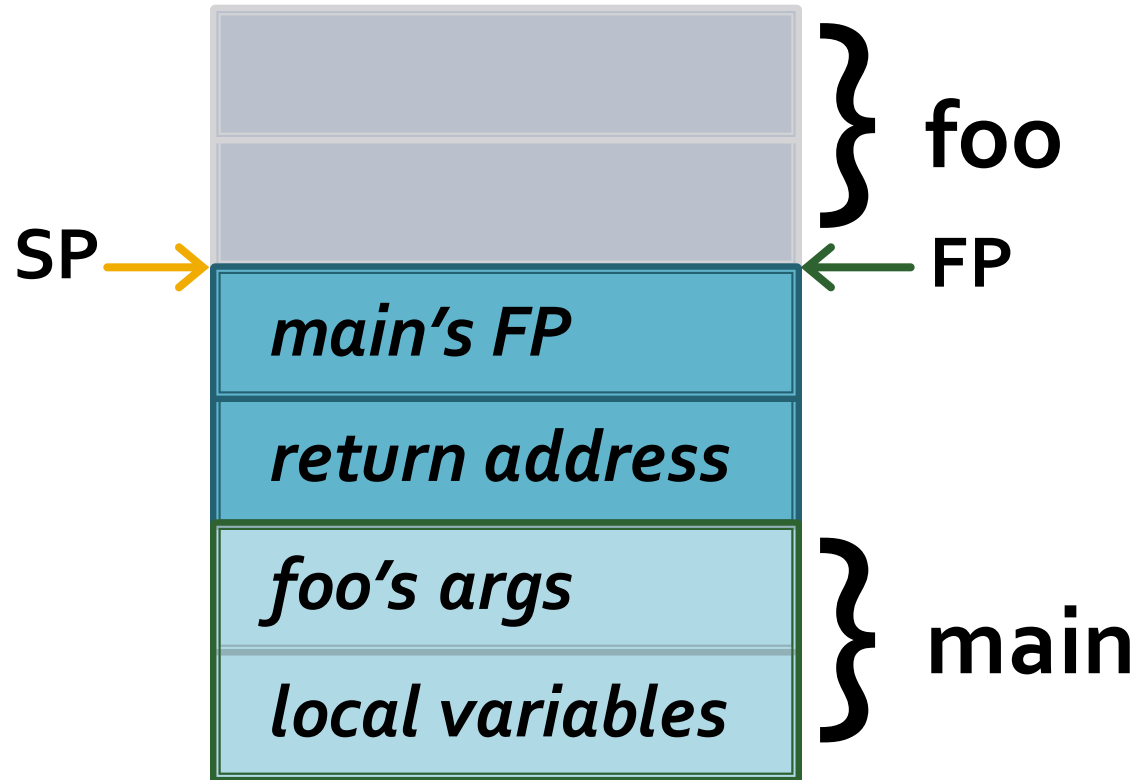
Stack frames



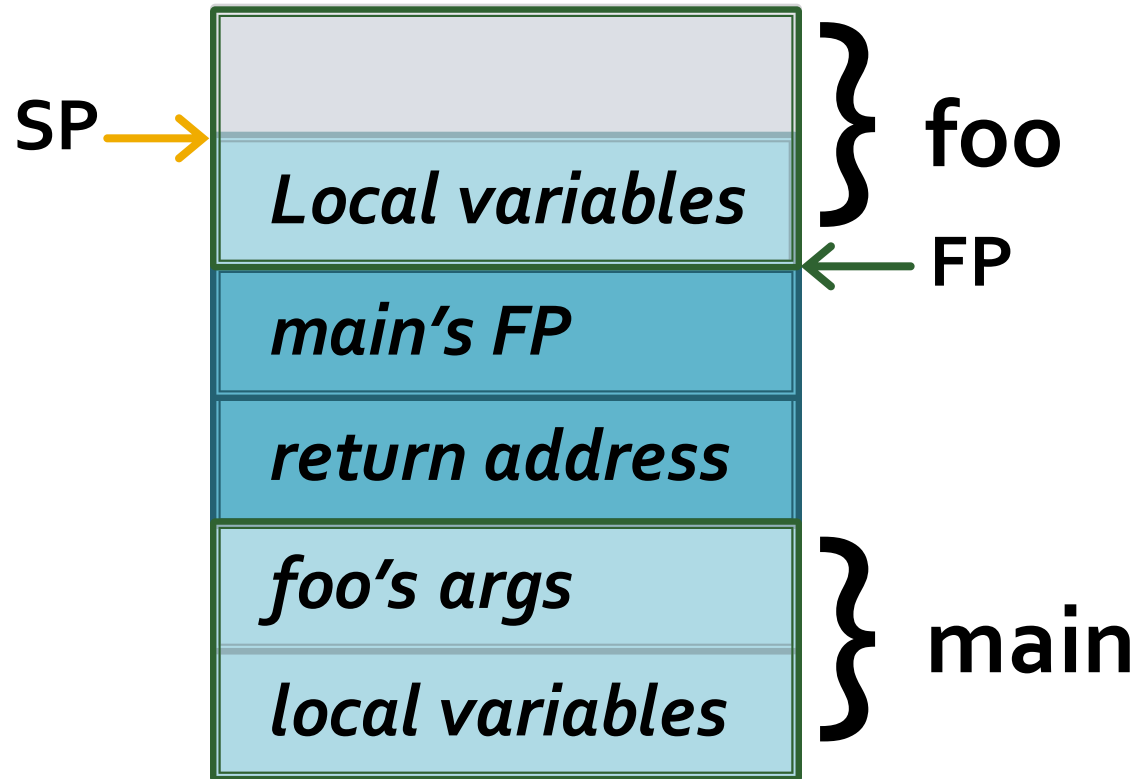
Stack frames



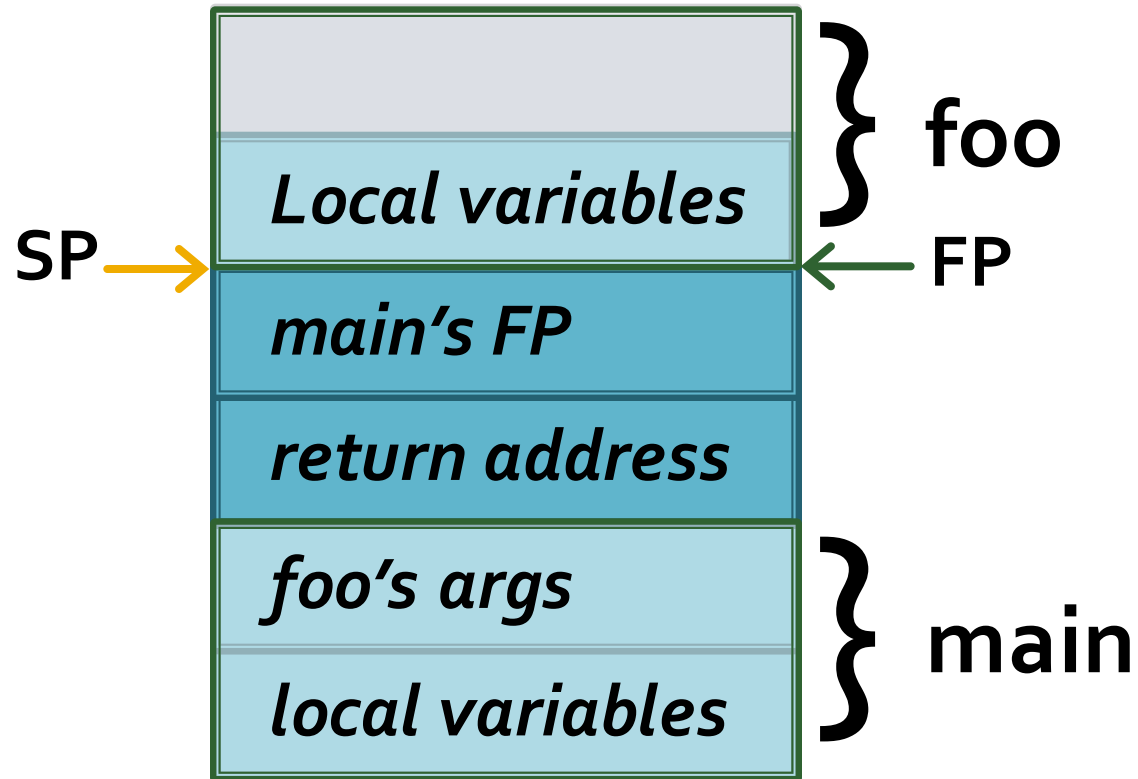
Stack frames



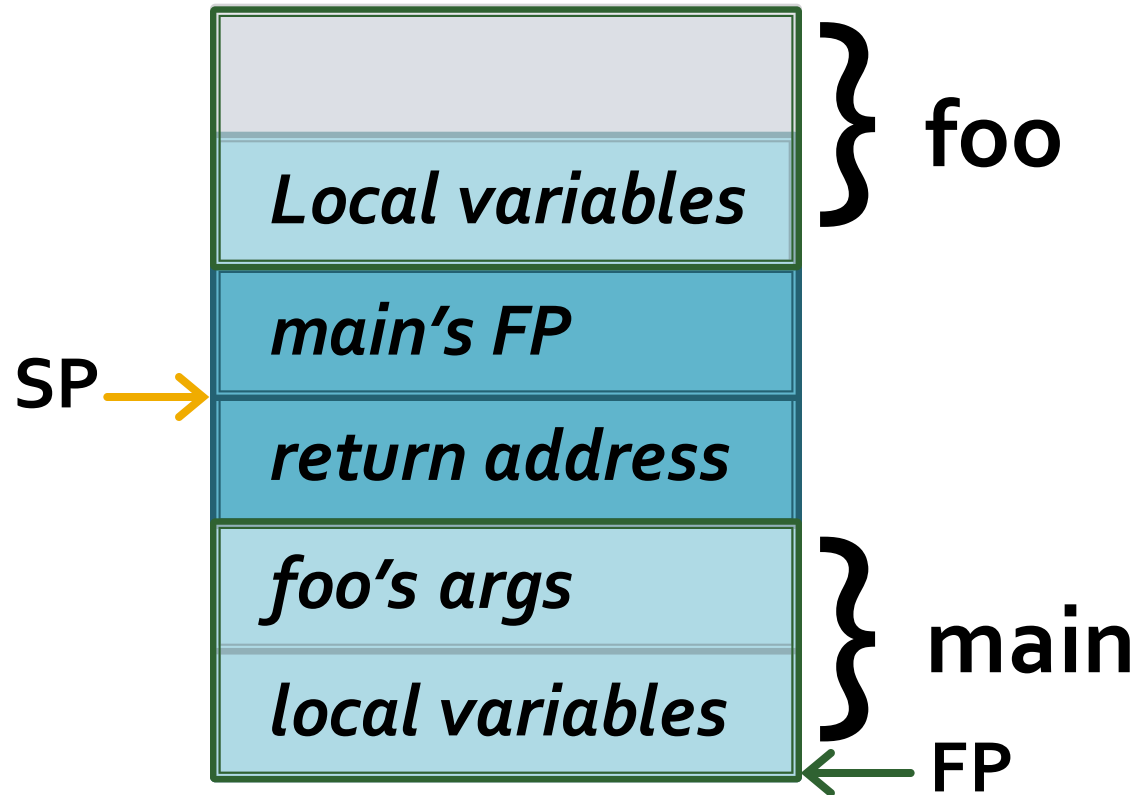
Stack frames



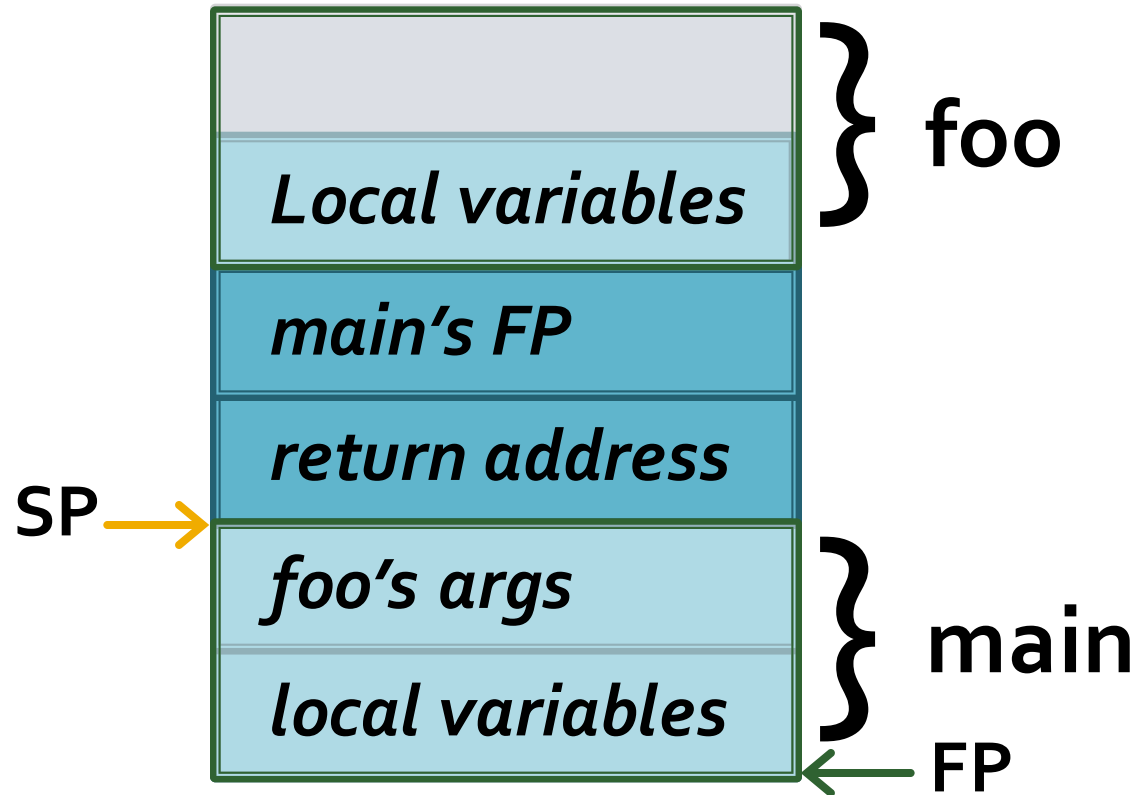
Stack frames



Stack frames



Stack frames



example.c

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

```
int main() {  
    foo(3, 6);  
}
```

example.s (x86)

```
void main() {  
    foo(3,6);  
}
```

main:

```
    push    ebp  
    mov     ebp, esp  
    sub     esp, 8  
    mov     esp+4, 6  
    mov     esp, 3  
    call    foo  
    leave  
    ret
```

```
void foo() {  
    char buf1[16];  
}
```

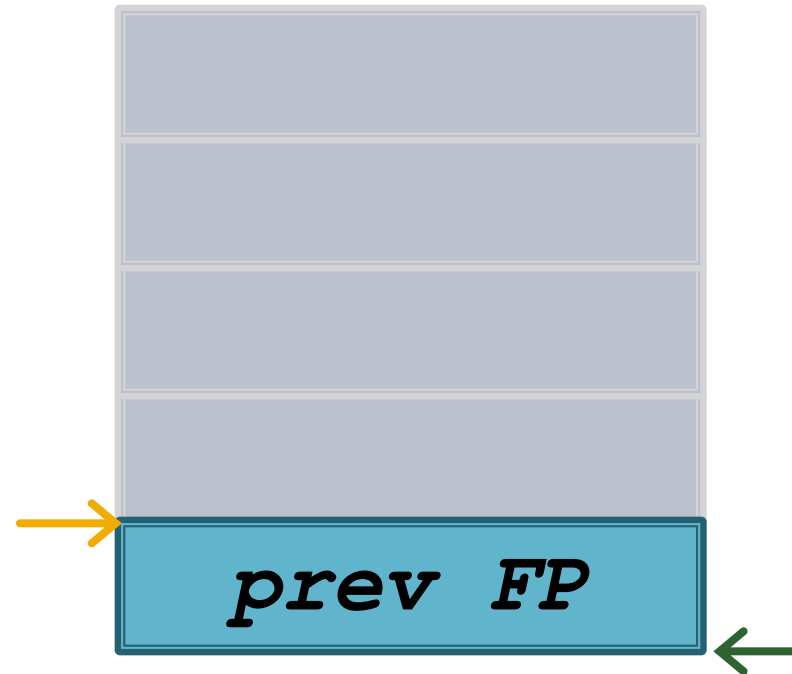
foo:

```
    push    ebp  
    mov     ebp, esp  
    sub     esp, 0x10  
    leave  
    ret
```

example.s (x86)

main:

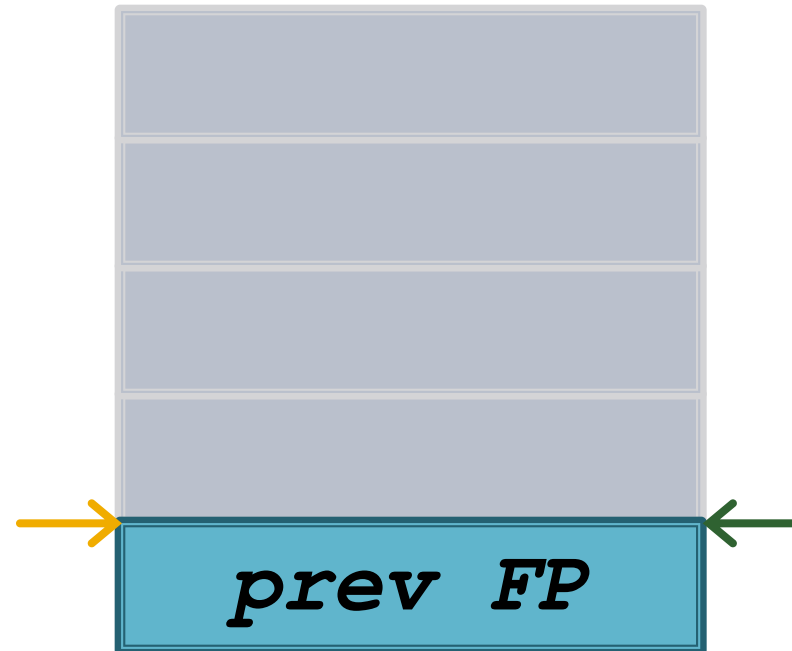
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

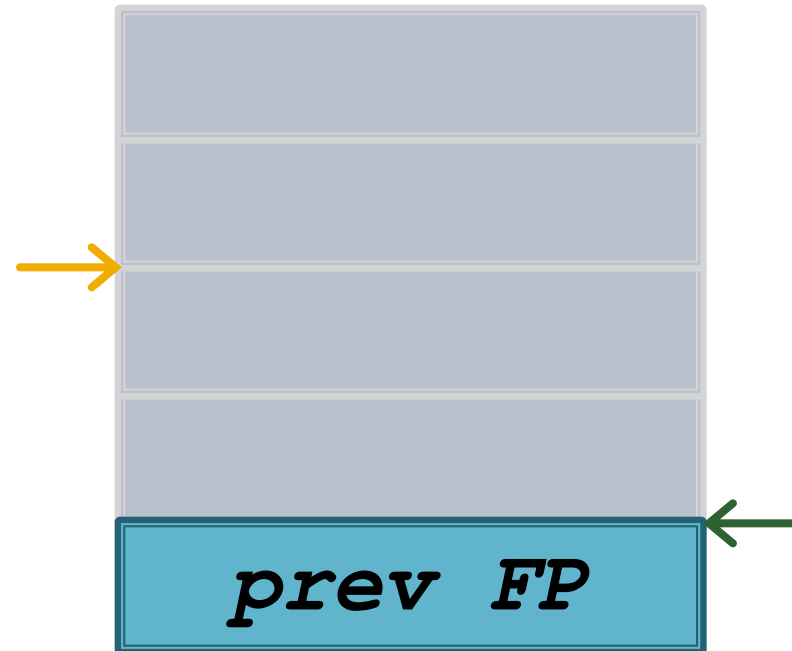
```
push    ebp
mov    ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

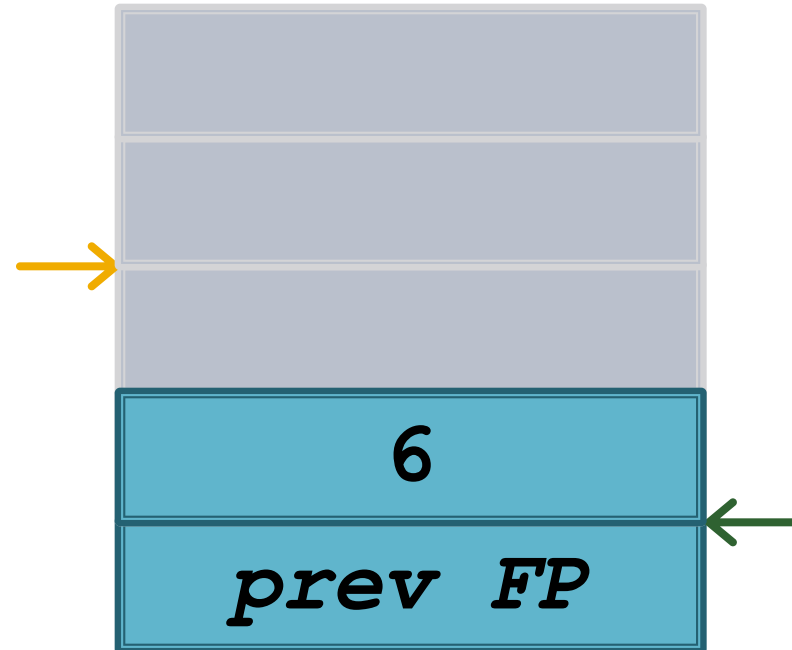
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

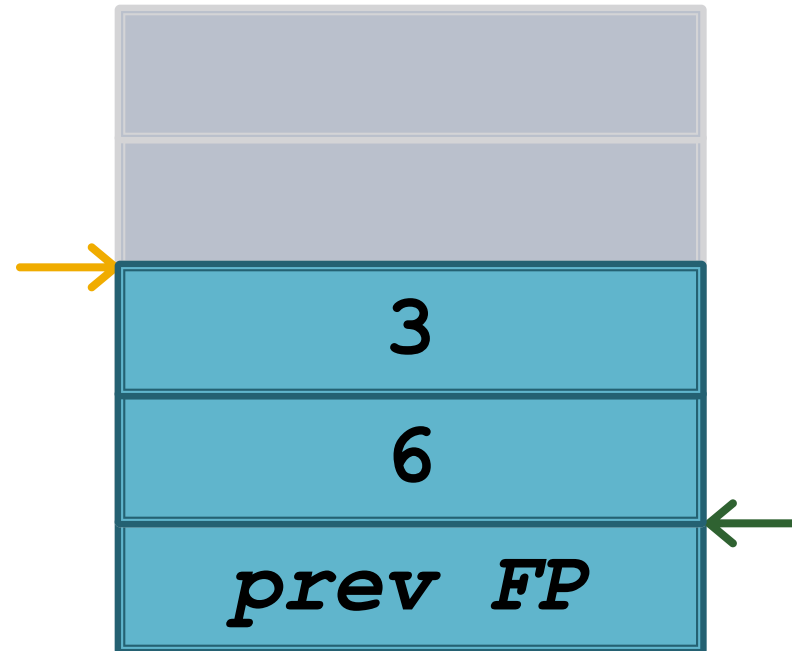
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

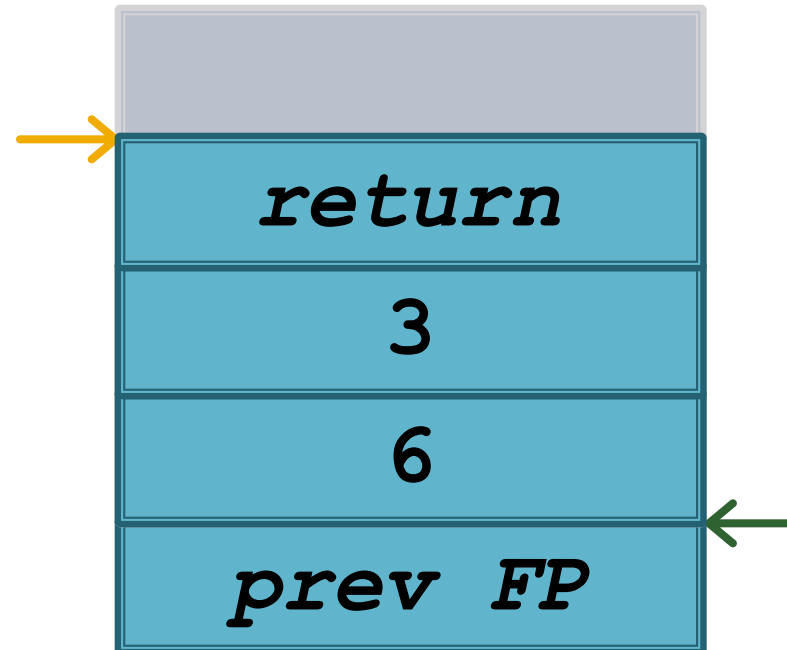
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

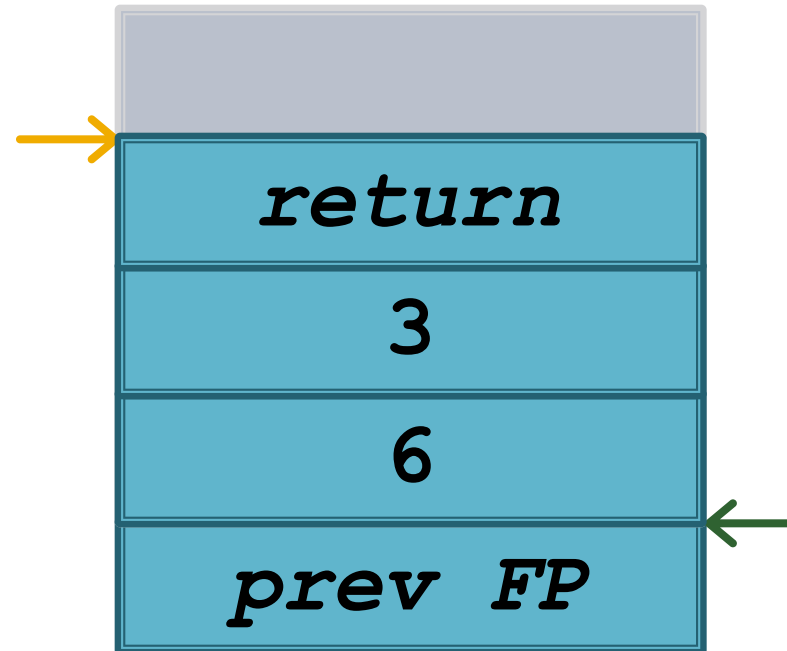
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call   foo
leave
ret
```



example.s (x86)

foo:

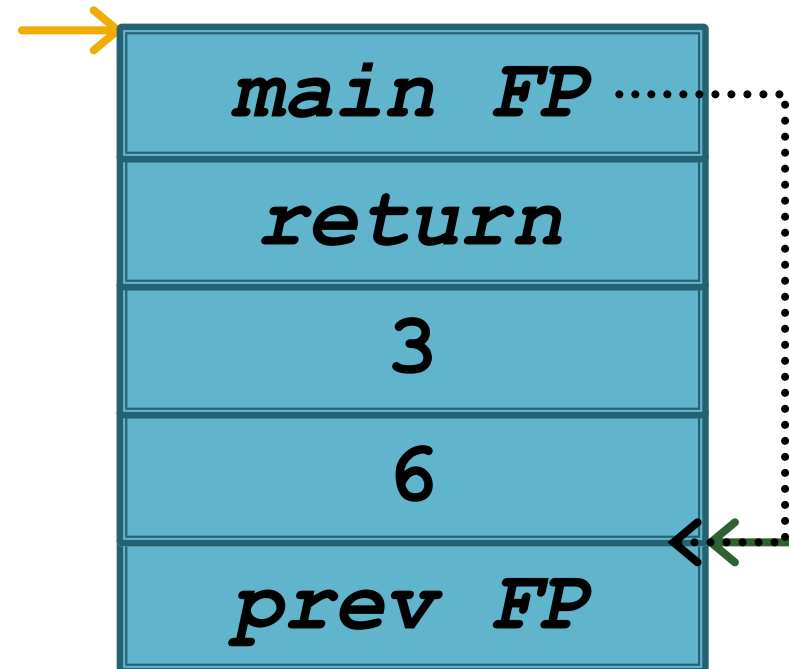
```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret
```



example.s (x86)

foo:

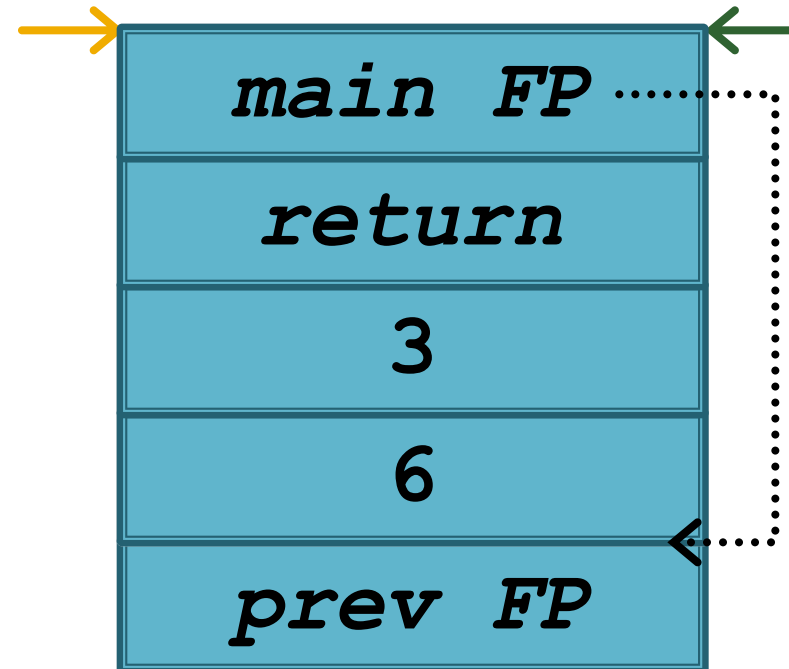
```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret
```



example.s (x86)

foo:

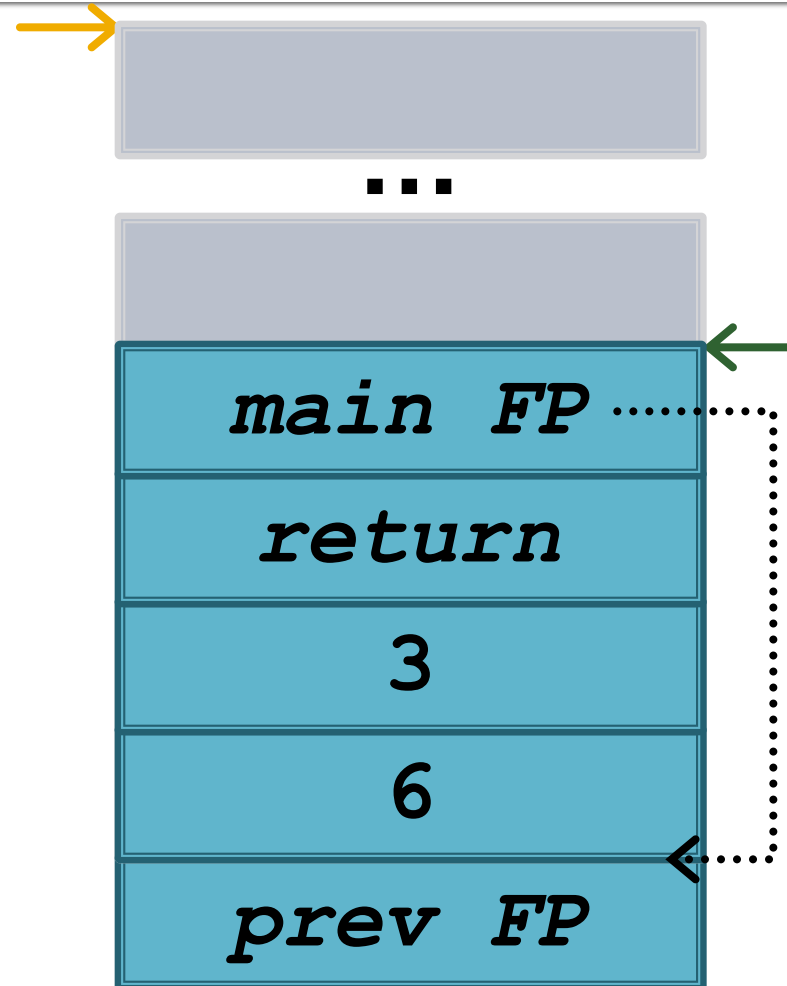
```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret
```



example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret
```

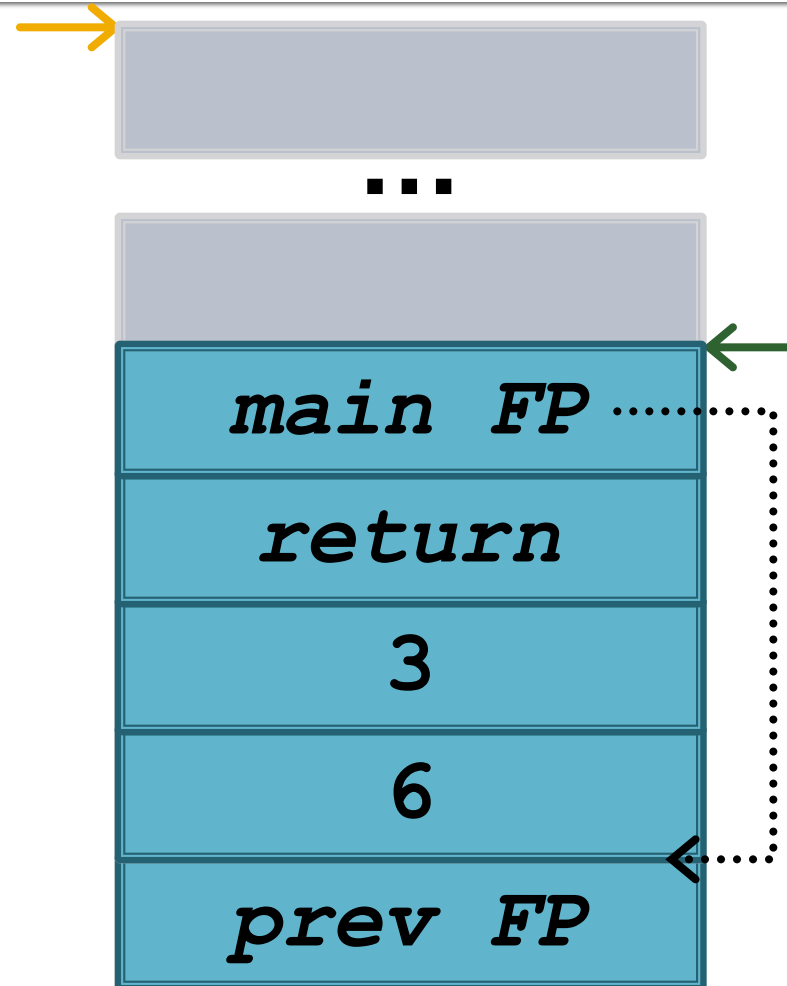


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave ← .....
ret
```

```
mov     esp, ebp
pop     ebp
```

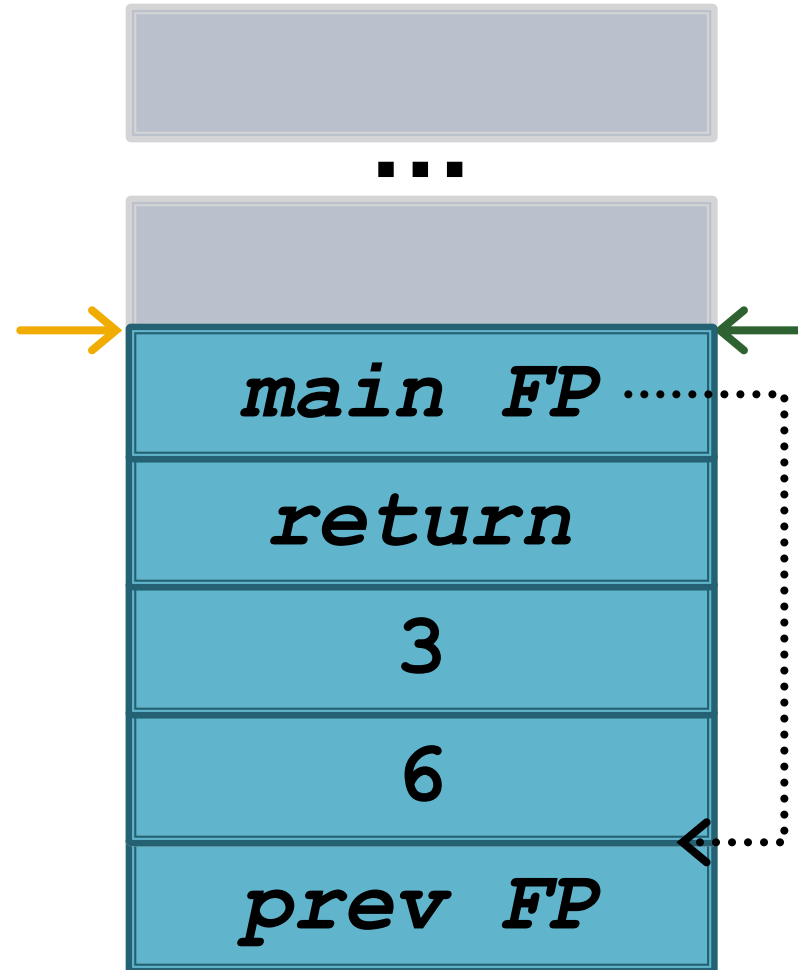


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave  ←
ret
```

```
mov     esp, ebp
pop     ebp
```

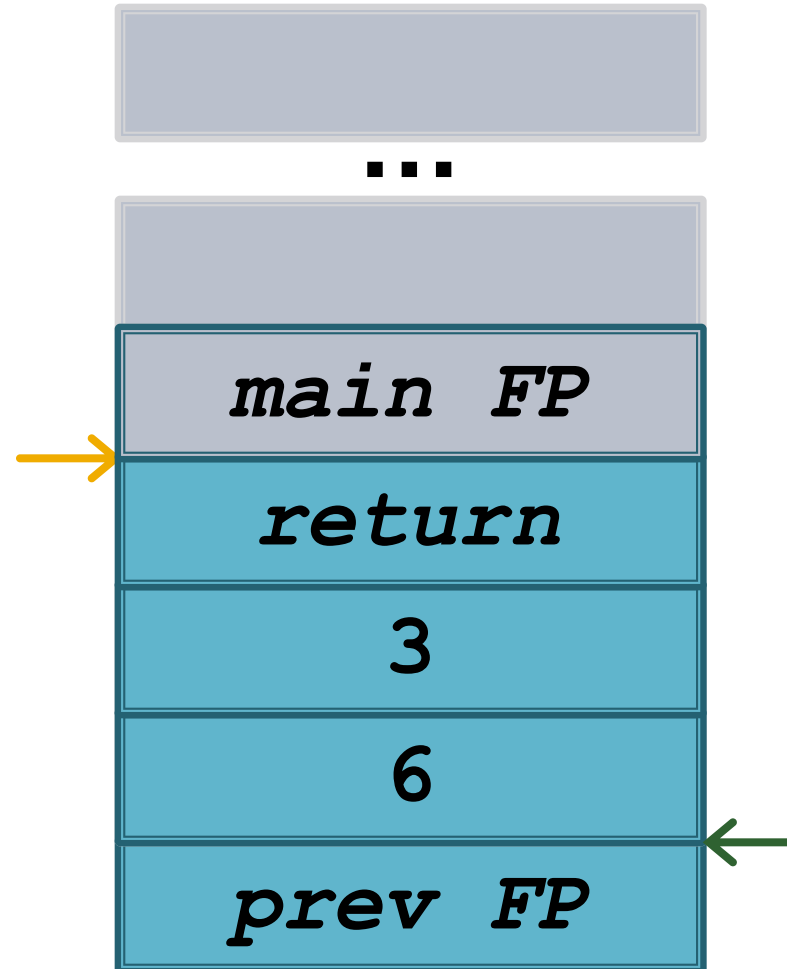


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave ← .....
ret
```

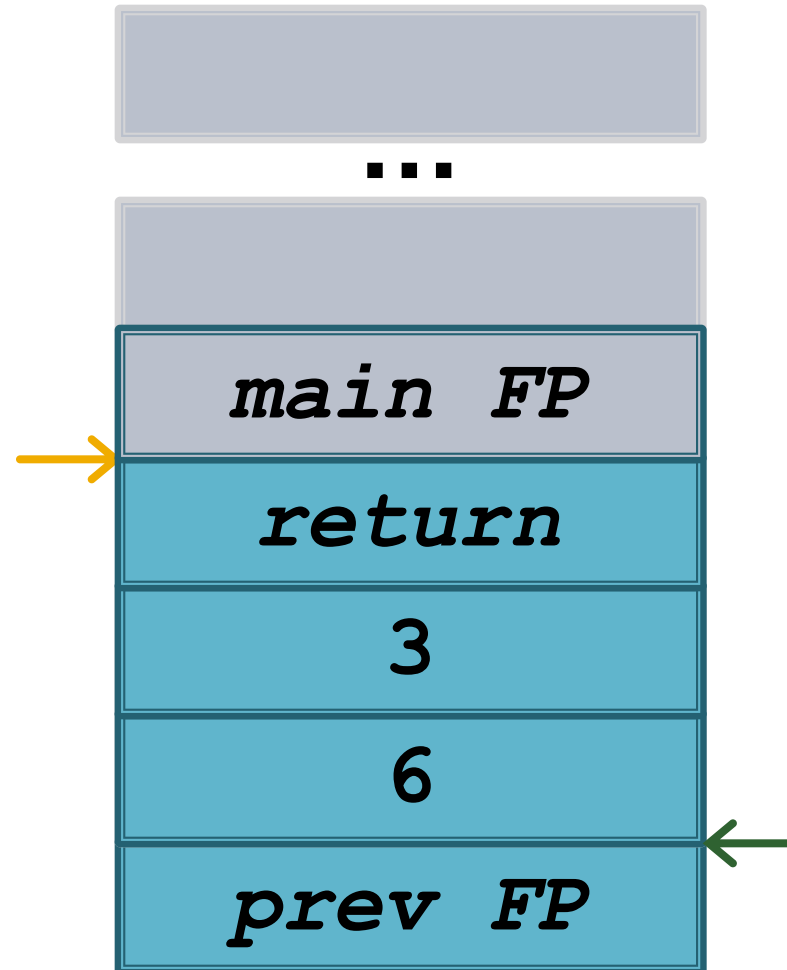
```
mov     esp, ebp
pop     ebp
```



example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret
```

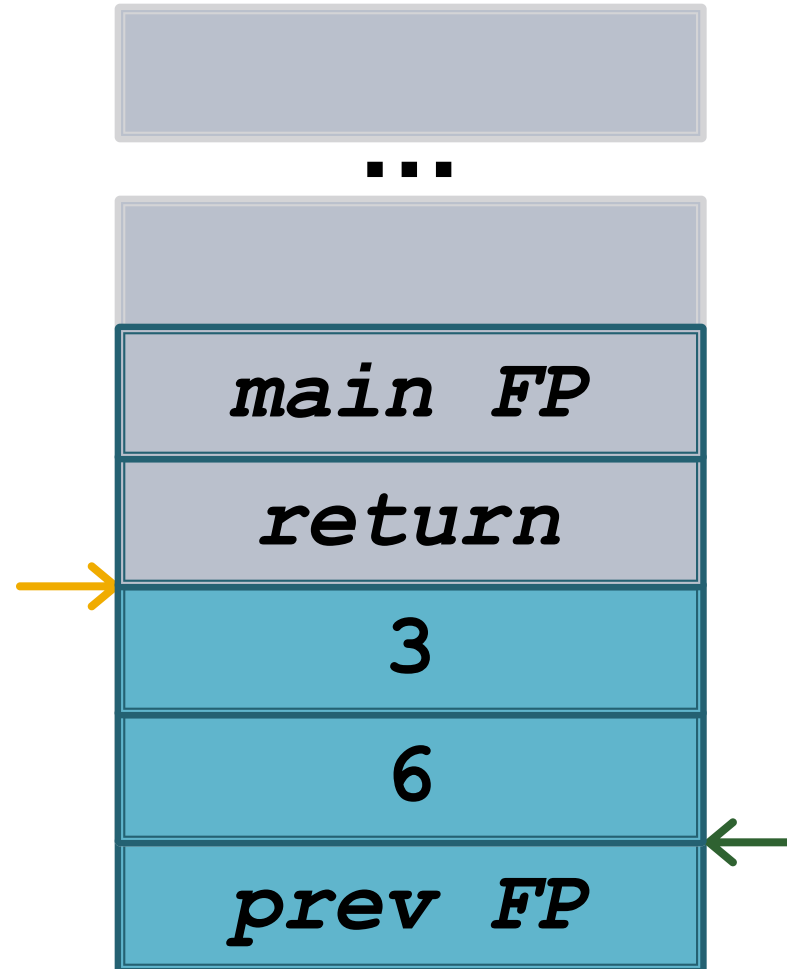


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 0x10
leave
ret ←
```

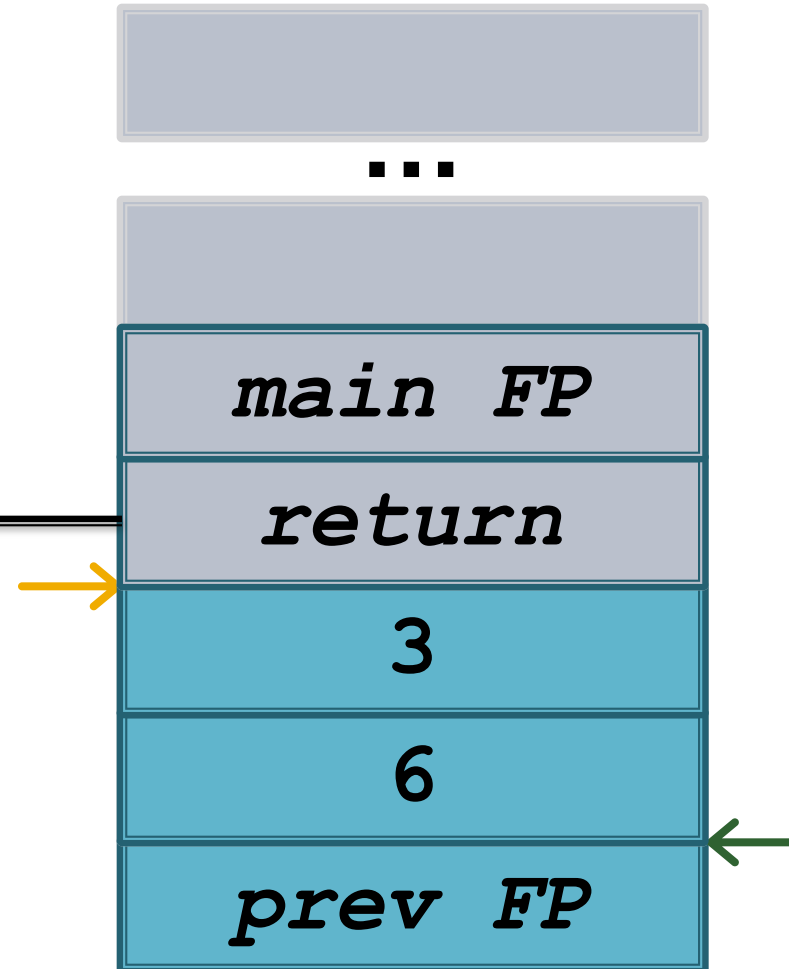
pop **eip**



example.s (x86)

main:

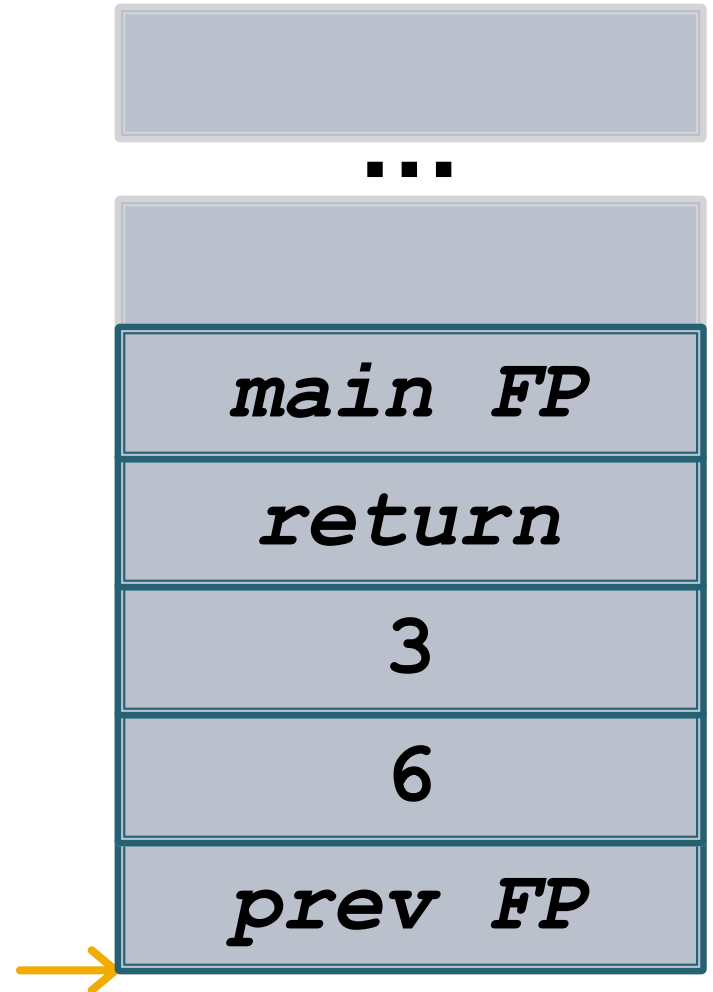
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave ←
ret
```



example.s (x86)

main:

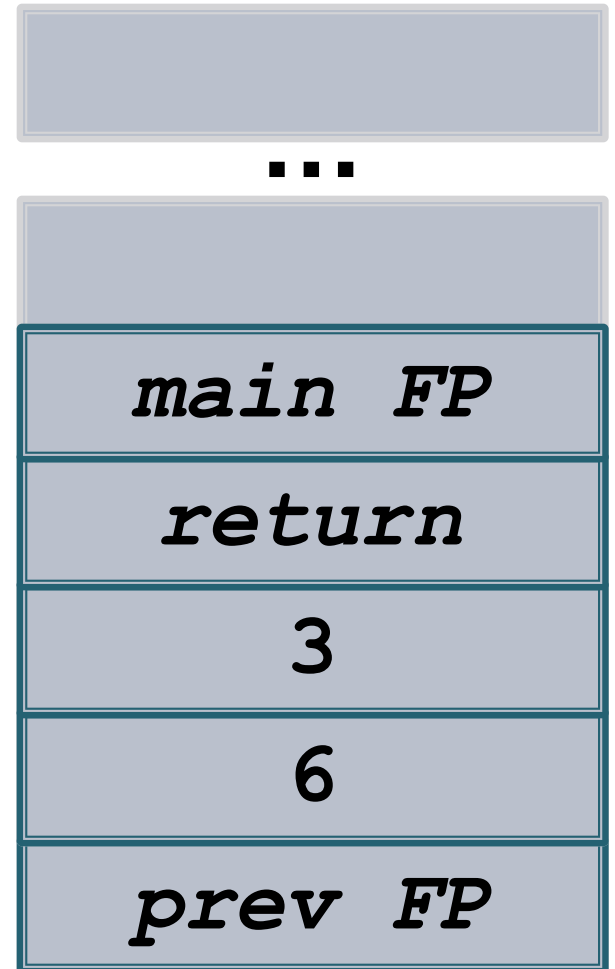
```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```



example.s (x86)

main:

```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     esp + 4, 6
mov     esp, 3
call    foo
leave
ret
```




Buffer overflow example

```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}  
  
int main() {  
    char str = "1234567890AB";  
    foo(str);  
}
```

Buffer overflow example

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


```
int main() {  
    char str = "1234567890AB";  
    foo(str);  
}
```



12 Bytes


Buffer overflow example

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

A yellow arrow points from the text "4 Bytes" to the number "4" in the array declaration "char buffer[4];".

4 Bytes

```
int main() {  
    char str = "1234567890AB";  
    foo(str);  
}
```

A yellow bracket is placed under the string "1234567890AB".

12 Bytes

example.s (x86)

```
int main() {  
    char str = "123456789012";  
    foo(str);  
}
```

main:

```
    push    ebp  
    mov     ebp, esp  
    push    str_ptr  
    call    foo  
    leave  
    ret
```

Text:

```
str_ptr: "1234567890AB"
```

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

foo:

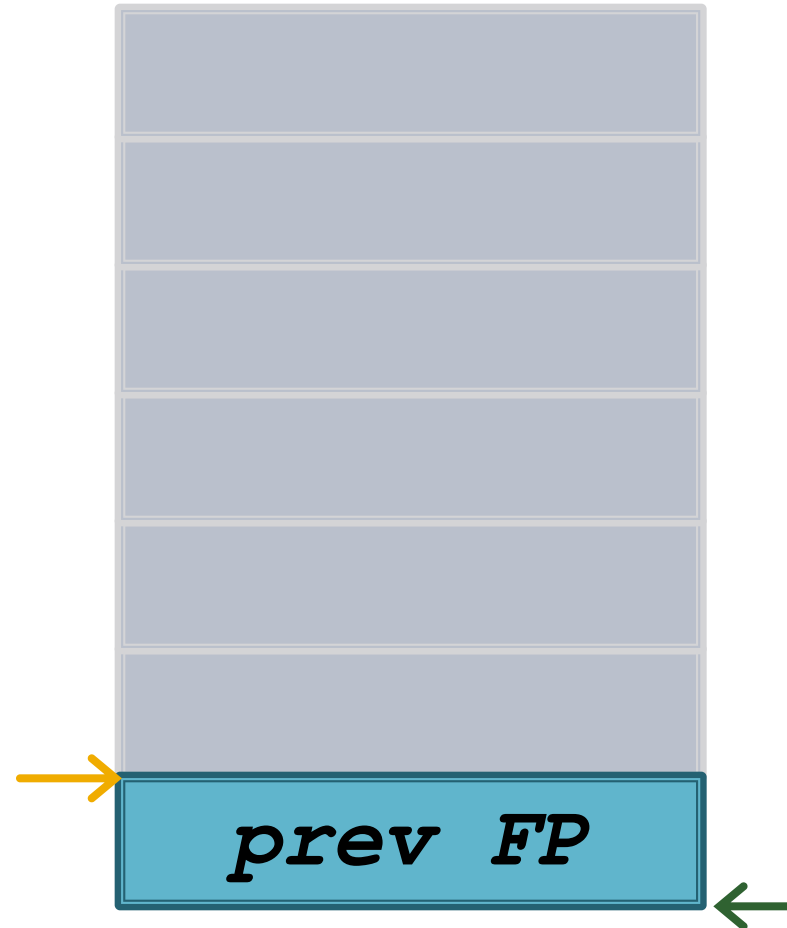
```
    push    ebp  
    mov     ebp, esp  
    sub     esp, 4  
    push    [ebp + 8]  
    push    ebp - 4  
    call    strcpy  
    leave  
    ret
```


example.s (x86)

main:

```
push    ebp
mov     ebp, esp
push    str_ptr
call    foo
leave
ret
```

str_ptr: "1234567890AB"

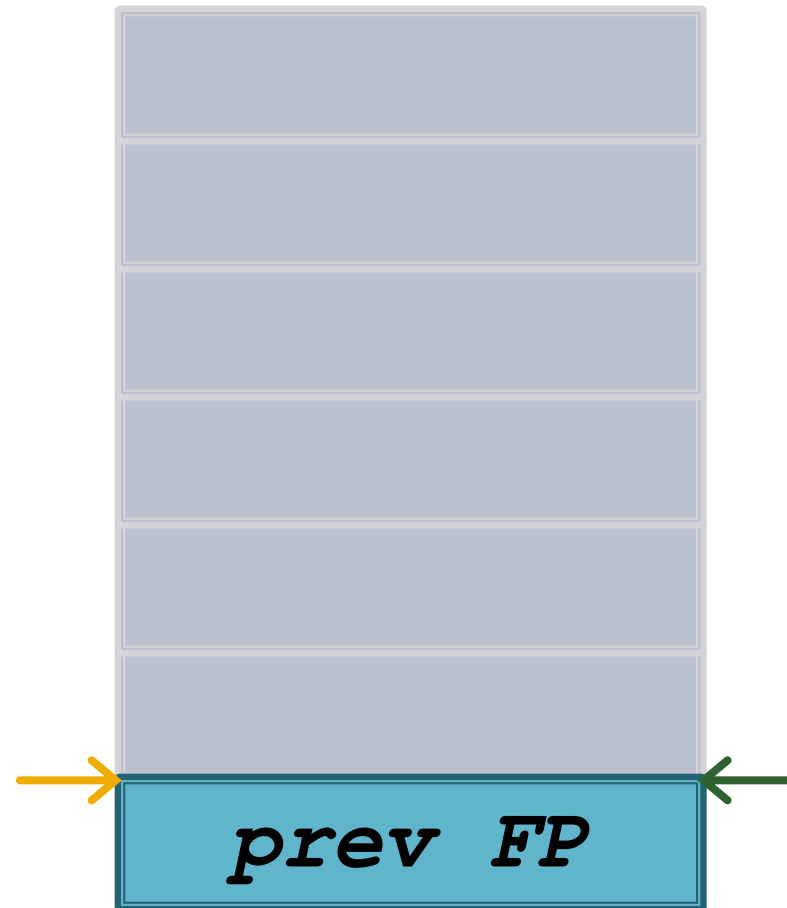


example.s (x86)

main:

```
push    ebp
mov     ebp, esp
push    str_ptr
call    foo
leave
ret
```

str_ptr: "1234567890AB"

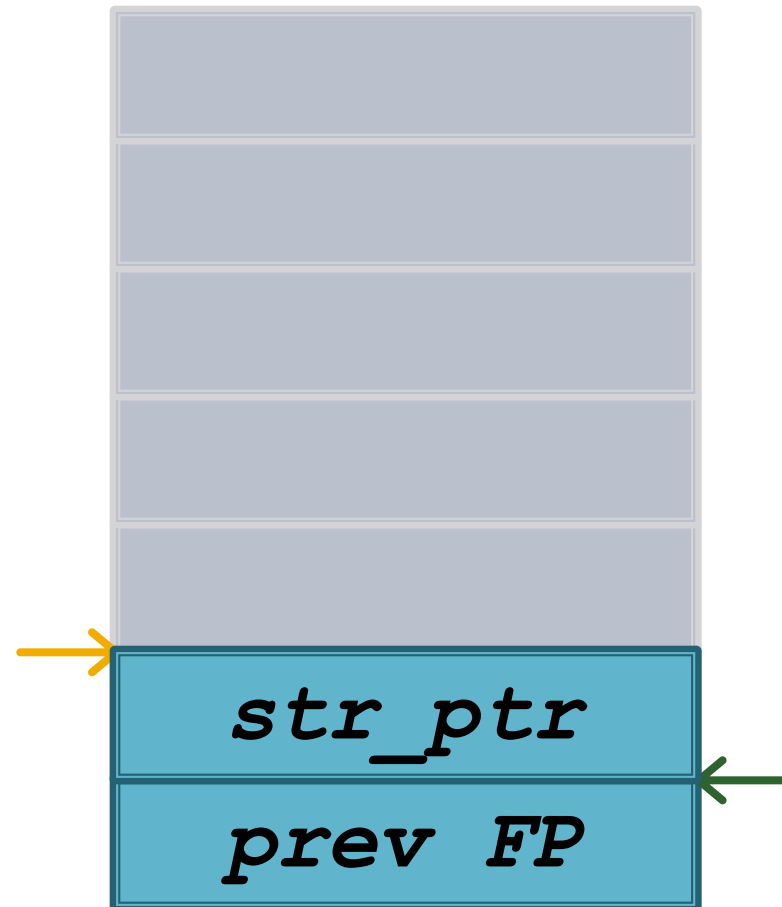


example.s (x86)

main:

```
push    ebp
mov     ebp, esp
push    str_ptr
call    foo
leave
ret
```

str_ptr: "1234567890AB"

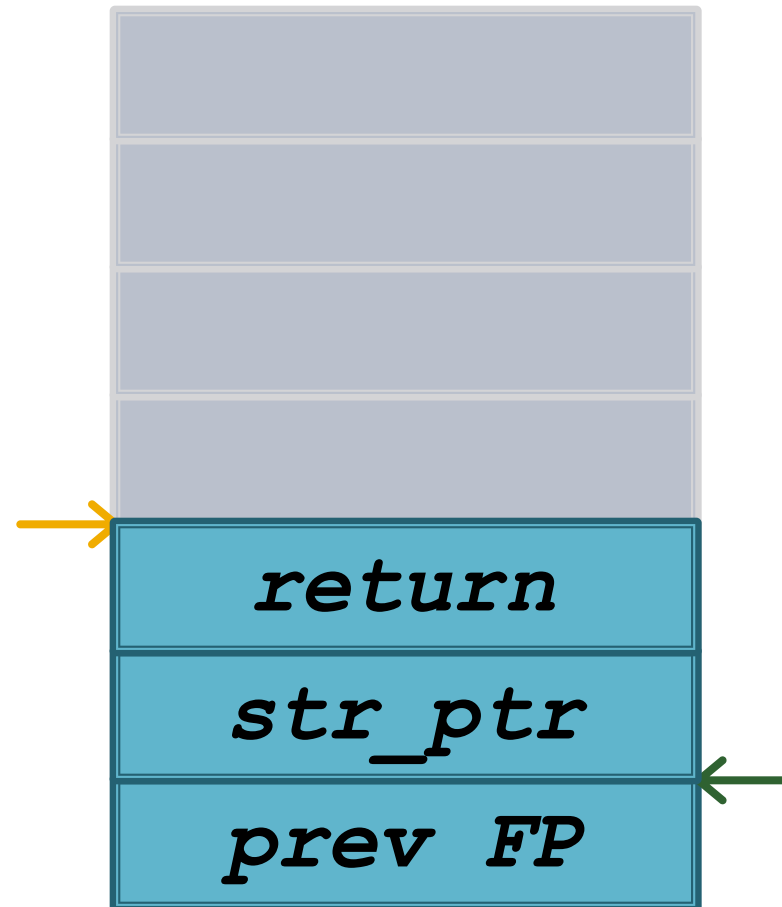


example.s (x86)

main:

```
push    ebp
mov     ebp, esp
push    str_ptr
call    foo
leave
ret
```

str_ptr: "1234567890AB"

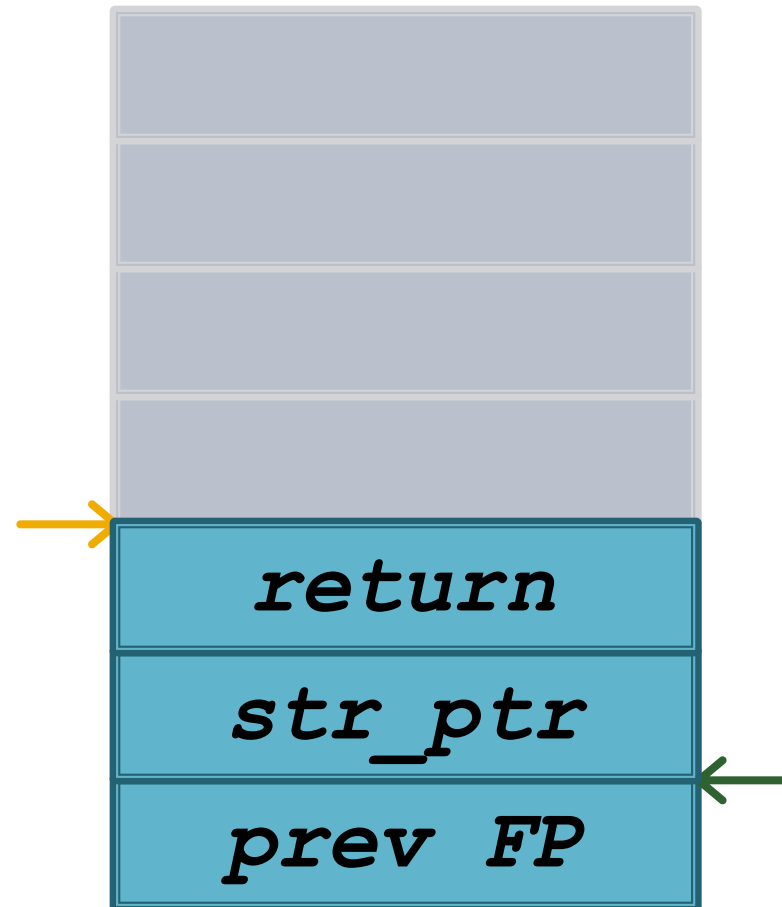


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

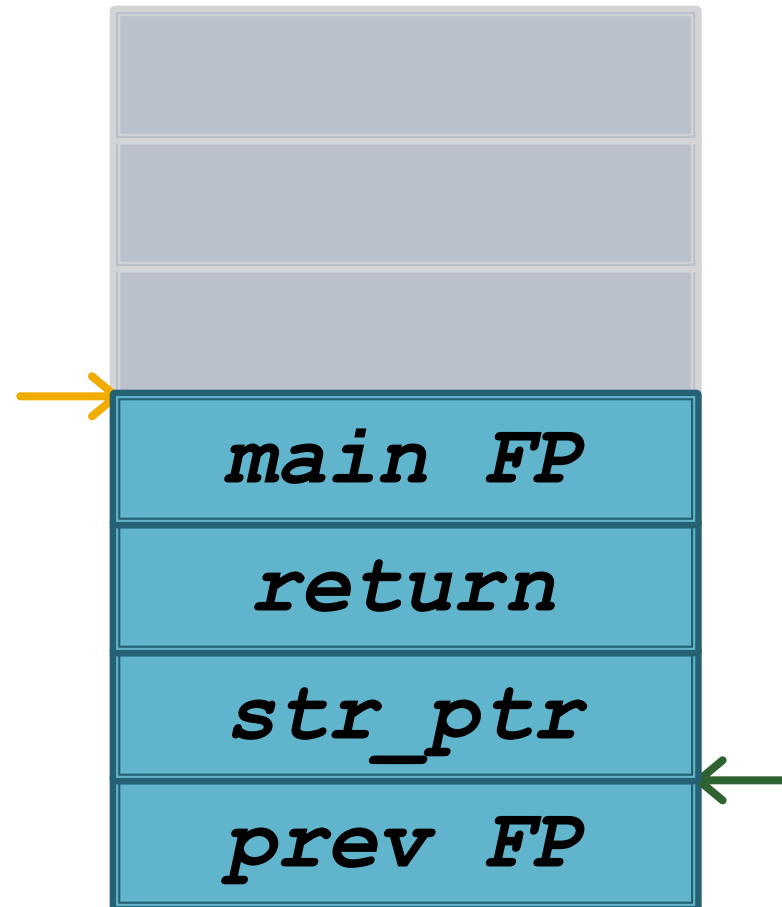


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

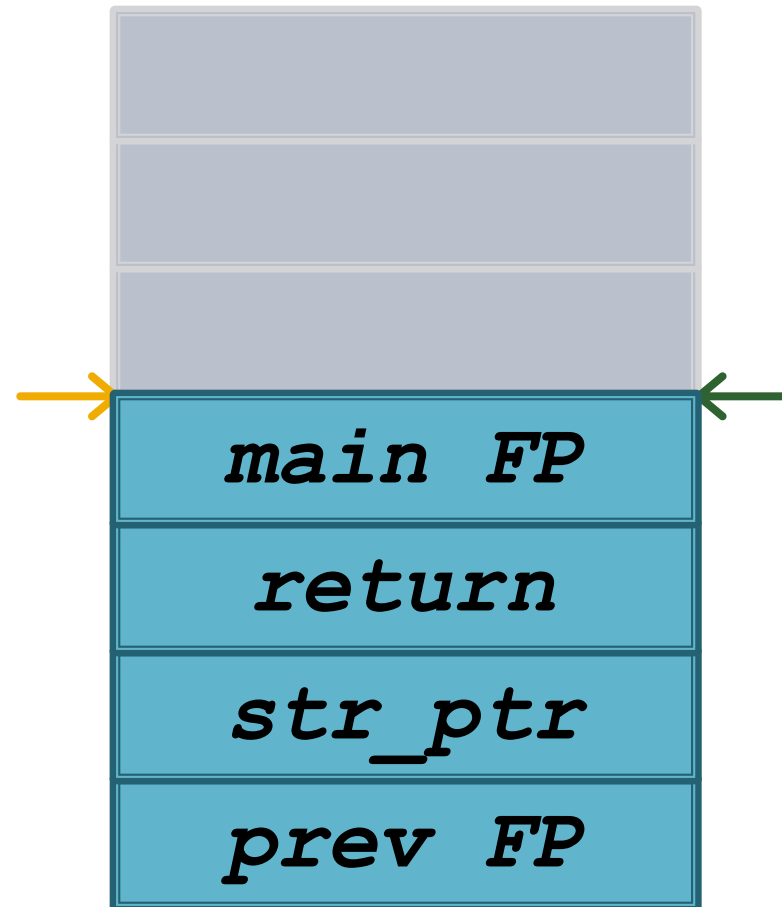


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

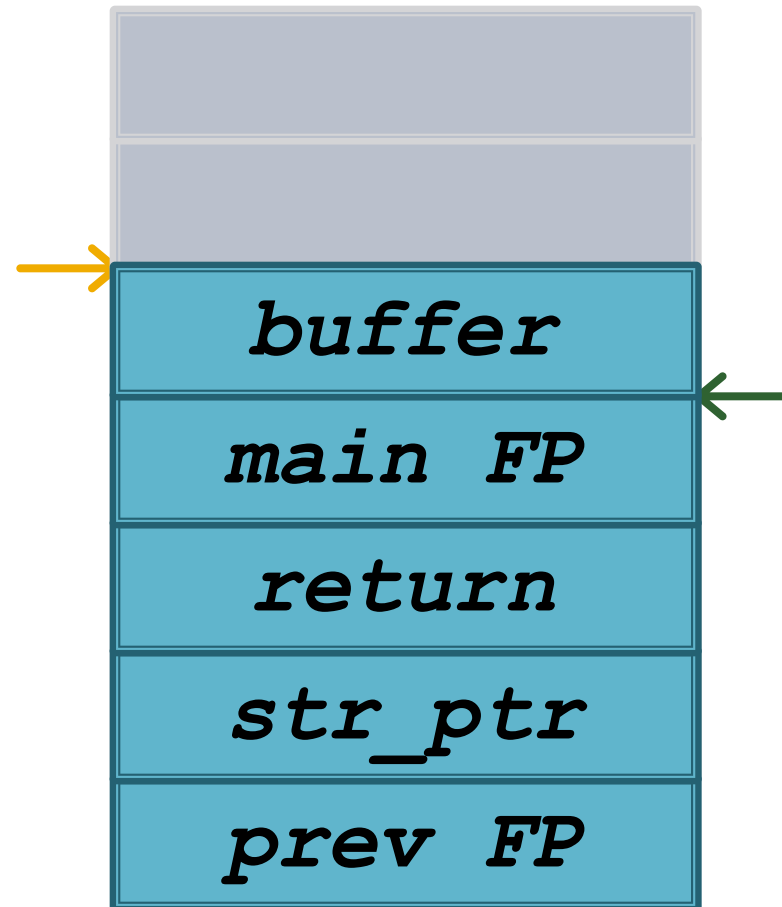


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

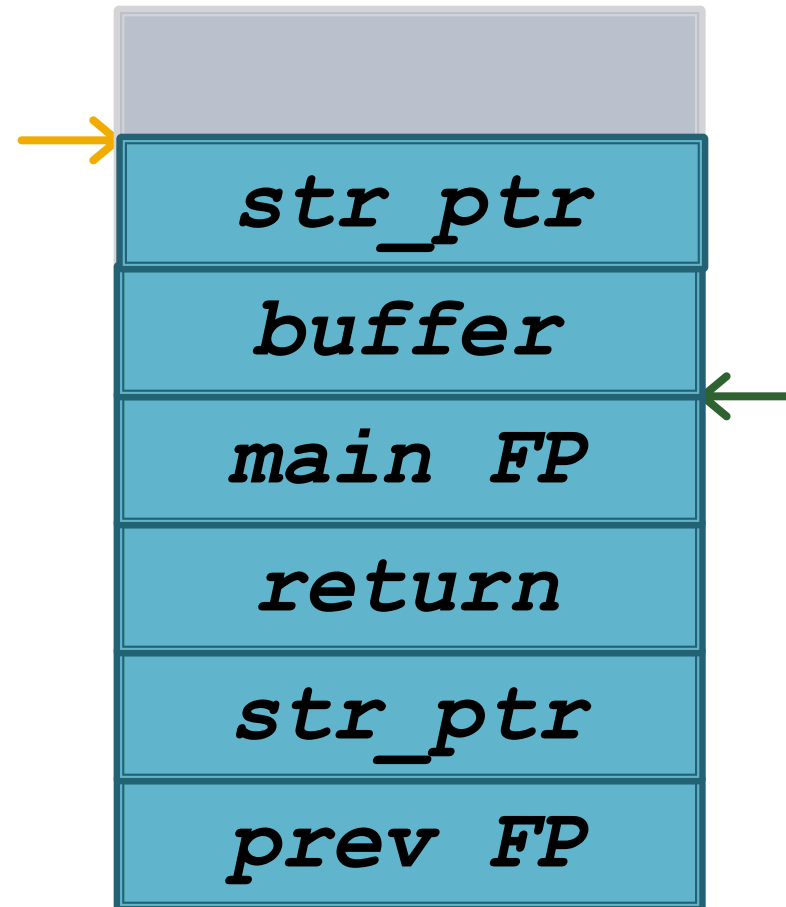


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

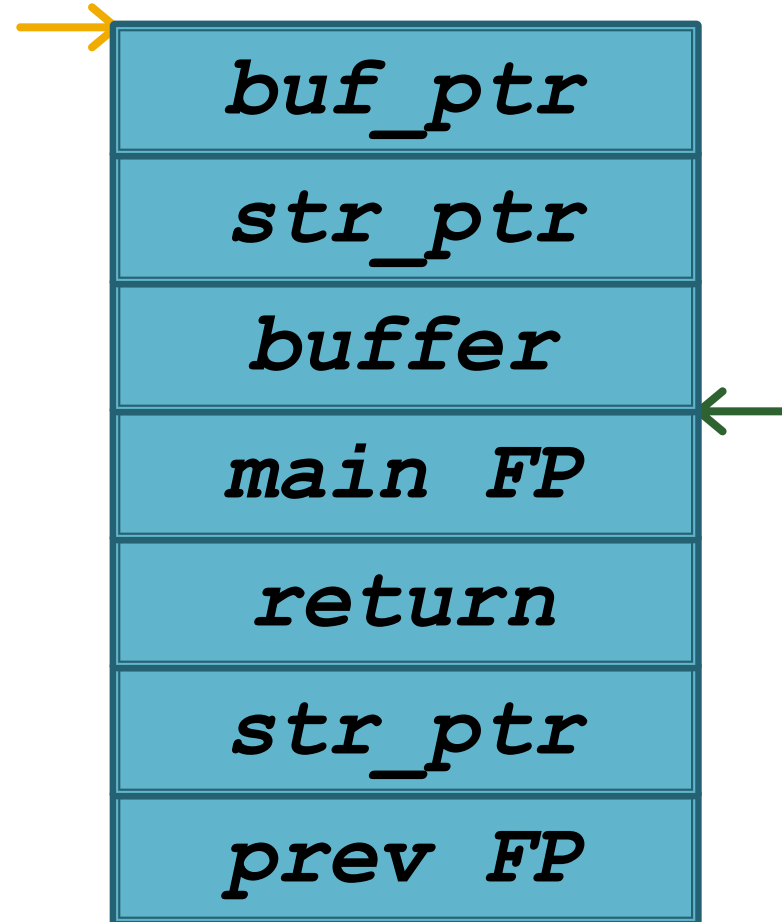


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

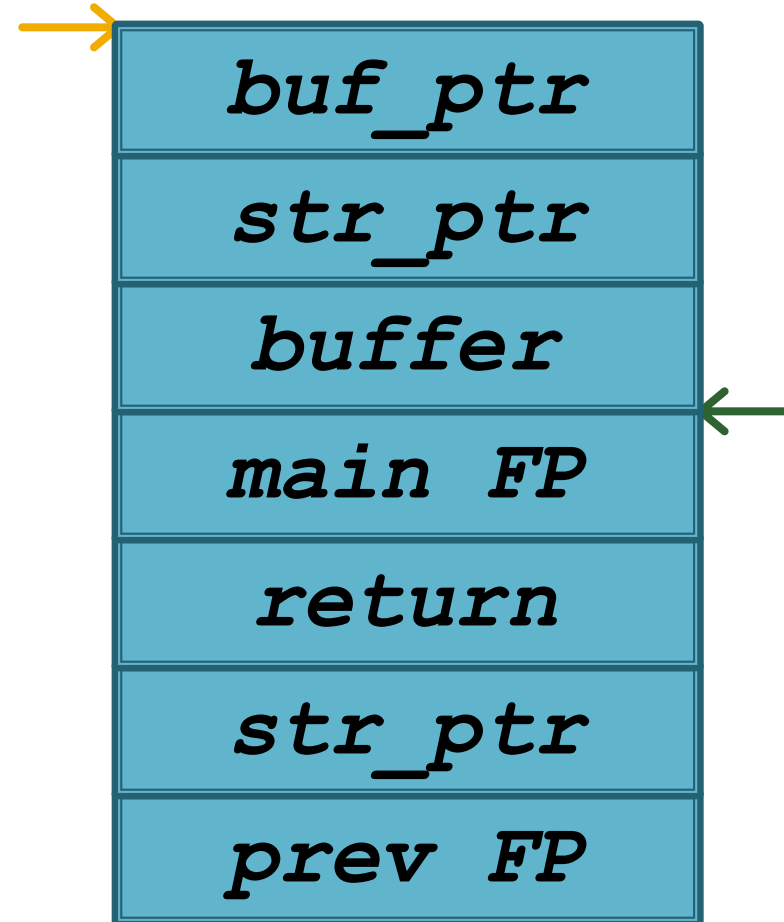


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call   strcpy
leave
ret
```

str_ptr: "1234567890AB"

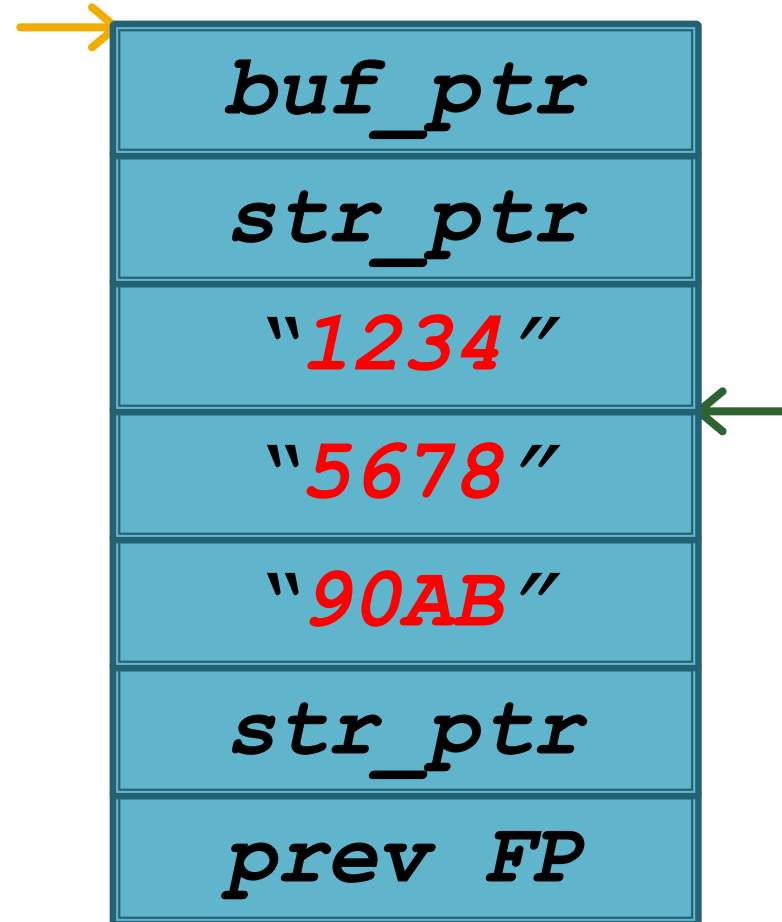


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"

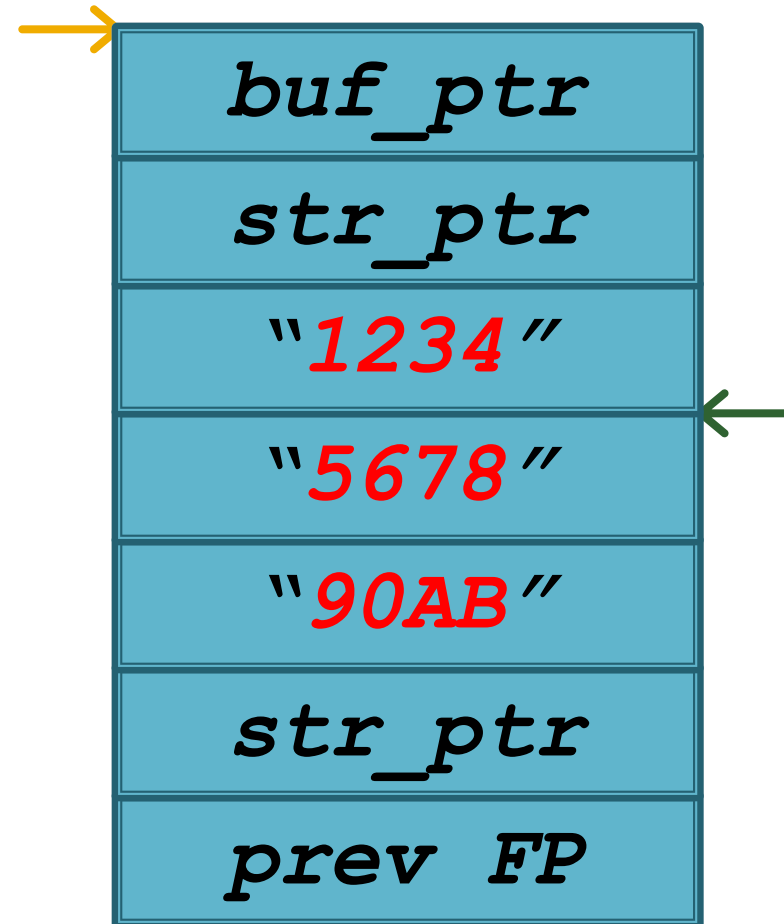


example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "1234567890AB"



example.s (x86)

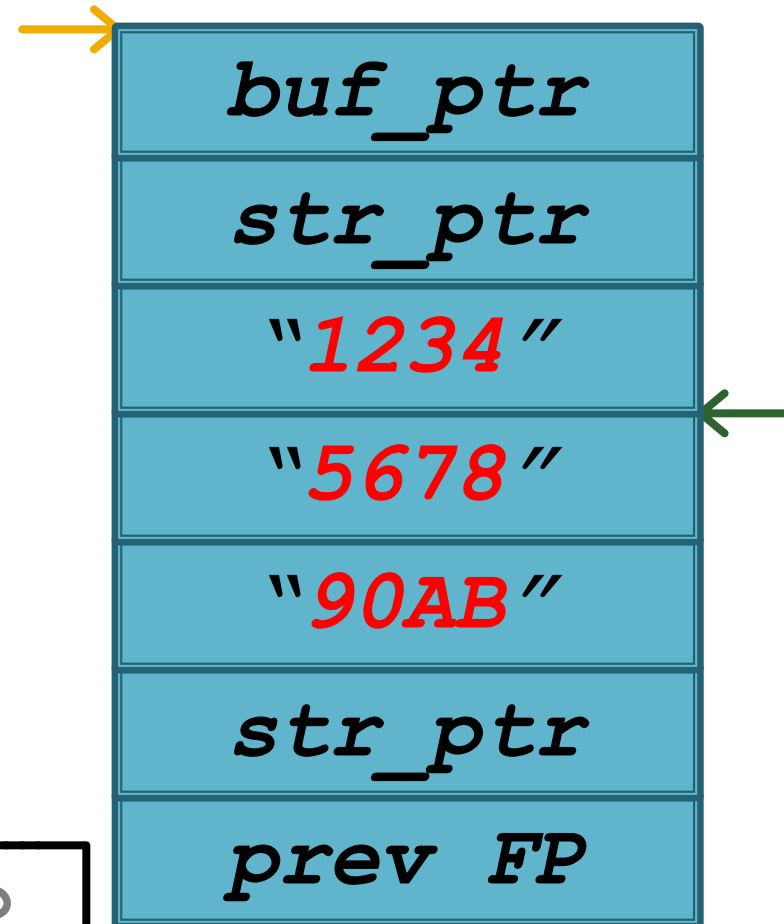
foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave   ←.....
```

```
ret
```

str_ptr:

```
mov     esp, ebp
pop     ebp
```



example.s (x86)

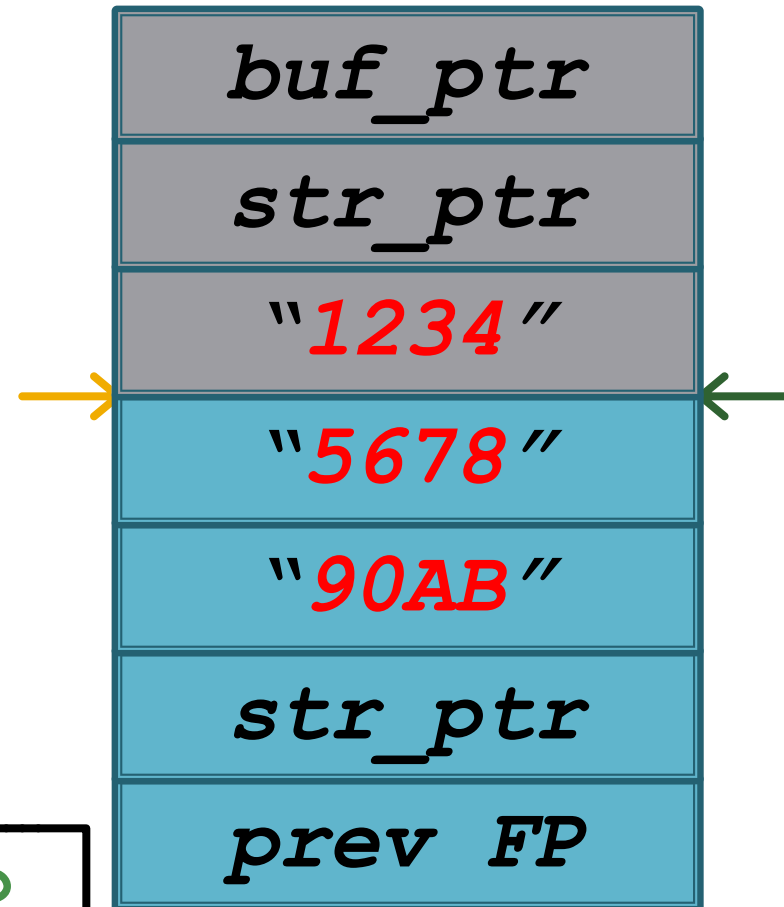
foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave   ←.....
```

```
ret
```

str_ptr:

```
mov     esp, ebp
pop     ebp
```



example.s (x86)

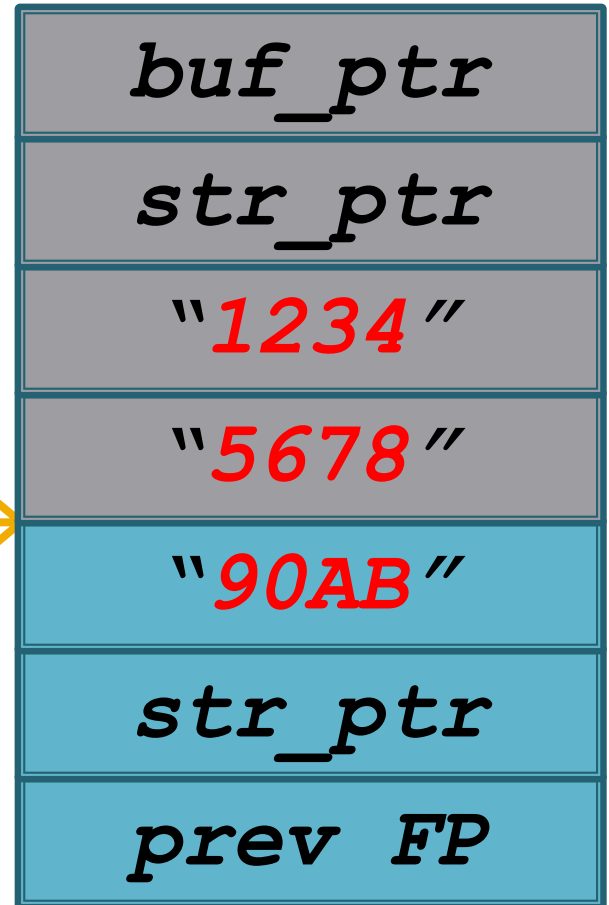
?? FP ← ?? == 0x35363738

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave   ←.....:
ret
```

str_ptr:

```
mov     esp, ebp
pop     ebp
```



example.s (x86)

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "123456789AB"

?? FP ← ?? == 0x35363738
?? EIP ?? == 0x39304142



example.s (x86)



example.s (x86)



This program has performed an illegal operation and will be shut down.

Close

If the problem persists, contact the program vendor.

Details>>

OPERA caused an invalid page fault in
module <unknown> at 0000:79e82379.

Registers:

EAX=79e82379 CS=015f EIP=79e82379 EFLGS=00000202

EBX=679e0000 SS=0167 ESP=0065f878 EBP=0065f8ac

ECX=67f879a8 DS=0167 ESI=67f330ec FS=0eaf

EDX=00000003 ES=0167 EDI=00000000 GS=0000

Bytes at CS:EIP:

Buffer overflow example

```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}  
  
int main() {  
    char str = "1234567890AB";  
    foo(str);  
}
```

User Input Buffer Overflow

```
void welcome_user() {  
    char buffer[100];  
    printf("Enter name: ");  
    gets(buffer);  
    printf("Hello, %s!\n", buffer);  
}
```

User Input Buffer Overflow

```
void welcome_user() {  
    char buffer[100];  
    printf("Enter name: ");  
    gets(buffer);  
    printf("Hello, %s!\n", buffer);  
}
```

User Input Buffer Overflow

```
void welcome_user() {  
    char buffer[100];  
    printf("Enter name: ");  
    gets(buffer);  
    printf("Hello, %s!\n", buffer);  
}
```

```
python -c "print 'a' * 1024" | ./a.out
```

Network Input Buffer Overflow

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


Network Input Buffer Overflow

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

Network Input Buffer Overflow

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

```
python -c "print '\x00\x01\x00\x00' +\  
'a' * 65536" | nc <IP> <PORT>
```

Stack Shellcode

Let's do something more usefull than crashing

1. Compile your own code to be executed
2. Inject into the application
3. Jump to your binary instructions

Stack Shellcode

Let's do something more usefull than crashing

1. Compile your own code to be executed
2. Inject into the application
3. Jump to your binary instructions

```
int main() {  
    goto_target:  
    goto goto_target;  
}
```

Stack Shellcode

```
int main() {  
    goto_target:  
    goto goto_target;  
}
```

00000000 <_main>:

0: 55

1: 89 e5

3: 50

4: c7 45 fc 00 00 00 00

b: e9 fb ff ff ff

push ebp

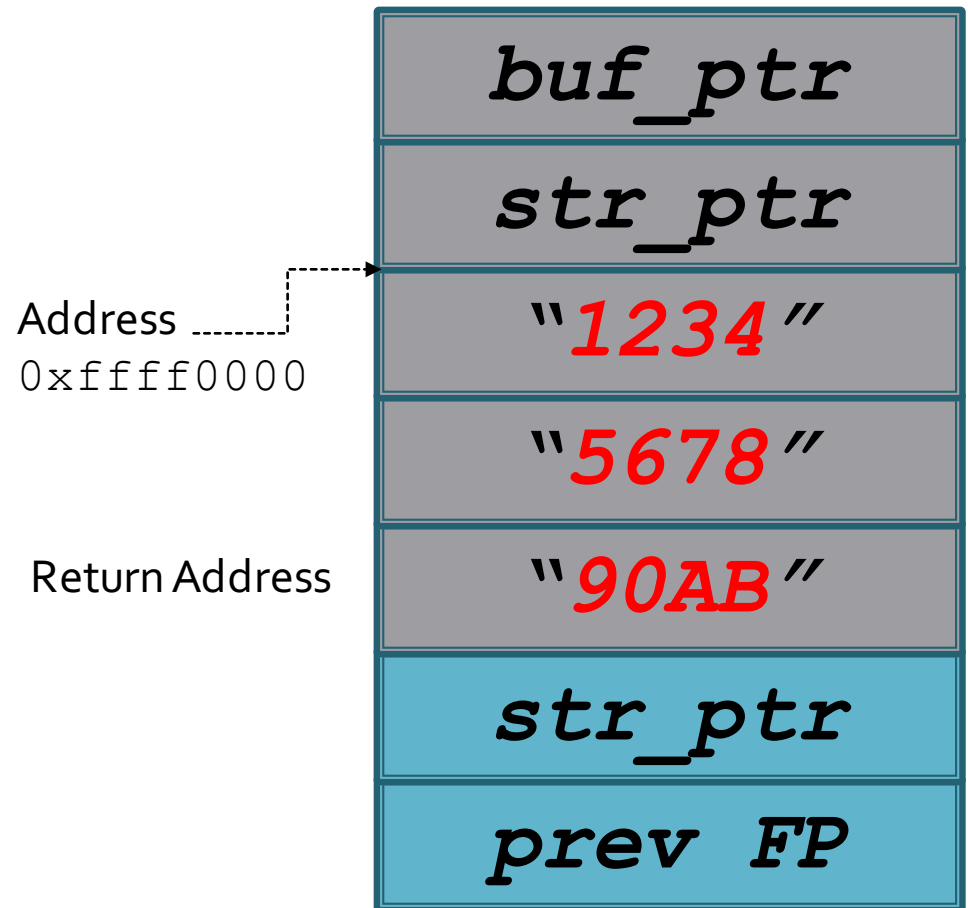
mov ebp, esp

push eax

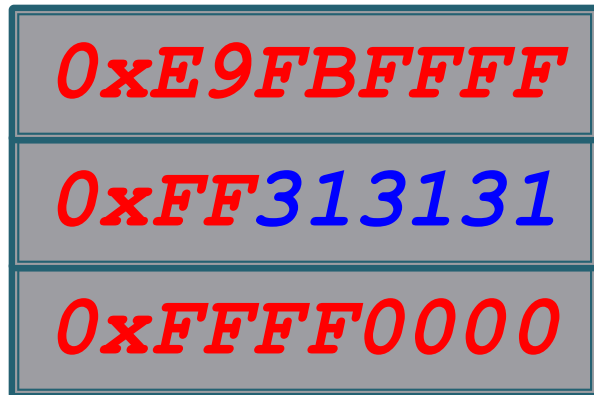
mov DWORD PTR [ebp-0x4], 0x0

jmp b <_main+0xb>

Stack Shellcode



Stack Shellcode



Address
0xffff0000

Return Address



b: e9 fb ff ff ff

jmp

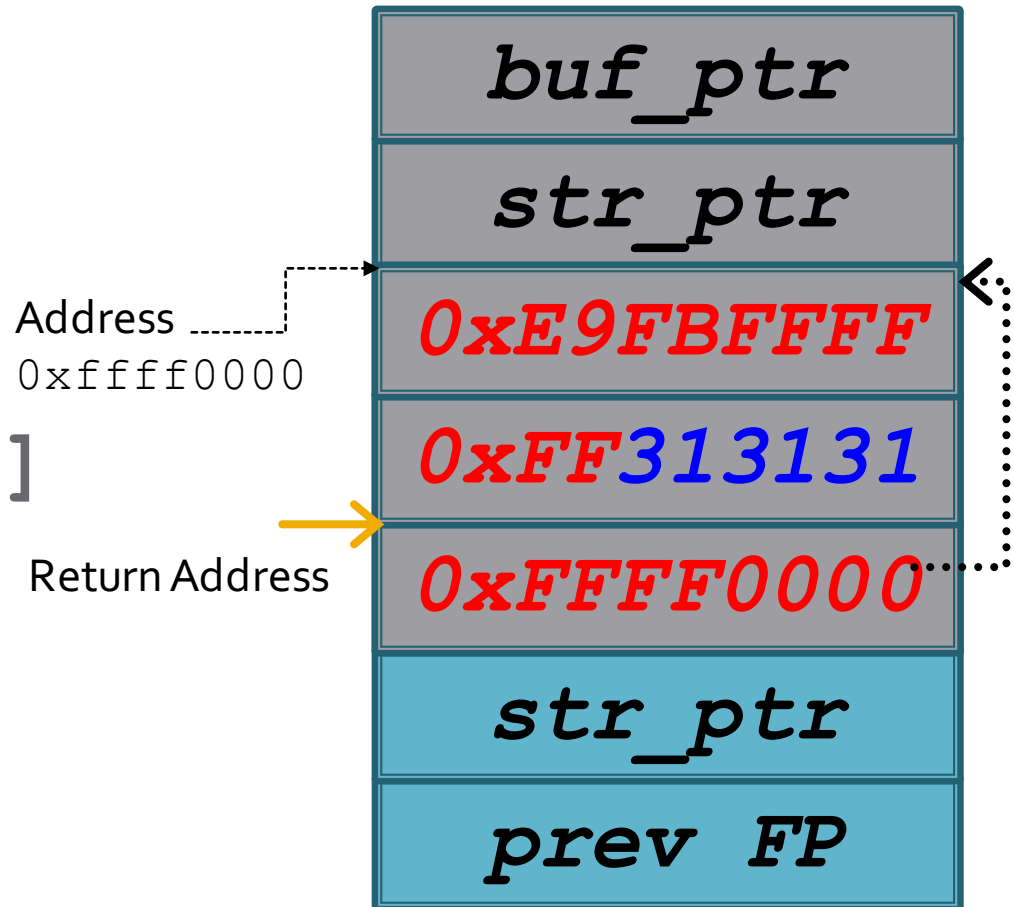
b <_main+0xb>

Stack Shellcode

foo:

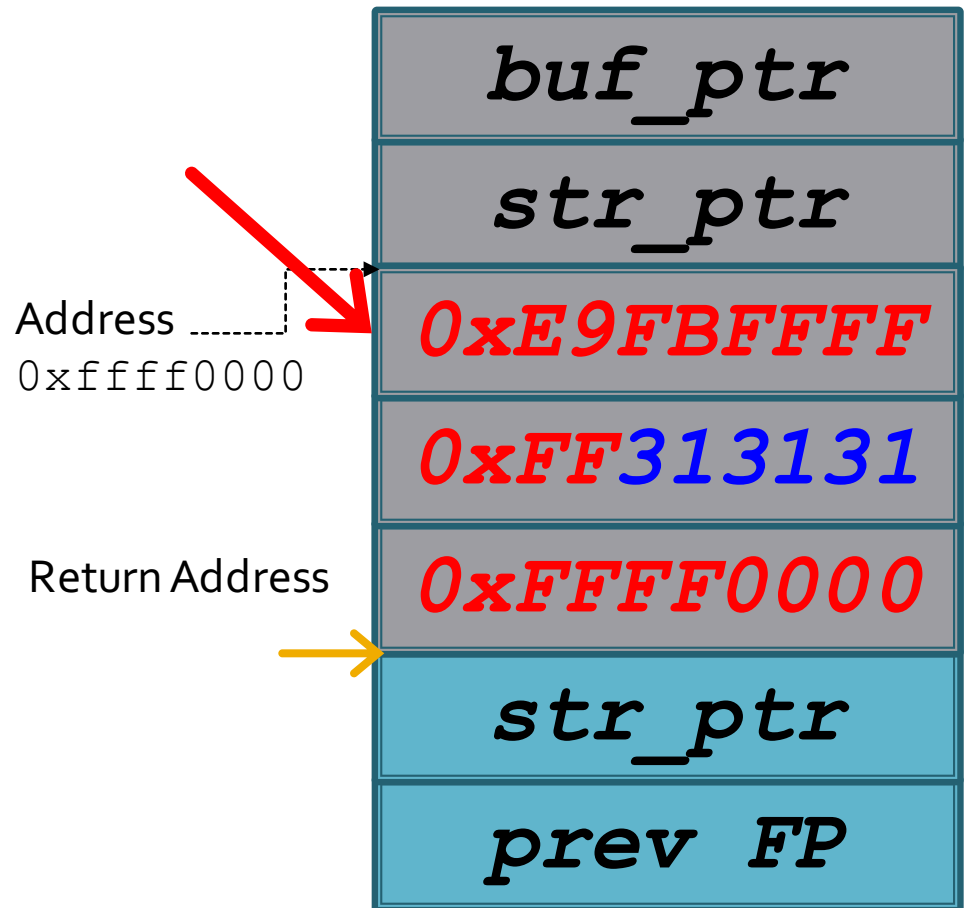
```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "123456789AB"



Stack Shellcode

```
shellcode:  
  jmp shellcode
```



Stack Shellcode

Let's do something more usefull than crashing

1. Compile your own code to be executed
2. Inject into the application
3. Jump to your binary instructions

Shellcode caveats

“Forbidden” characters

`strcpy()`:

`0x00`

`gets()`:

`“\n”`

`scanf()`:

Any whitespace

Heavily dependent on the vulnerability

Shellcode caveats

Hard to guess addresses

Shellcode address

Where is the code I injected?

Return address

Where do I tell the CPU where my code is?

Hard to guess address

shellcode

ret guess

?buff?

?buff?

?buff?

?buff/ret?

?buff/ret?

?buff/ret?

?ret?

Hard to guess address

shellcode

ret guess

ret guess

...

ret guess

?buff?

?buff?

?buff?

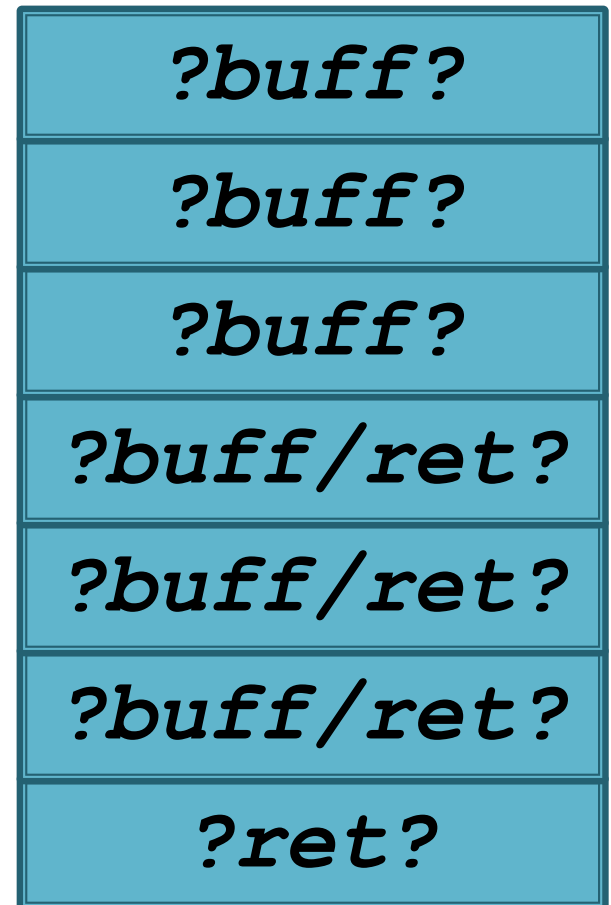
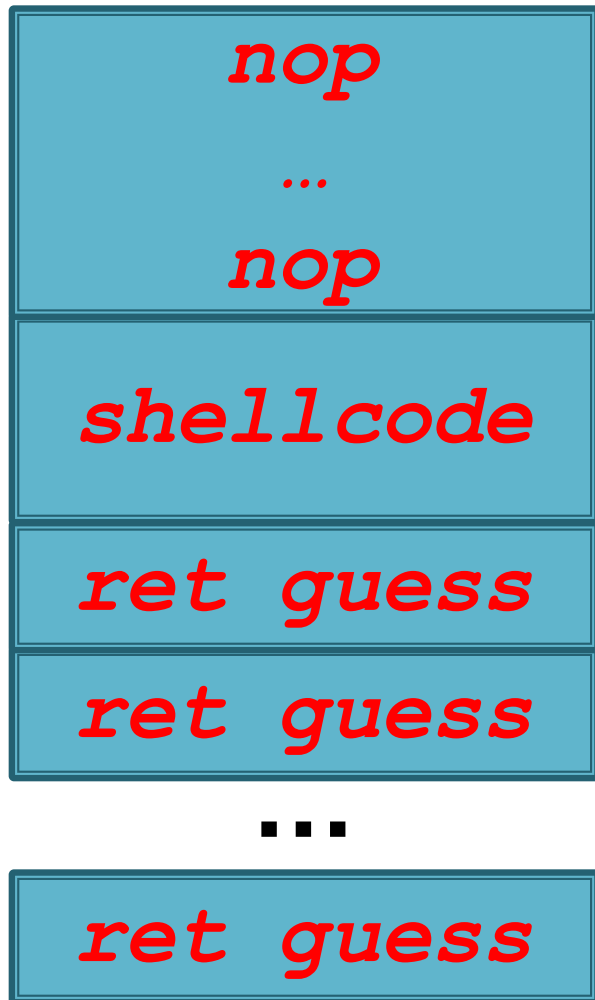
?buff/ret?

?buff/ret?

?buff/ret?

?ret?

Hard to guess address



Review

1. Find vulnerable code
(e.g. uncontrolled write)
2. Inject shellcode into the application
(i.e. any commands we want)
3. Redirect control to your shellcode

Vulnerability vs Exploit

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

Buffer Overflow
Stack Shellcode

DEP

Return-to-libc

Stack Canaries

Buffer Overread

ASLR

ROP

Fine-Grained ASLR

.....

Cat-and-Mouse Exploitation

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei^{*‡}, Dawn Song[‡]

[†]*Stony Brook University*

[‡]*University of California, Berkeley*

^{*}*Peking University*

Abstract—Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program’s behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated.

This paper sheds light on the primary reasons for this by describing attacks that succeed on today’s systems. We systematize the current knowledge about various protection techniques by setting up a general model for memory corruption attacks. Using this model we show what policies can stop which attacks. The model identifies weaknesses of currently deployed techniques, as well as other proposed protections enforcing stricter policies.

We analyze the reasons why protection mechanisms implementing stricter policies are not deployed. To achieve wide adoption, protection mechanisms must support a multitude of features and must satisfy a host of requirements. Especially important is performance, as experience shows that only solutions whose overhead is in reasonable bounds get deployed.

A comparison of different enforceable policies helps designers of new protection mechanisms in finding the balance between effectiveness (security) and efficiency. We identify some open research problems, and provide suggestions on improving the adoption of newer techniques.

try to write safe programs. The memory war effectively is an arms race between offense and defense. According to the MITRE ranking [1], memory corruption bugs are considered one of the top three most dangerous software errors. Google Chrome, one of the most secure web browsers written in C++, was exploited four times during the Pwn2Own/Pwnium hacking contests in 2012.

In the last 30 years a set of defenses has been developed against memory corruption attacks. Some of them are deployed in commodity systems and compilers, protecting applications from different forms of attacks. Stack cookies [2], exception handler validation [3], Data Execution Prevention [4] and Address Space Layout Randomization [5] make the exploitation of memory corruption bugs much harder, but several attack vectors are still effective under all these currently deployed basic protection settings. Return-Oriented Programming (ROP) [6], [7], [8], [9], [10], [11], information leaks [12], [13] and the prevalent use of user scripting and just-in-time compilation [14] allow attackers to carry out practically any attack despite all protections.

A multitude of defense mechanisms have been proposed to overcome one or more of the possible attack vectors. Yet most of them are not used in practice, due to one or more of the following factors: the *performance* overhead of the

Cat-and-Mouse Exploitation

DEP

Stack Canaries

ASLR

Fine-Grained ASLR

Buffer Overflow
Stack Shellcode

Return-to-libc

Buffer Overread

ROP

.....

DEP

Our problem:
data and code are the same

Solution:

$$W \oplus X$$

DEP

Pros:

Prevents attacker from injecting code to the stack

Cons:

Requires hardware support

Does not prevent all attacks

DEP

foo:

```
push    ebp
mov     ebp, esp
sub     esp, 4
push    [ebp + 8]
push    ebp - 4
call    strcpy
leave
ret
```

str_ptr: "123456789AB"

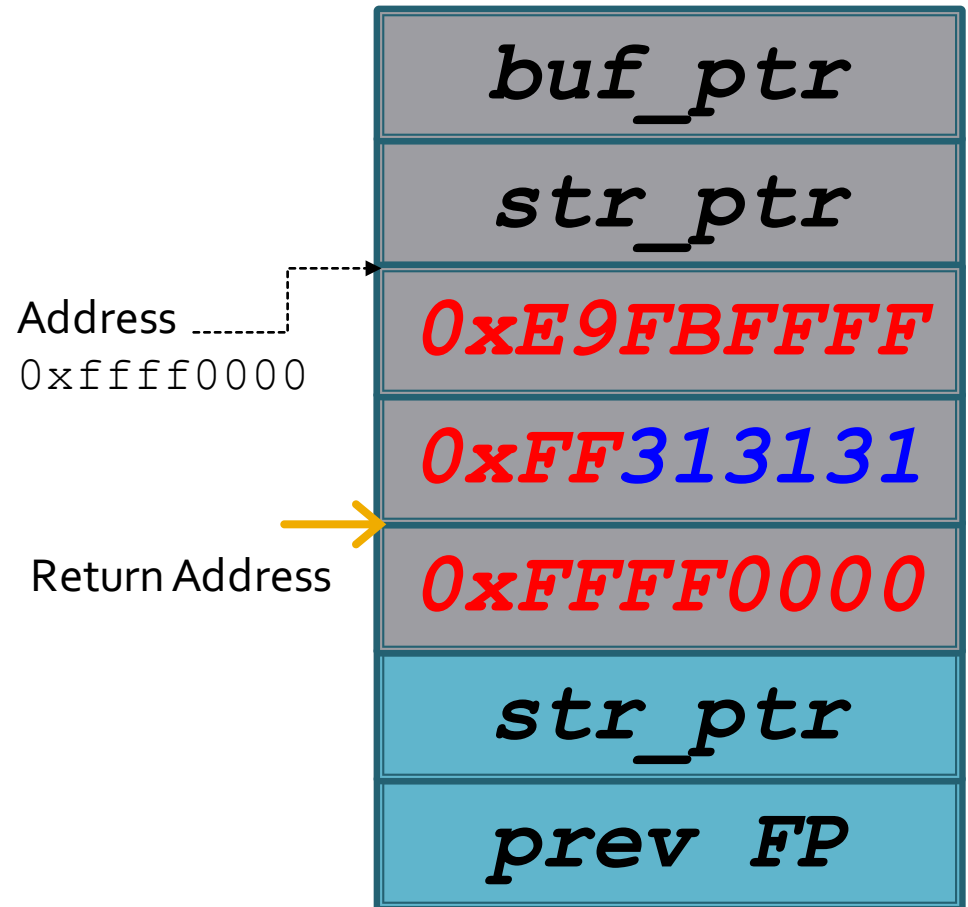
Address
0xffff0000

Return Address



DEP

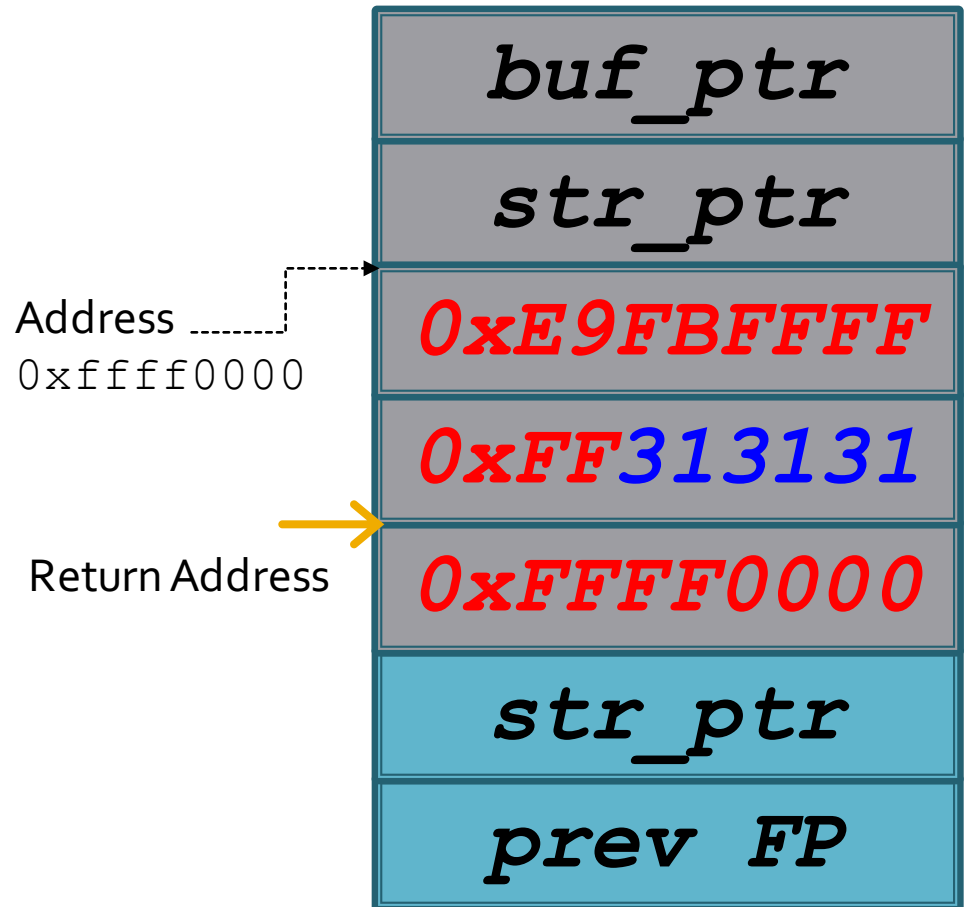
OS ERROR



DEP

OS ERROR

CONTROL FLOW
IS INCORRECT

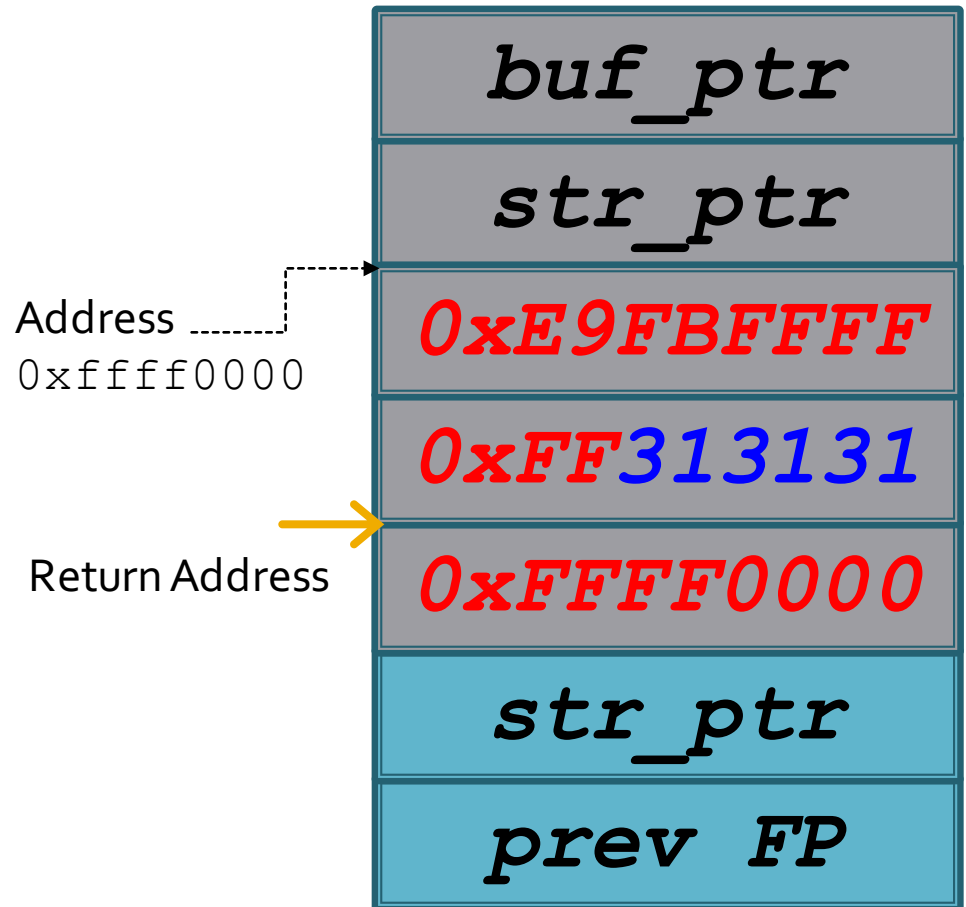


DEP

OS ERROR

CONTROL FLOW
IS INCORRECT

IMMEDIATELY
END PROCESS



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

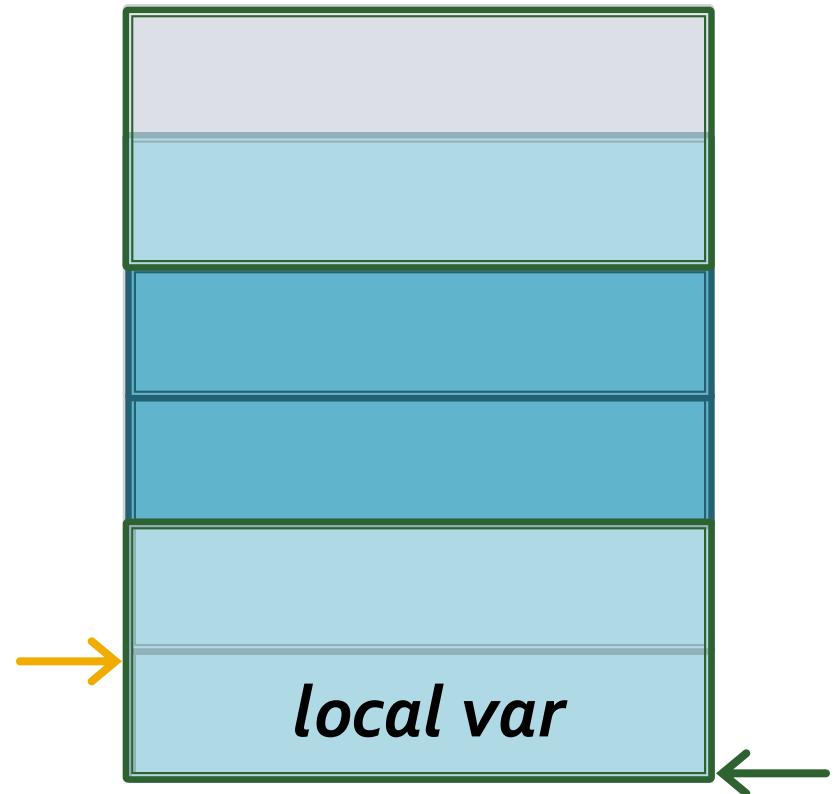
jump foo

; mov **eip**, foo

What are the contents of the stack?

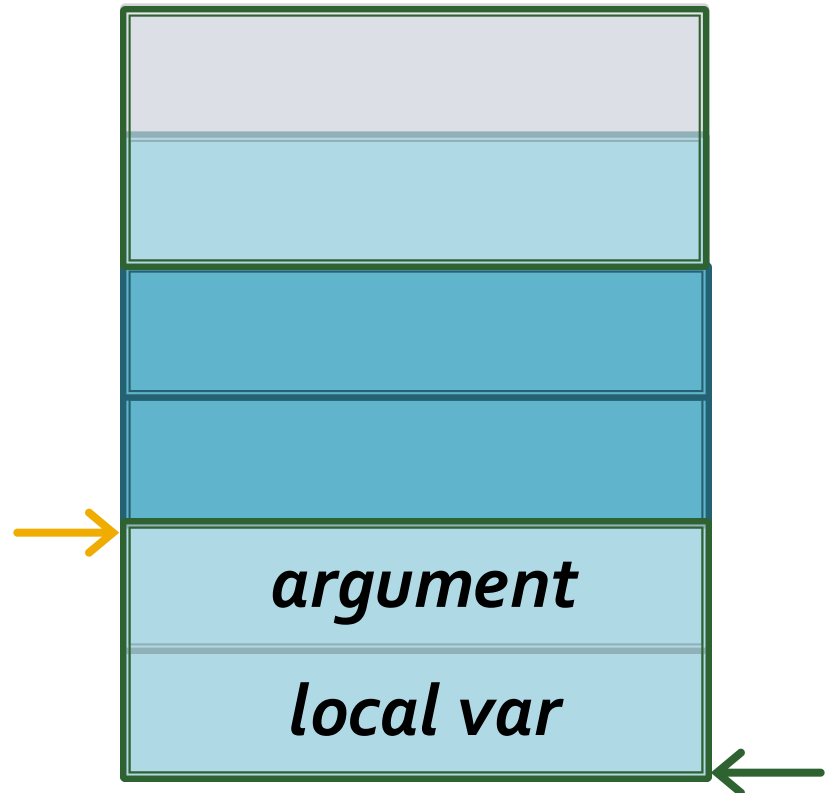
Return to libc

SETUP AS A FUNCTION CALL



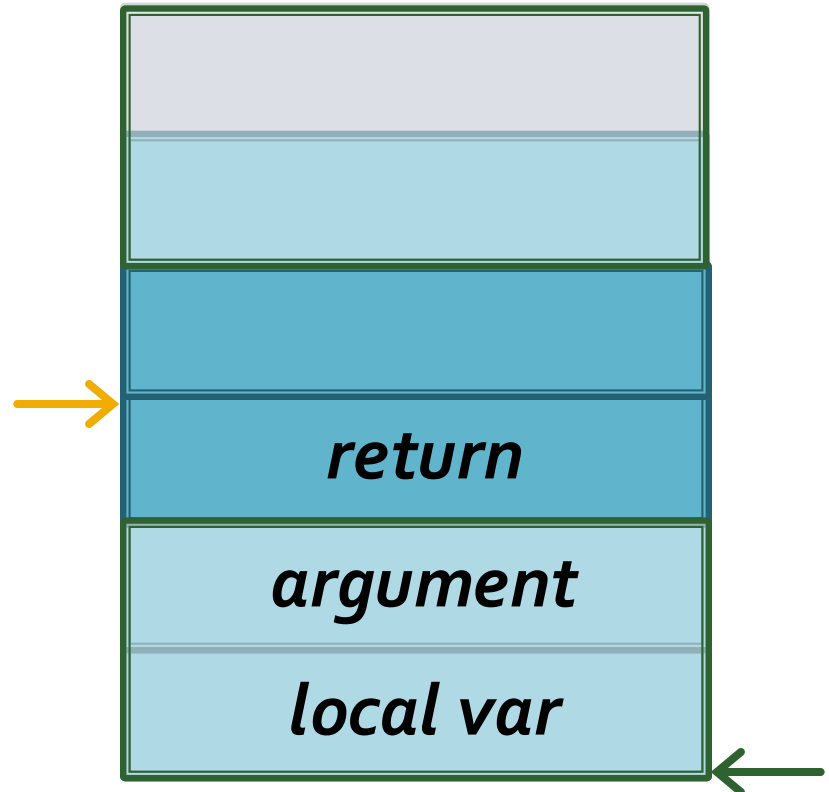
Return to libc

SETUP AS A FUNCTION CALL



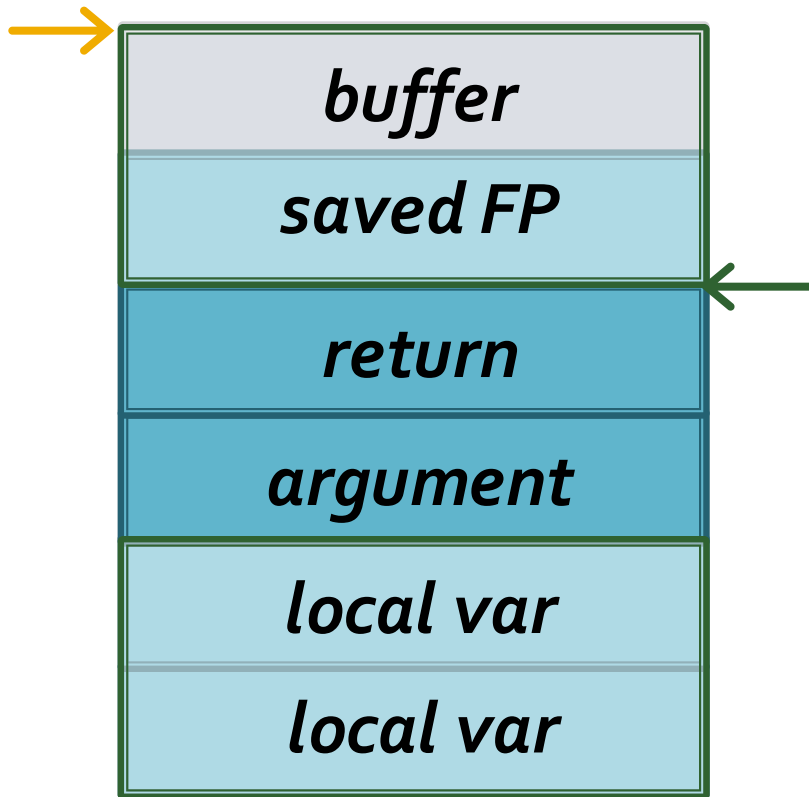
Return to libc

SETUP AS A FUNCTION CALL

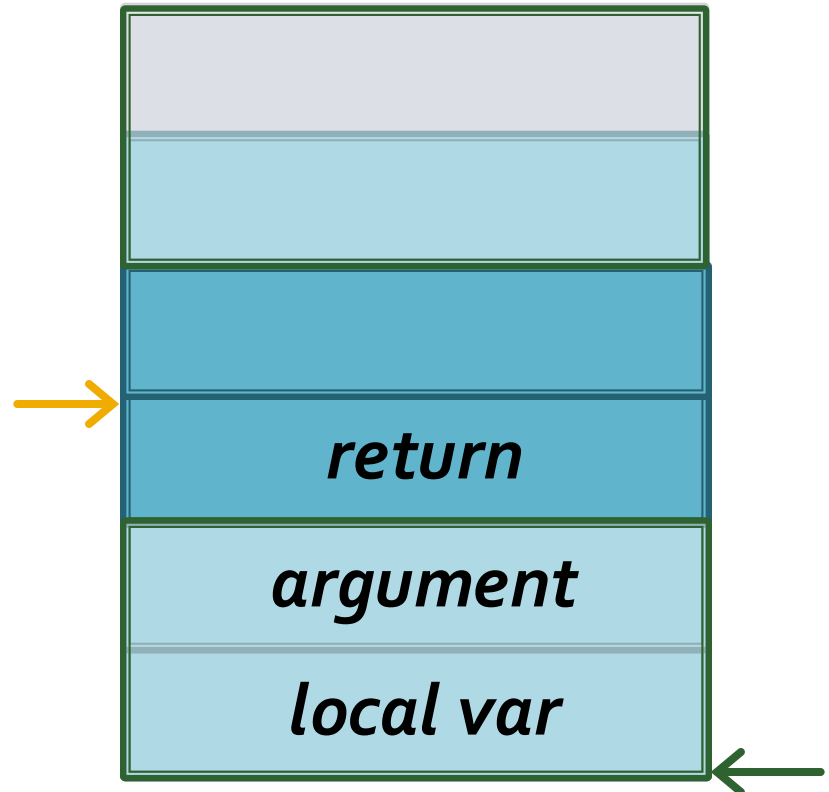


Return to libc

SETUP AS A RETURN

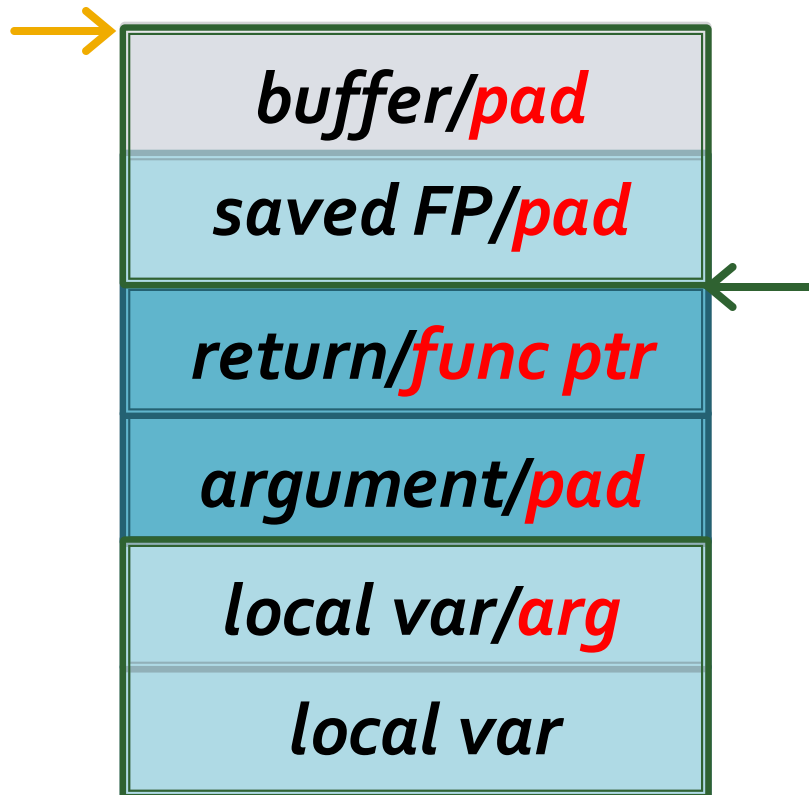


SETUP AS A FUNCTION CALL

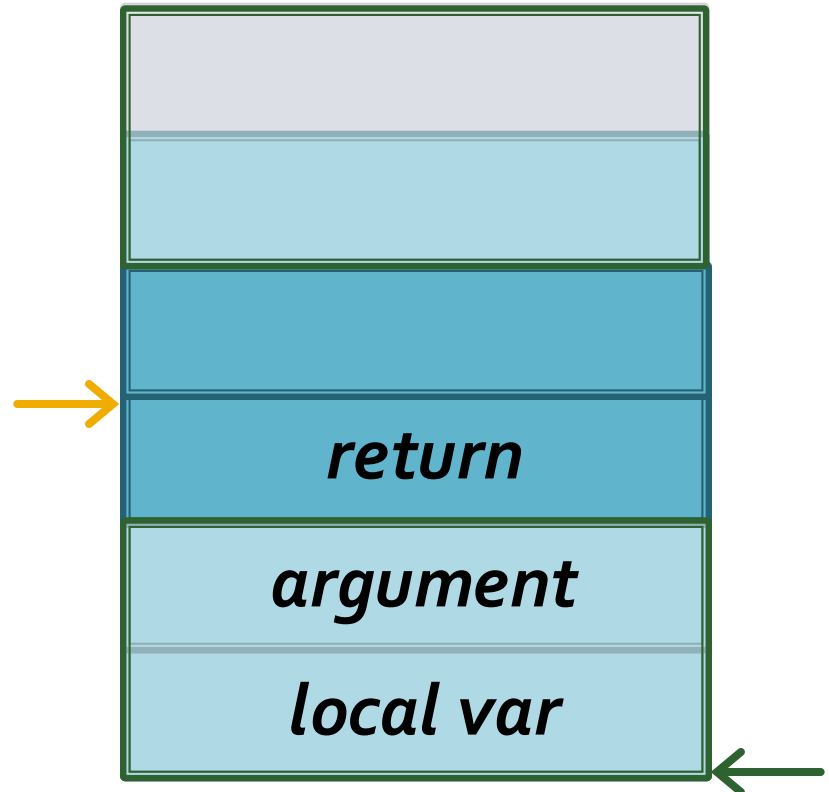


Return to libc

SETUP AS A RETURN

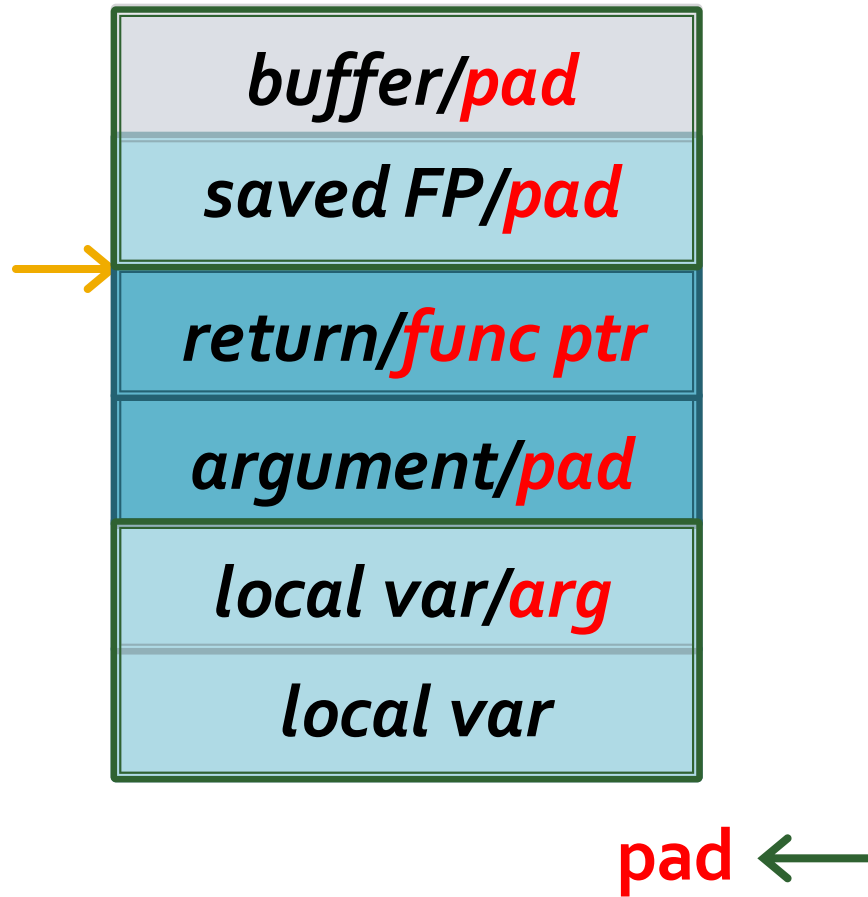


SETUP AS A FUNCTION CALL

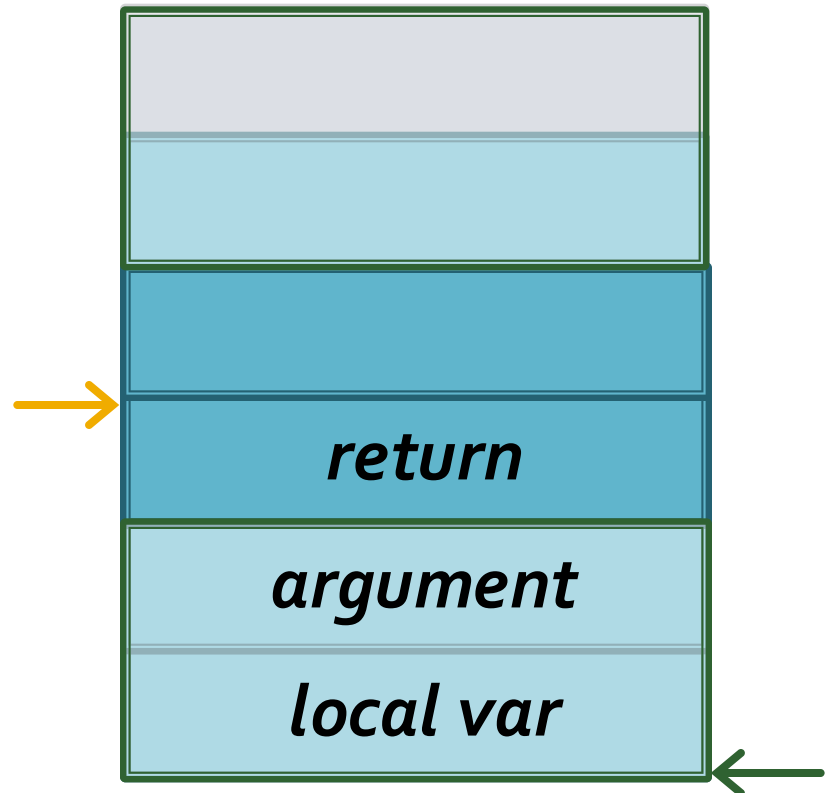


Return to libc

SETUP AS A RETURN

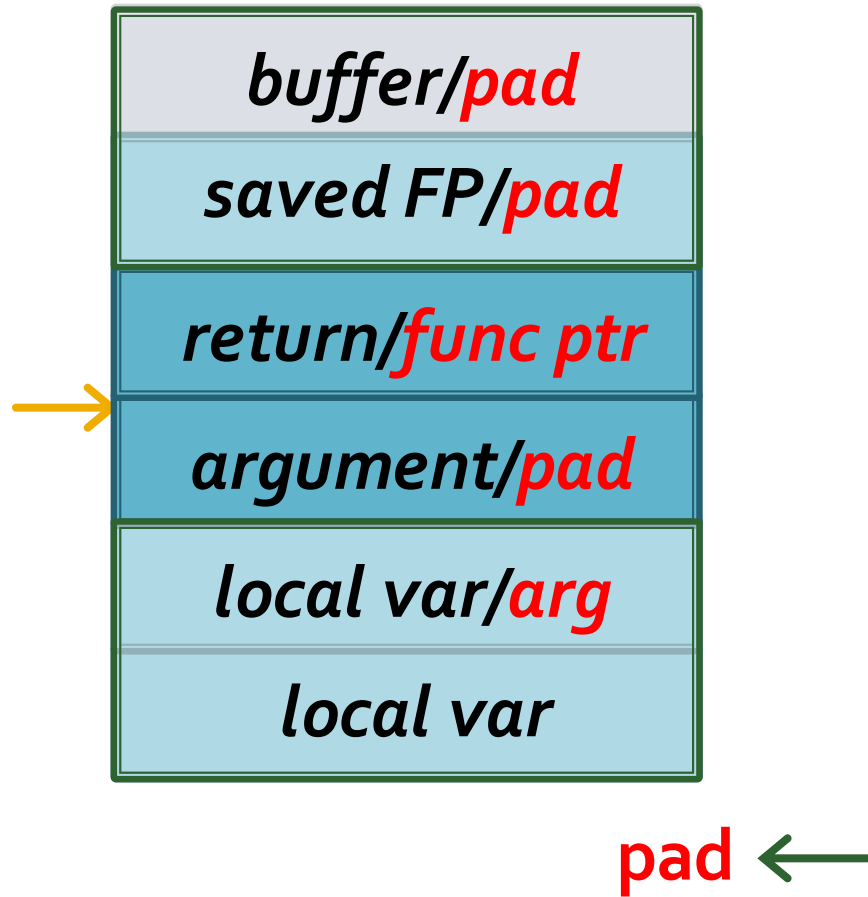


SETUP AS A FUNCTION CALL

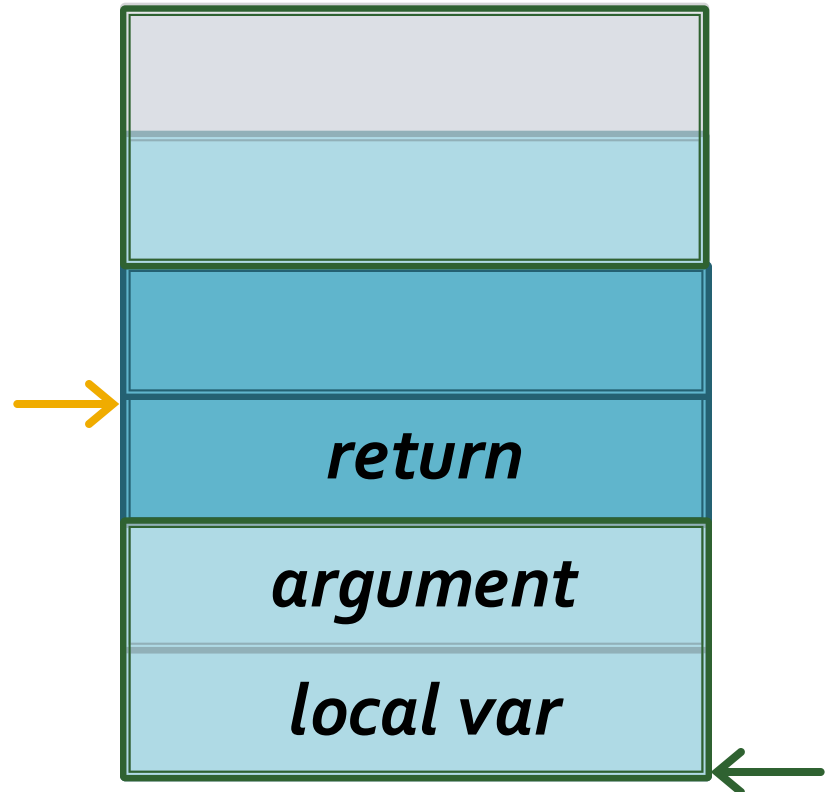


Return to libc

SETUP AS A RETURN

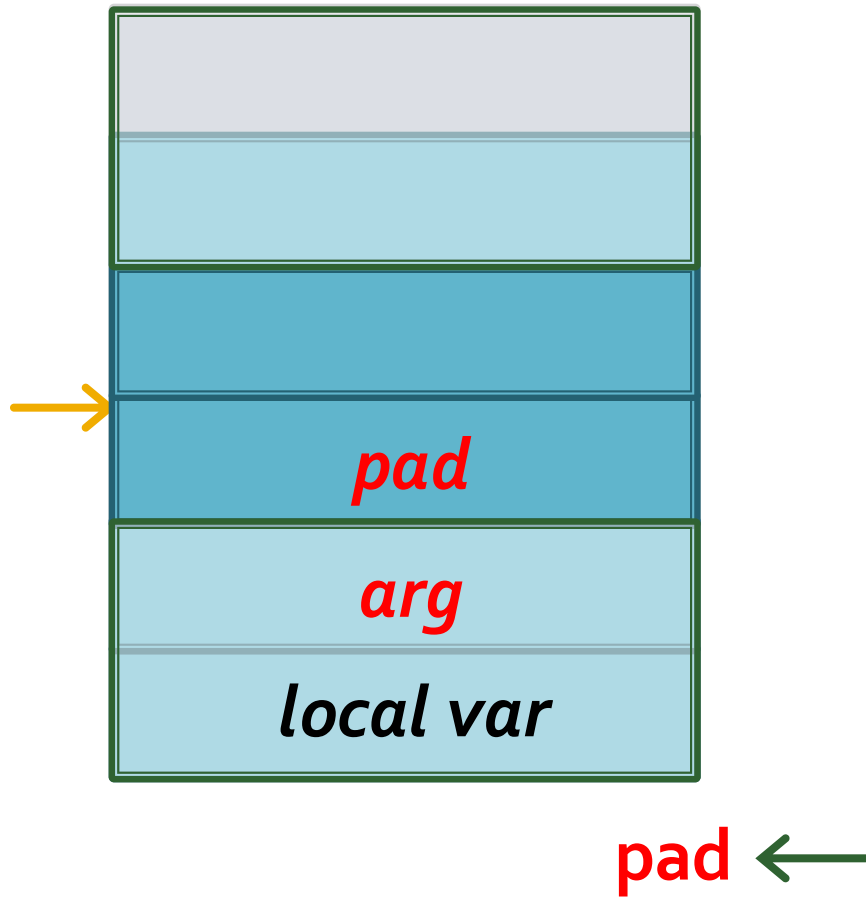


SETUP AS A FUNCTION CALL

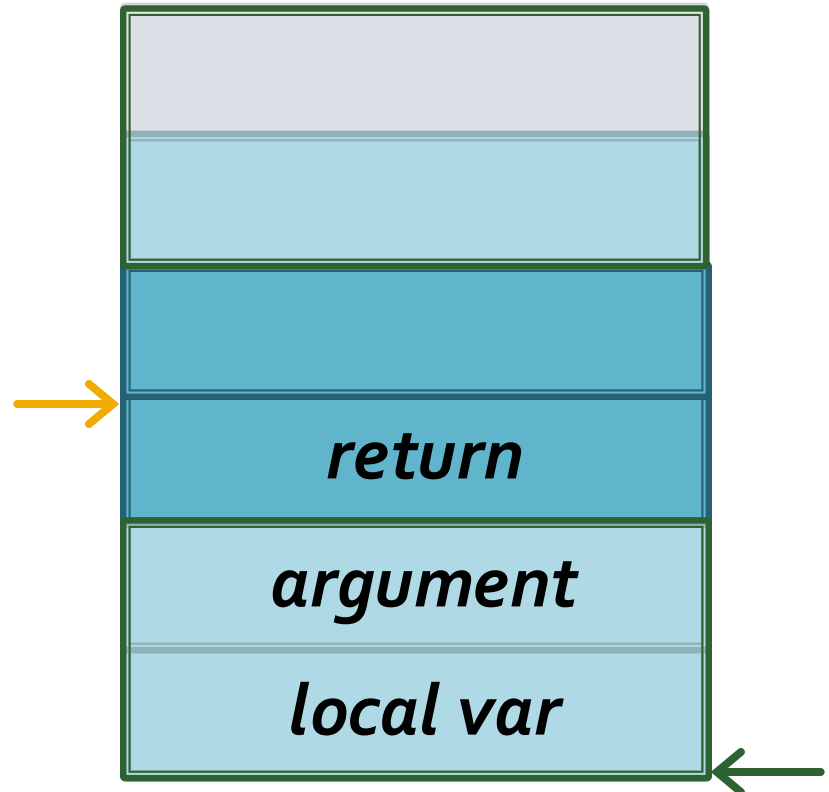


Return to libc

SETUP AS A RETURN

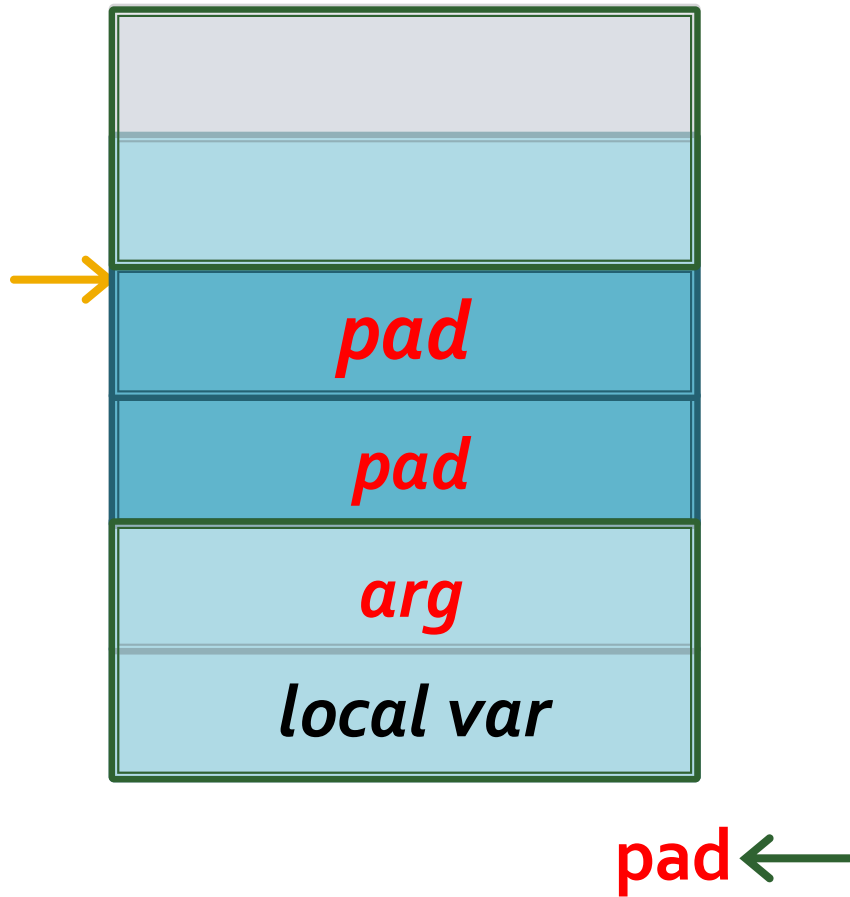


SETUP AS A FUNCTION CALL

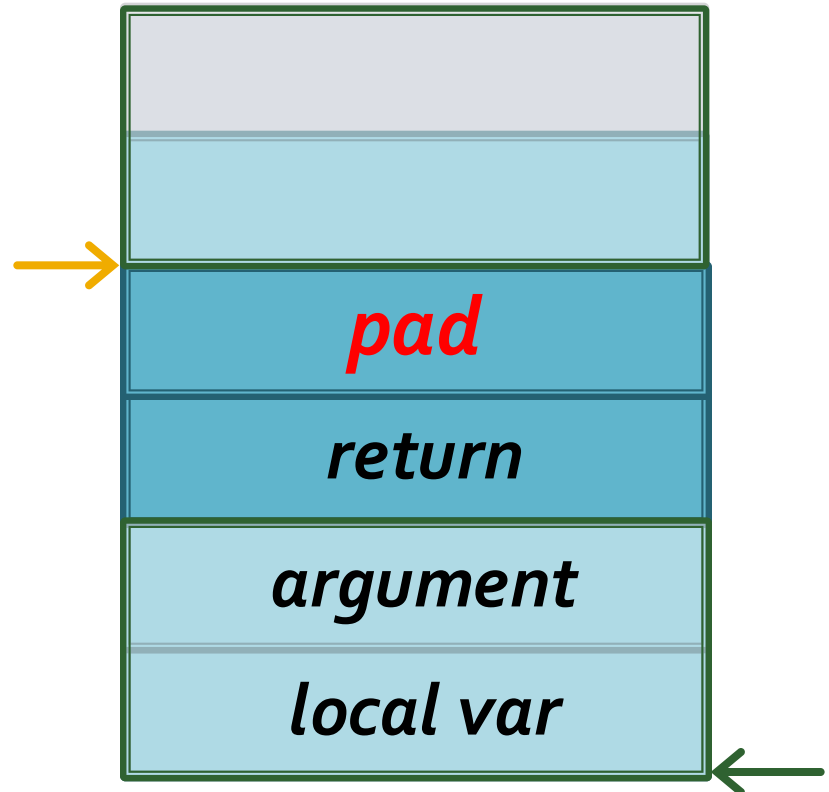


Return to libc

SETUP AS A RETURN

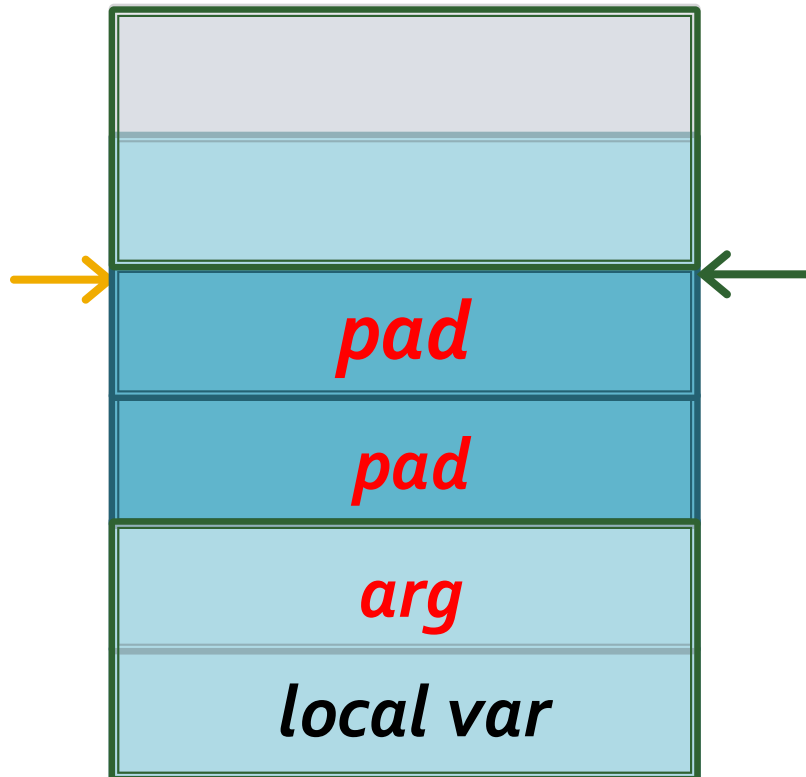


SETUP AS A FUNCTION CALL

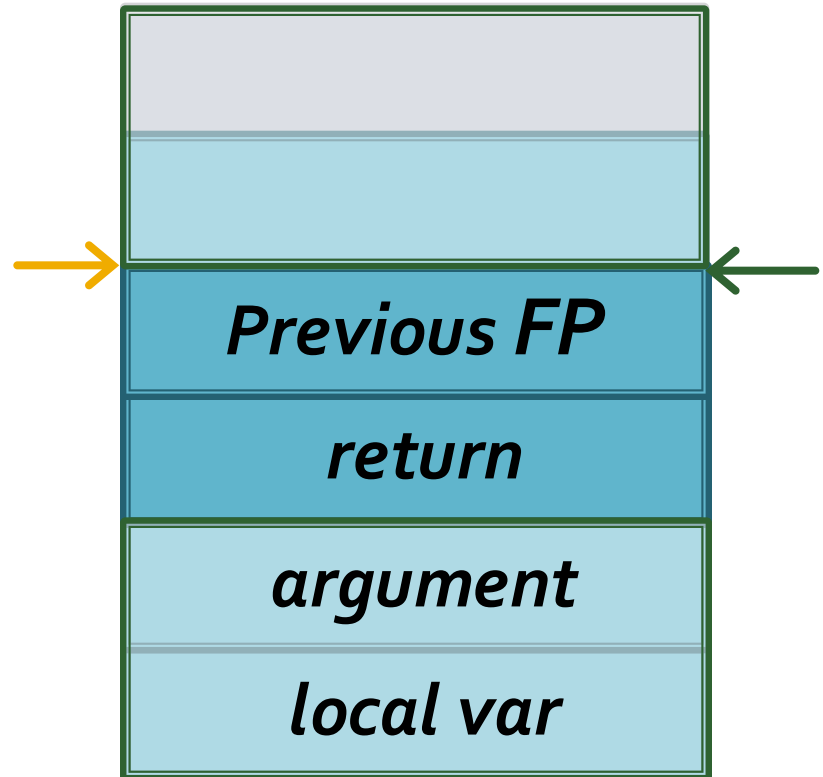


Return to libc

SETUP AS A RETURN

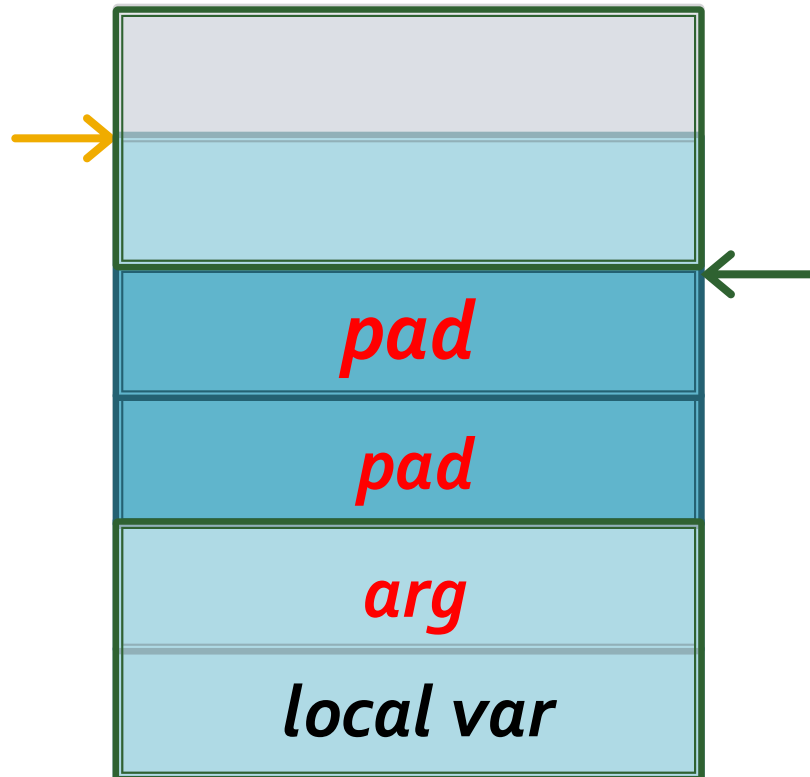


SETUP AS A FUNCTION CALL

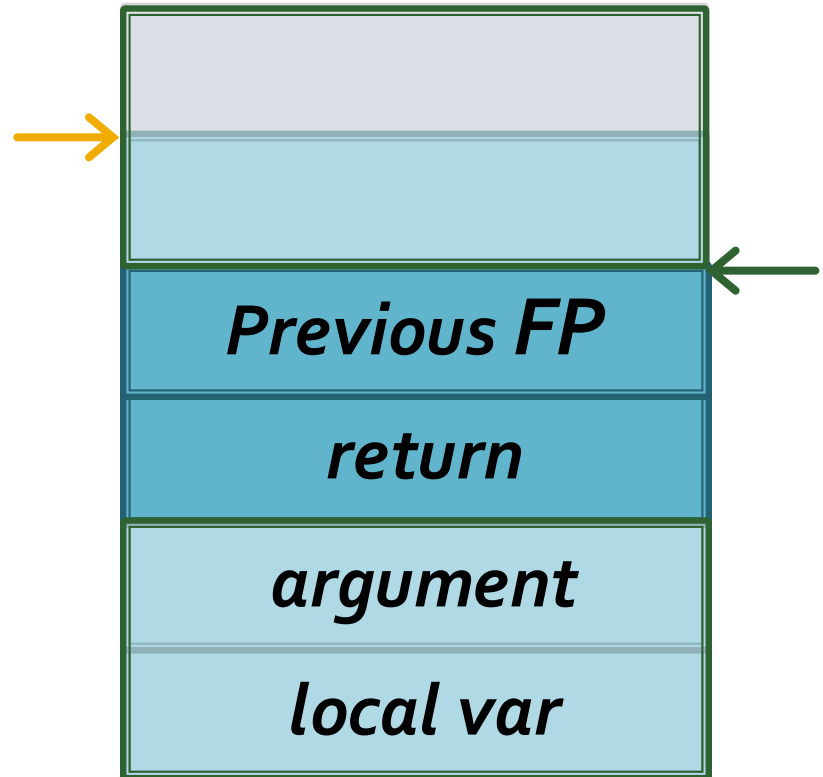


Return to libc

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

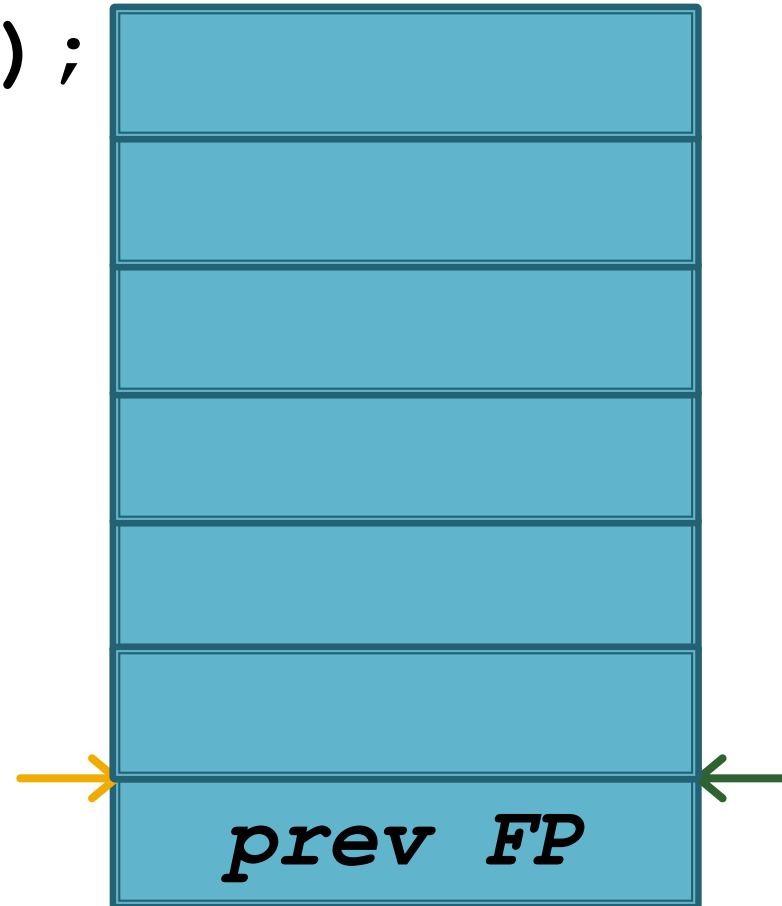
```
int main() {  
    char* args[] = {"/bin/ls",  
                     NULL};  
    execv("/bin/ls", args);  
}
```

Return to libc

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

Text:

path_str: "/bin/ls"

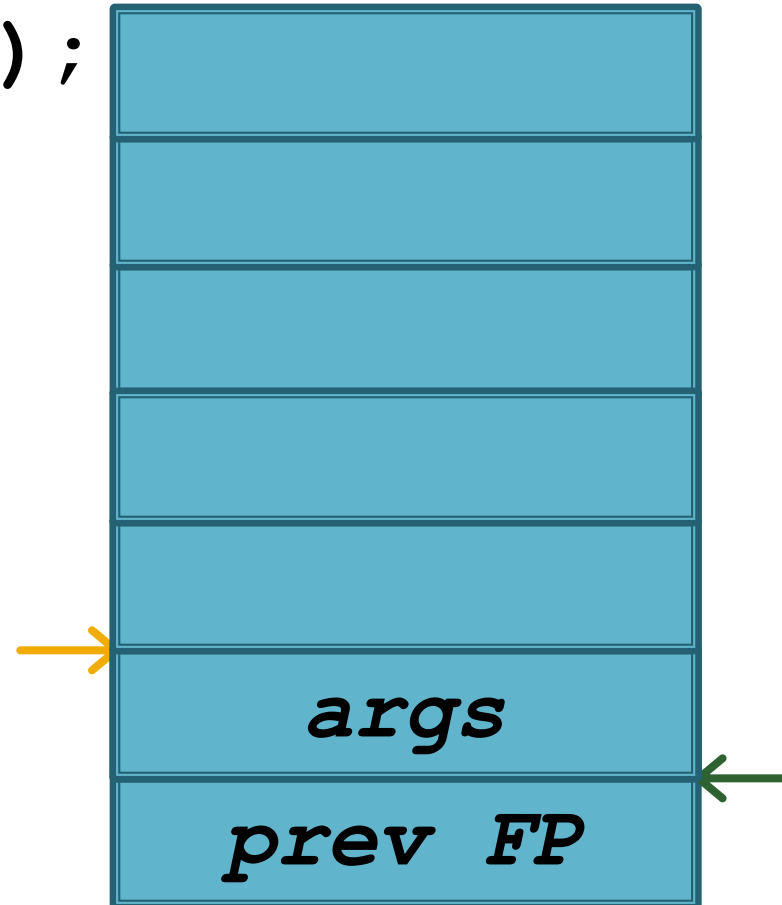


Return to libc

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

Text:

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

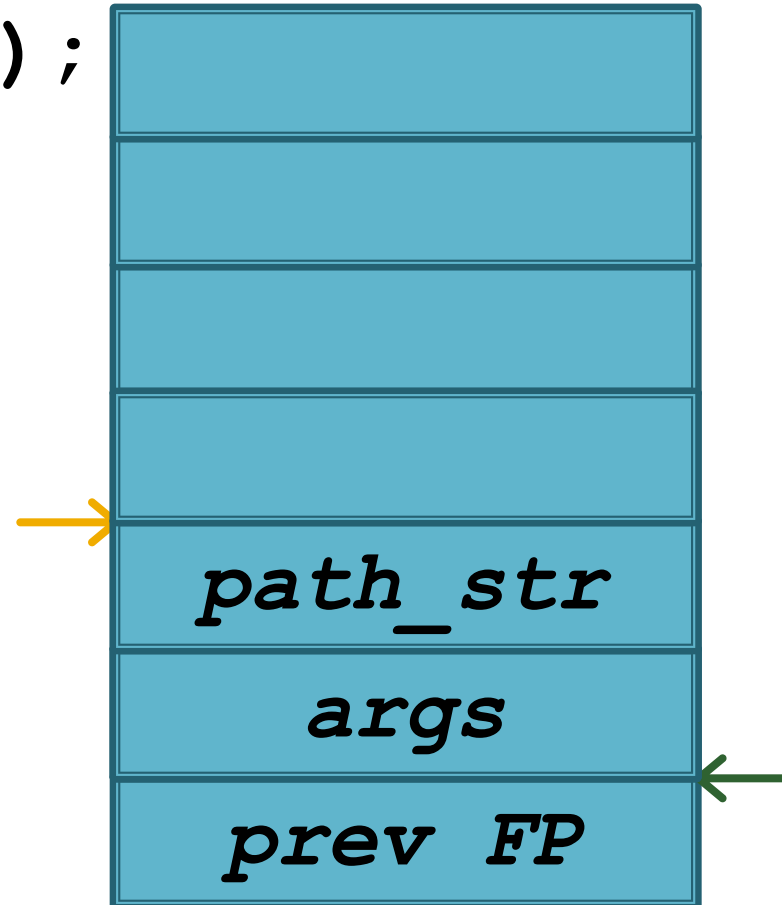


Return to libc

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

Text:

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

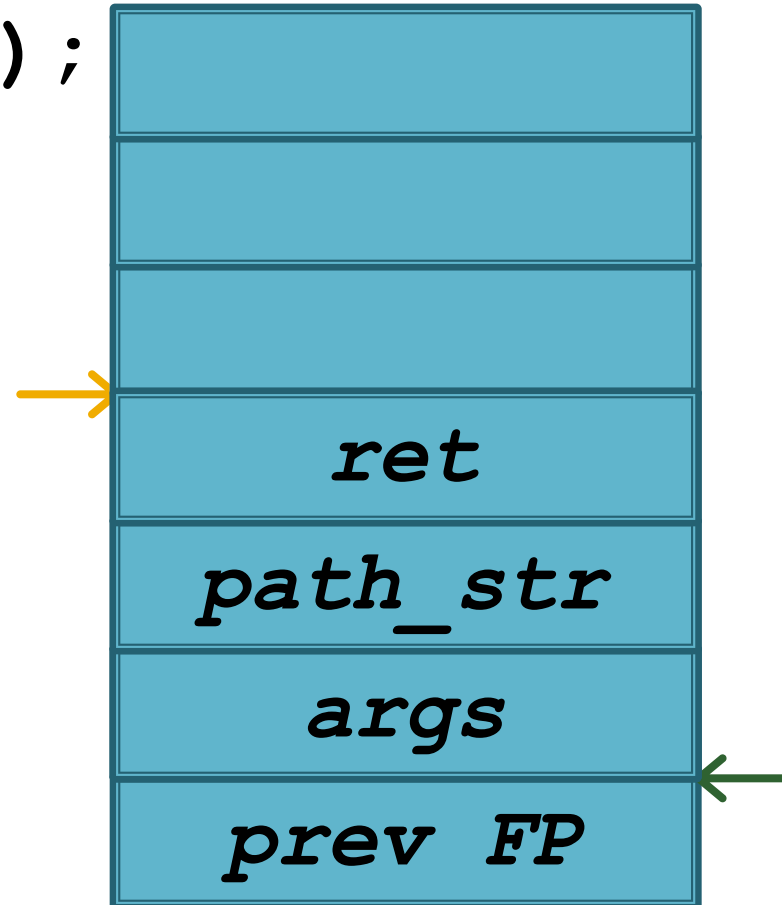


Return to libc

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

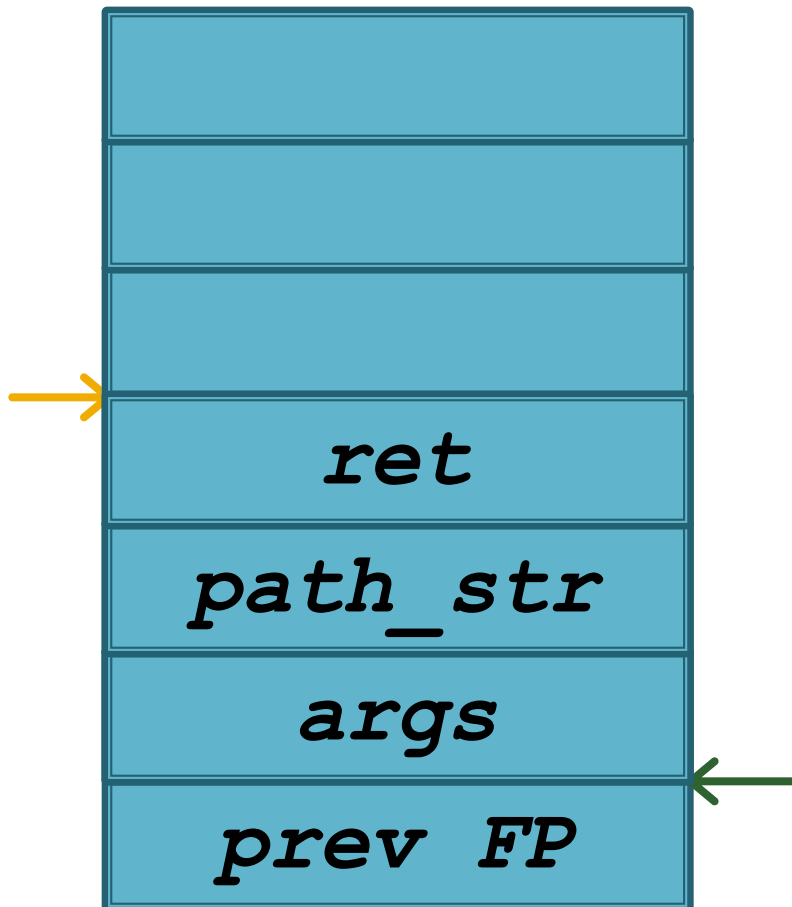
Text:

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



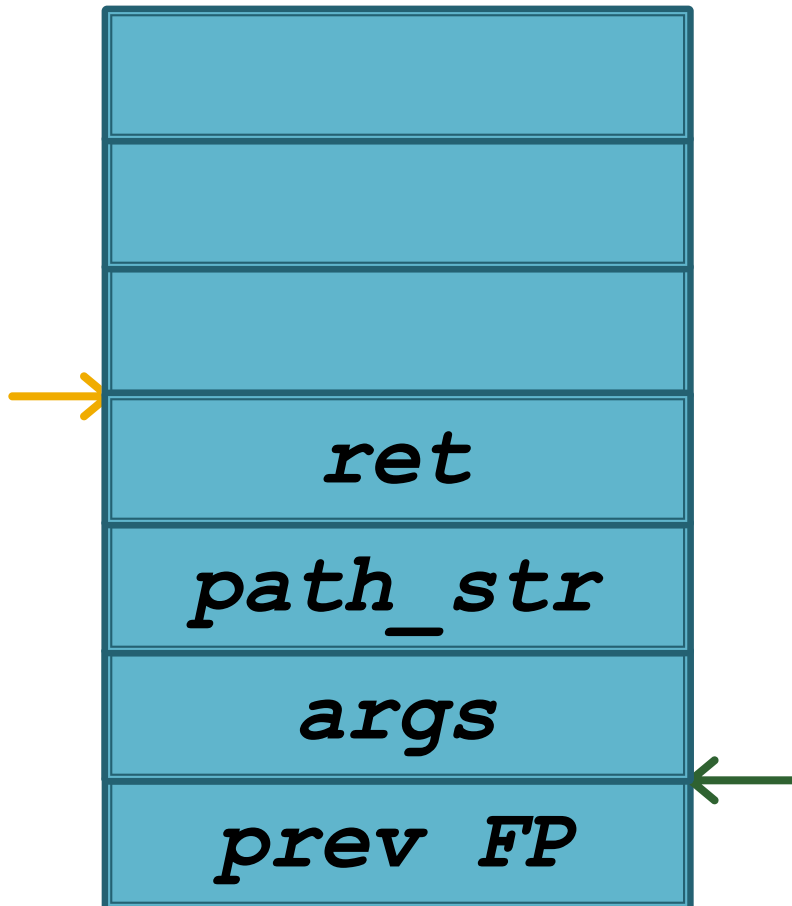
Return to libc

AS A FUNCTION CALL

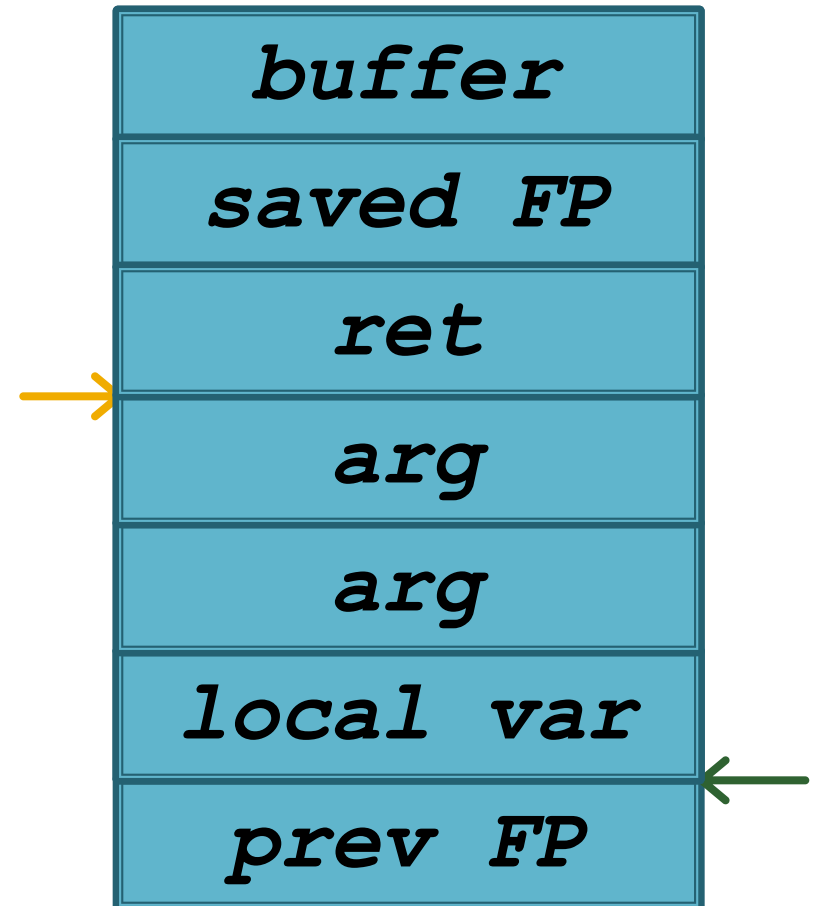


Return to libc

AS A FUNCTION CALL

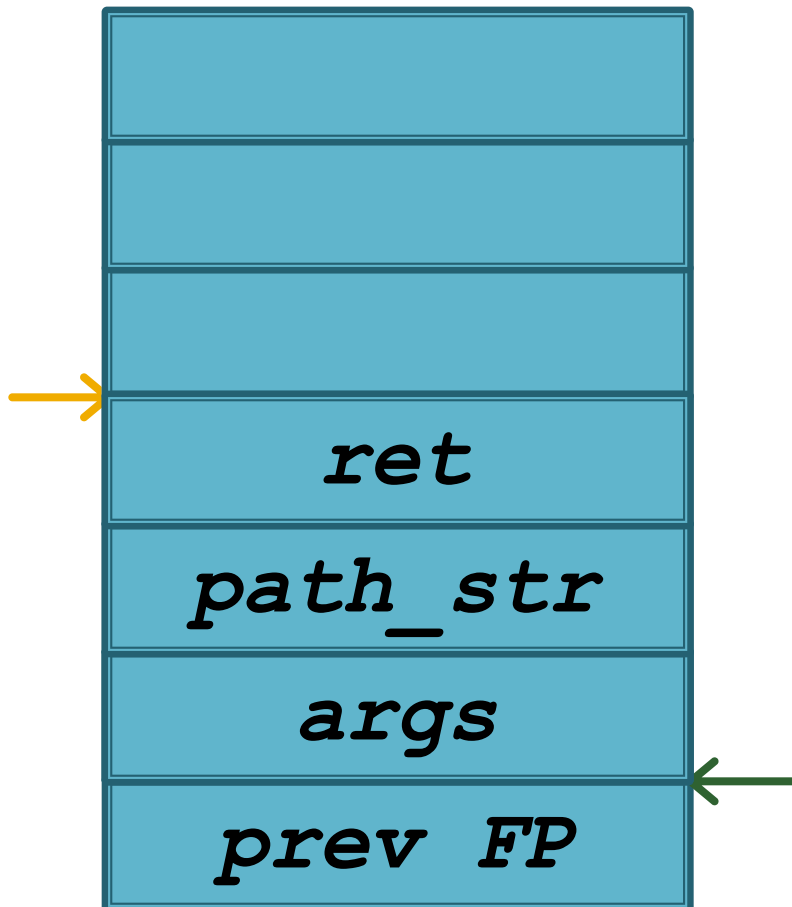


AS A RETURN TO LIBC

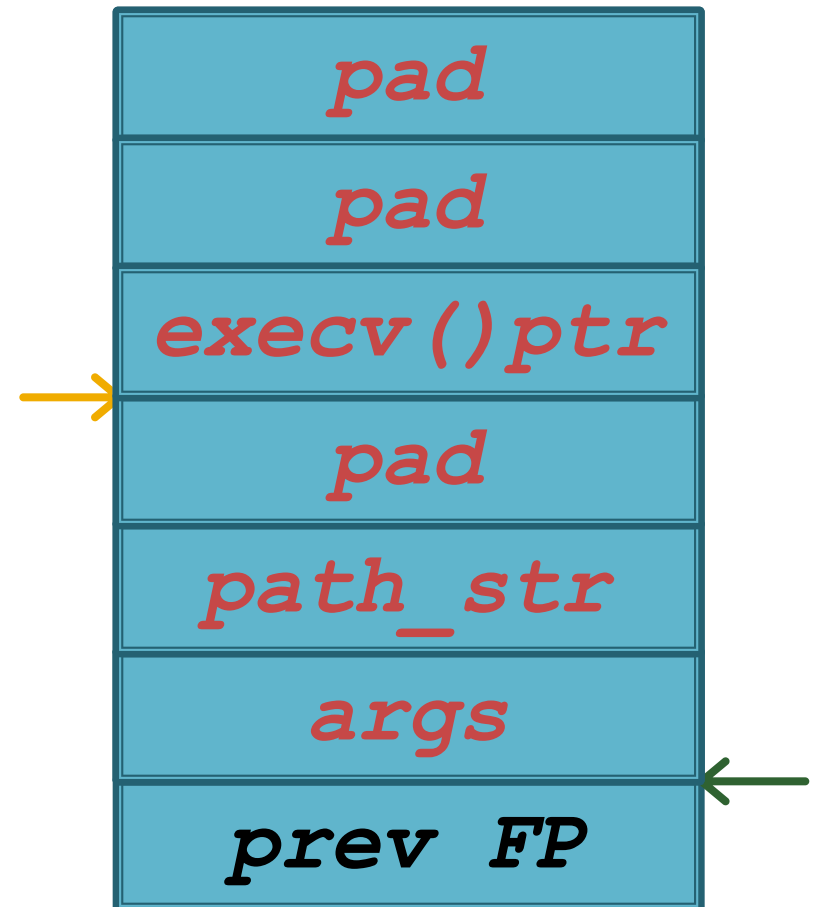


Return to libc

AS A FUNCTION CALL

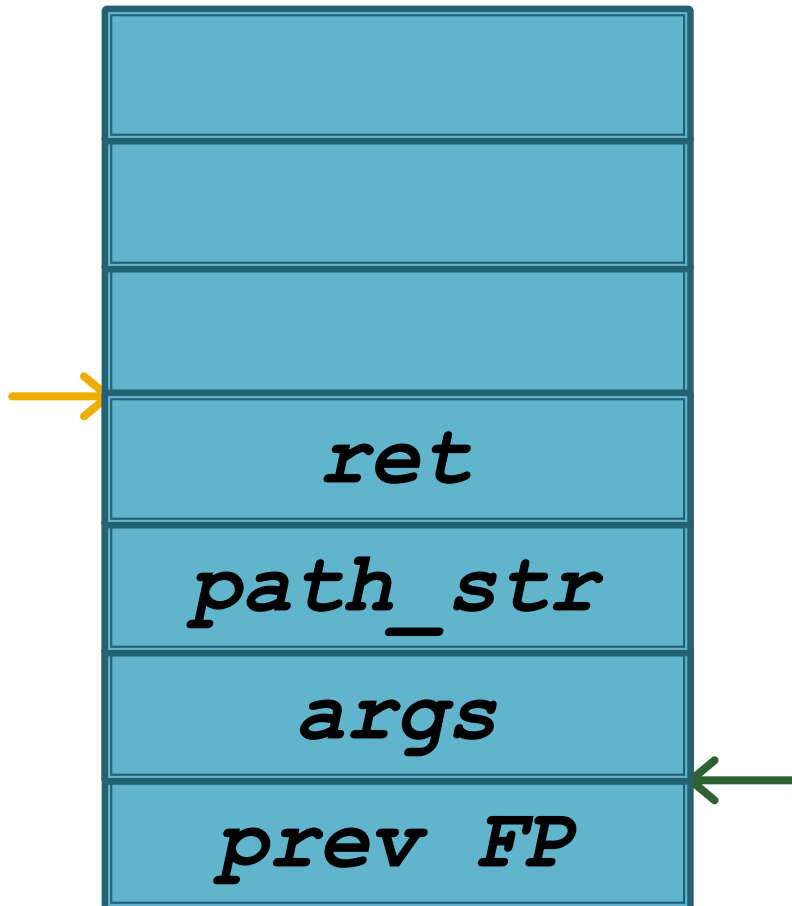


AS A RETURN TO LIBC

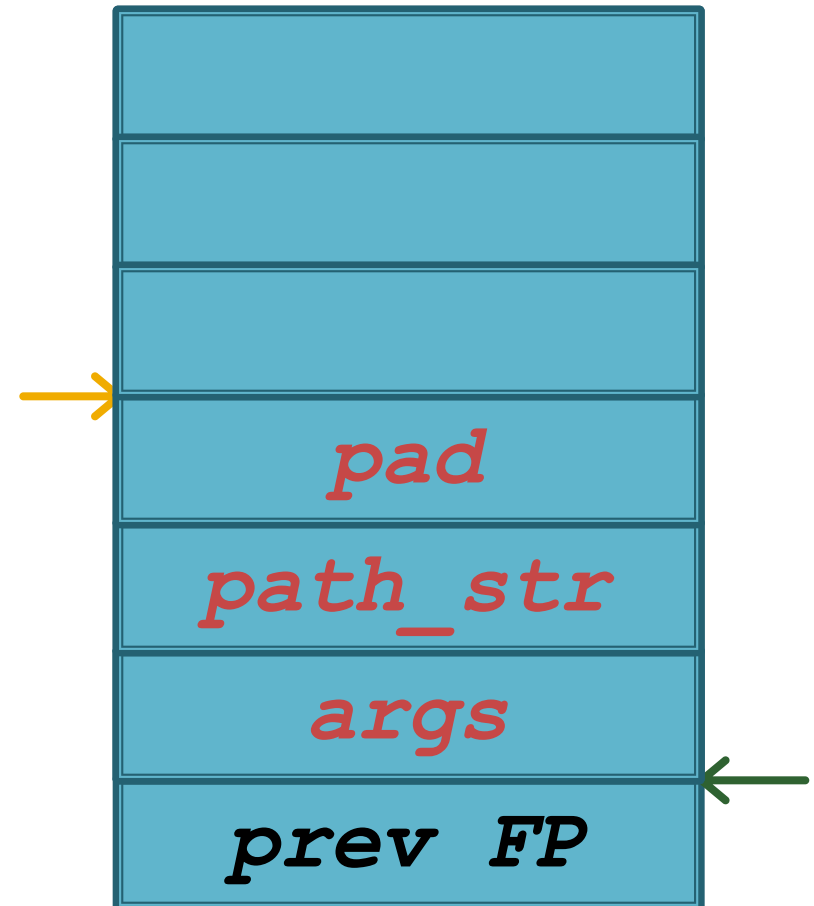


Return to libc

AS A FUNCTION CALL

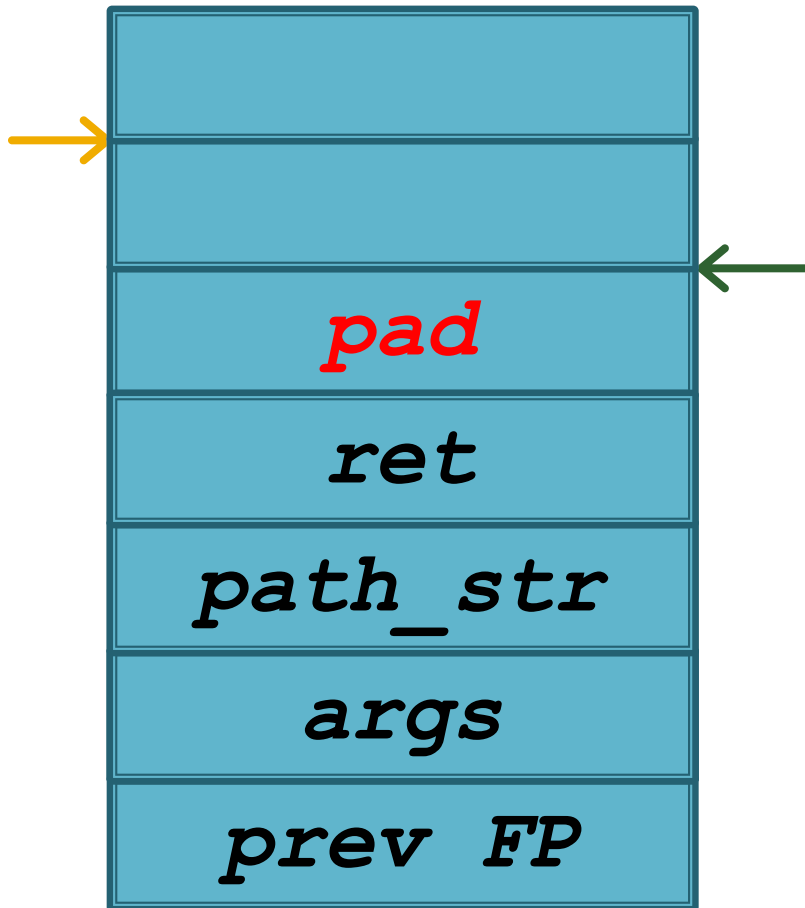


AS A RETURN TO LIBC

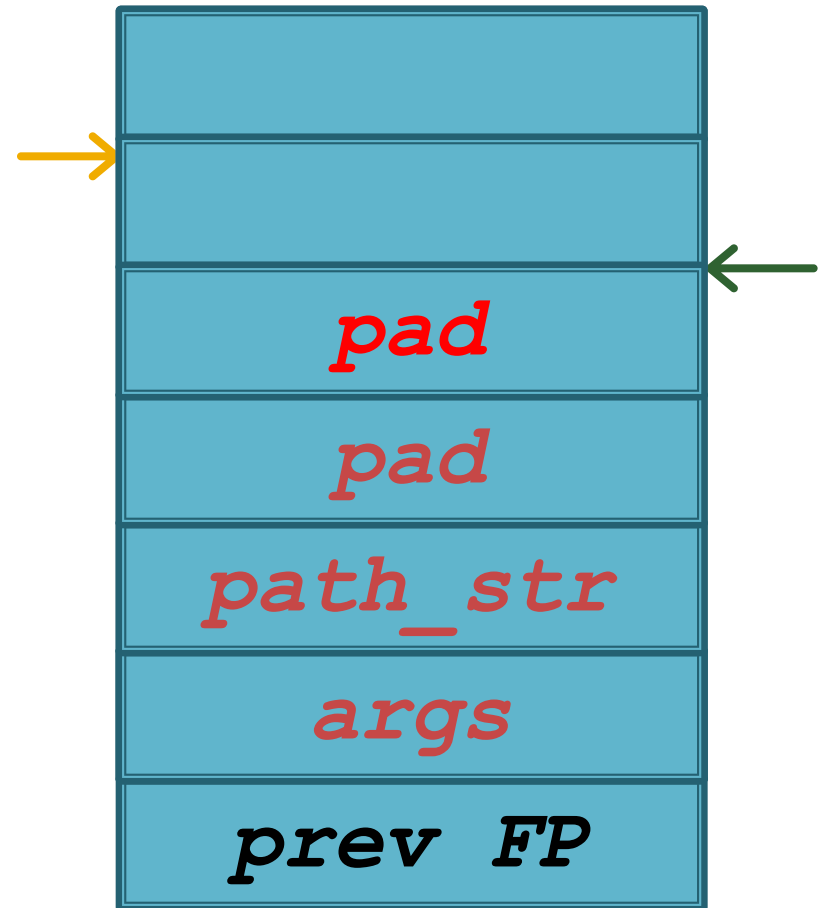


Return to libc

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

buffers

canary

main FP

return

Stack Canaries

```
# vulnerability:
```

```
strcpy(buffer, str)
```

AAAAAAAA...

0x41414141

0x41414141

0x41414141

Stack Canaries

```
# on return:
```

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

AAAAAAAAA...

0x41414141

0x41414141

0x41414141

Stack Canaries

*** stack smashing detected ***

```
# on return:
```

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

AAAAAAAAAA...

0x41414141

0x41414141

0x41414141

Cat-and-Mouse Exploitation

Return-to-libc

Stack Canaries

Buffer Over-read

Integer Overflow

ROP

ASLR

Automated Testing

Toolbox of Exploitation Techniques

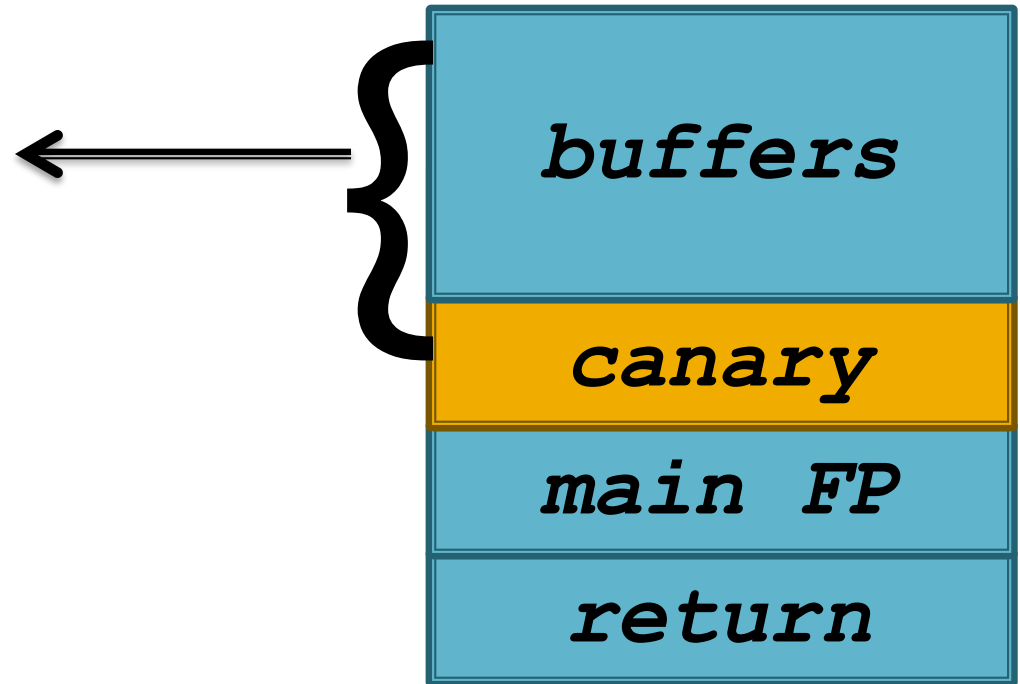
Buffer Over-read

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

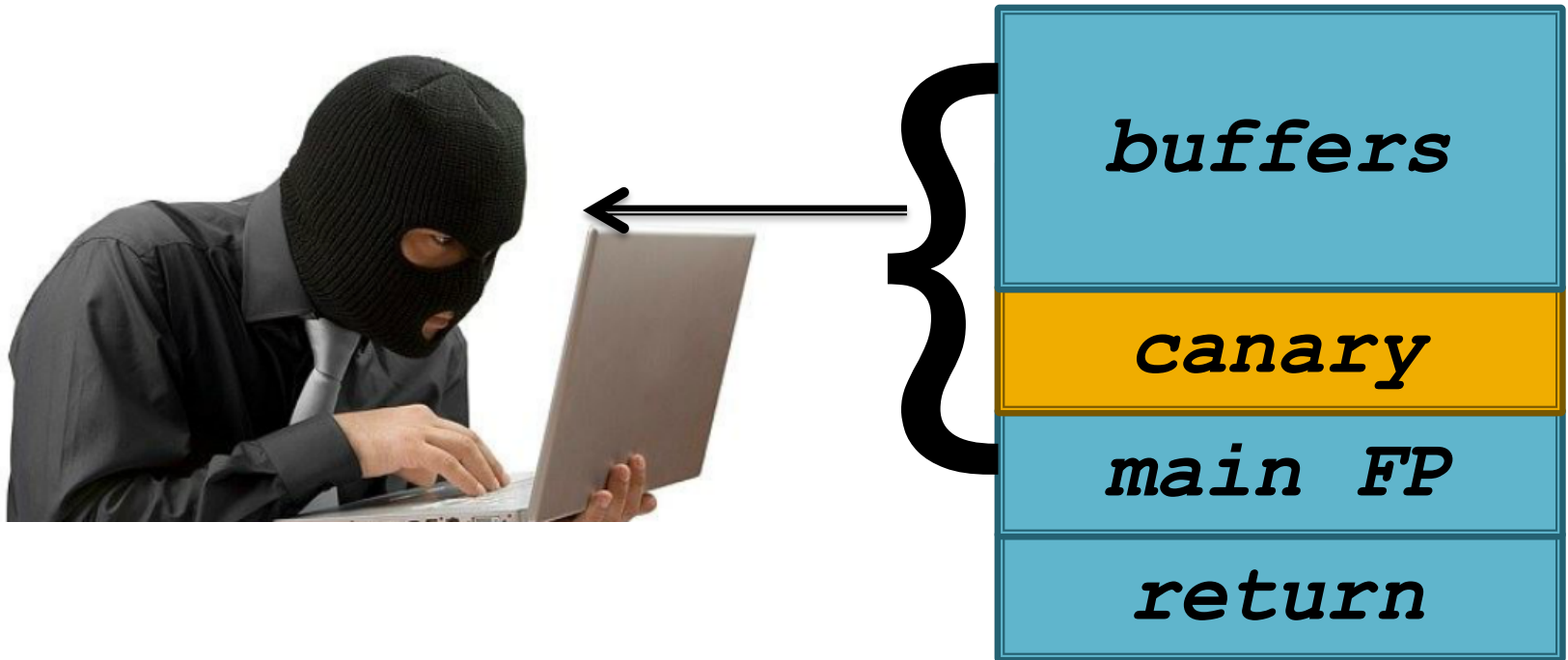
Buffer Over-read

```
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
```



buffers

canary

main FP

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 `strncpy` is easy

Solution:

Find values of `n` that break `strncpy`

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

```
void foo(int *array, int len) {  
    int *buf;  
    size  ➤ buf = malloc(len * sizeof(int));  
    200  if (!buf)  
    buffer    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

```
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



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`

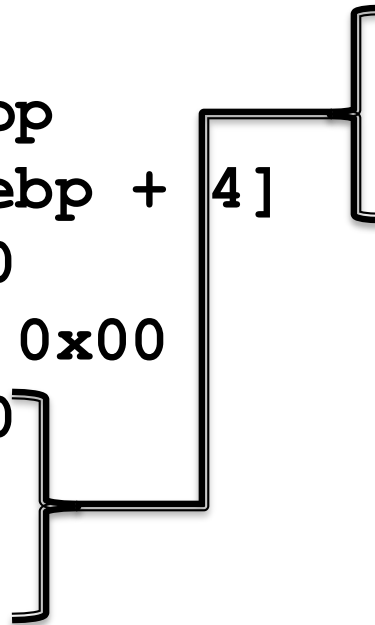
`ret`

`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

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%

File Edit Tabs Help

```
8052867: 74 1c je 8052885 <__sprintf_chk@plt+0x8ab5>
8052869: 8b 64 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 d8 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%

ROP

Bytes in the Code Section:

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

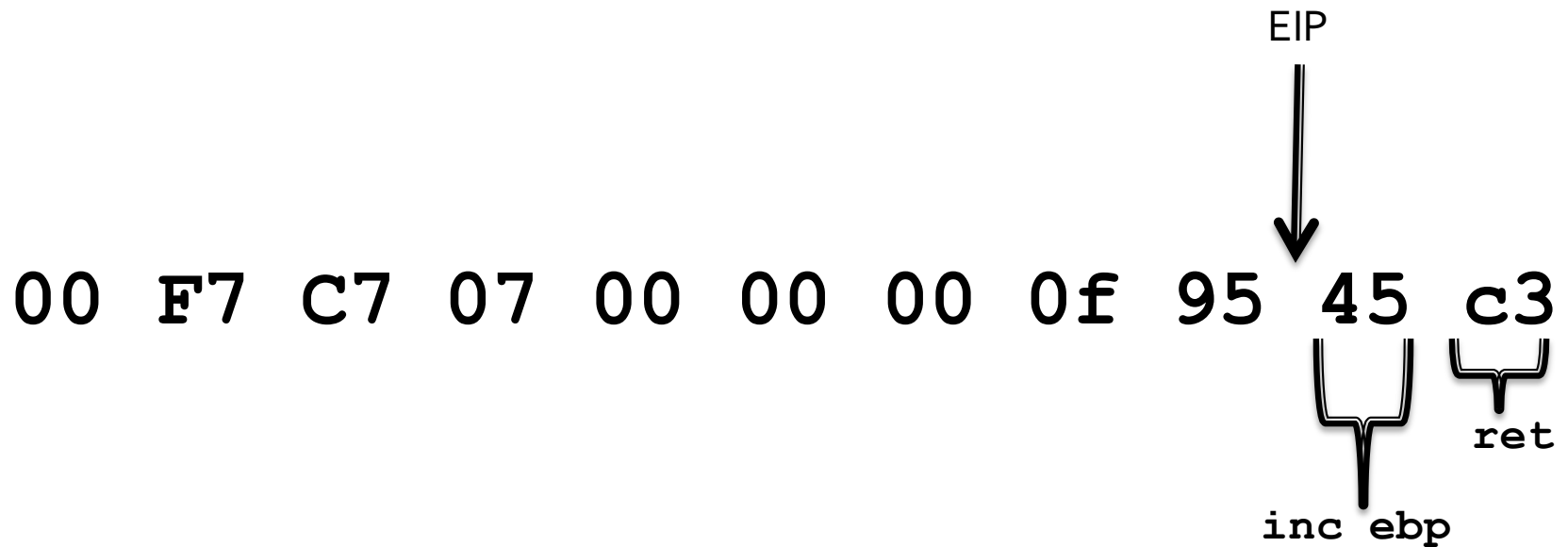
Full Gadget:

ROP

EIP
↓
00 F7 C7 07 00 00 00 0f 95 45 c3
└─┘
ret

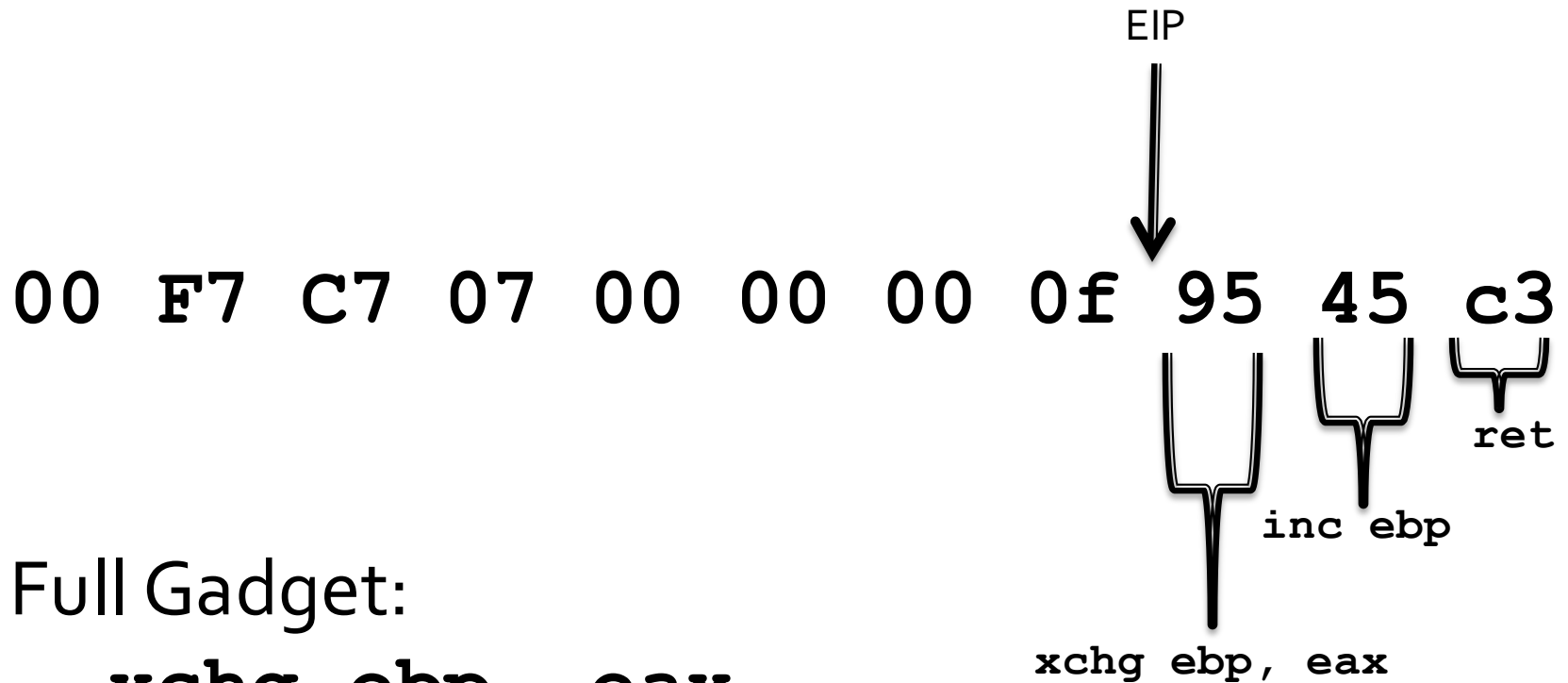
Full Gadget:
ret

ROP



Full Gadget:
`inc ebp`
`ret`

ROP



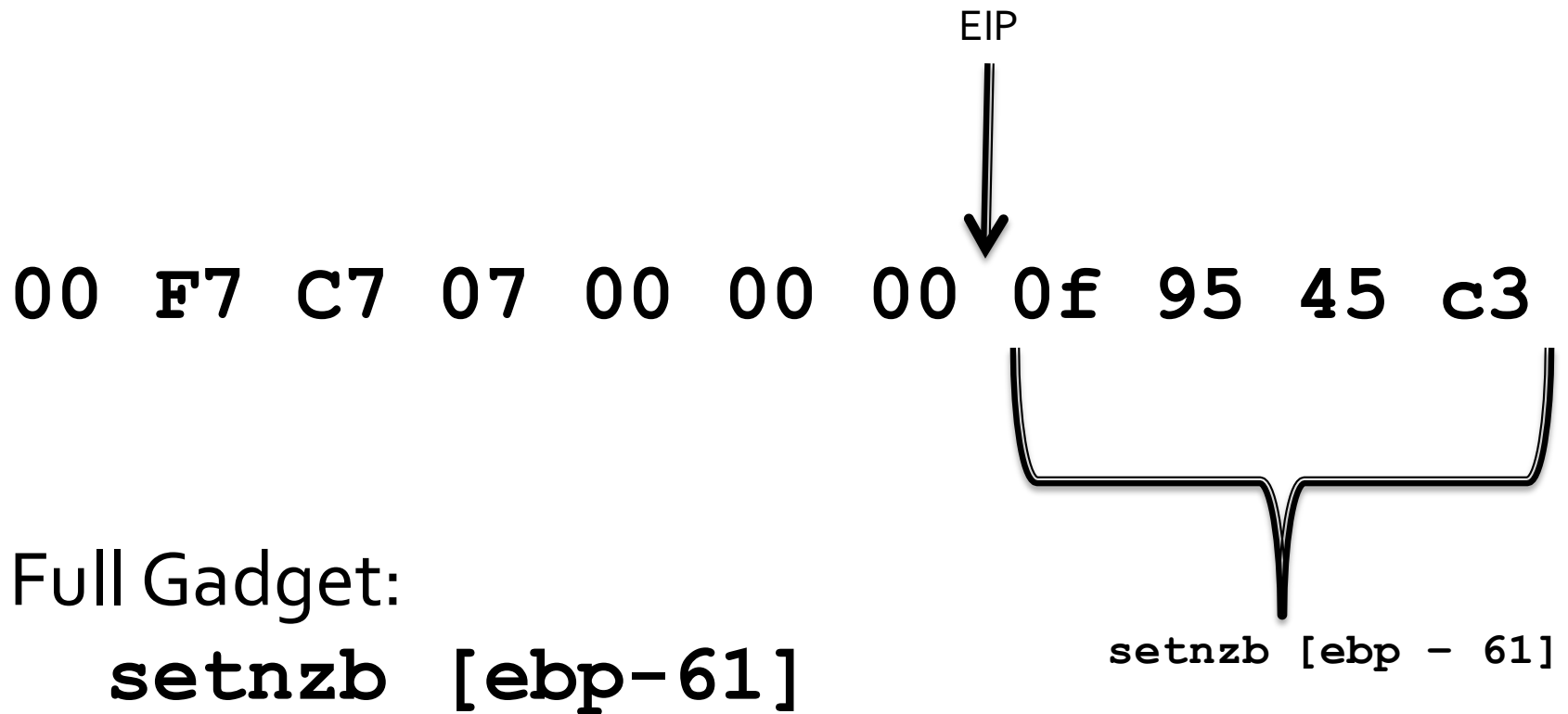
Full Gadget:

`xchg ebp, eax`

`inc ebp`

`ret`

ROP



<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>

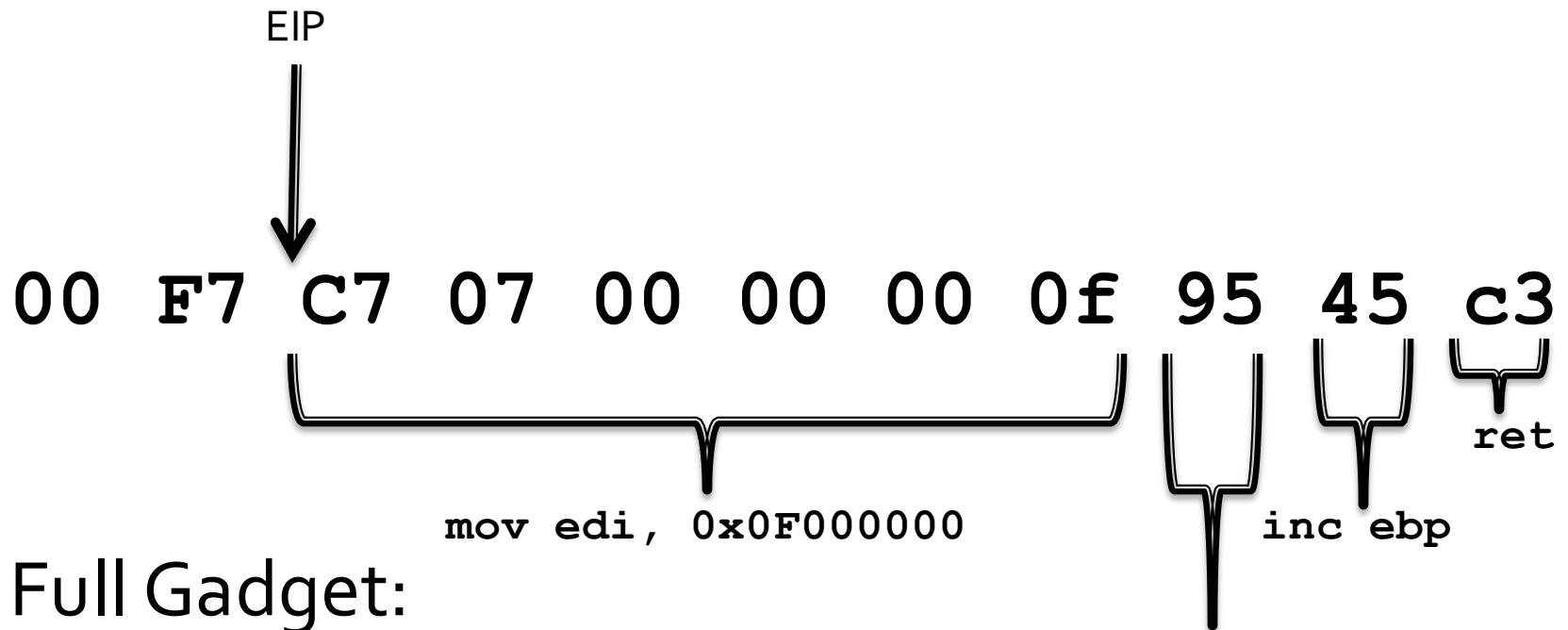
ROP

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

Full Gadget:

<none - invalid instruction>

ROP



Full Gadget:

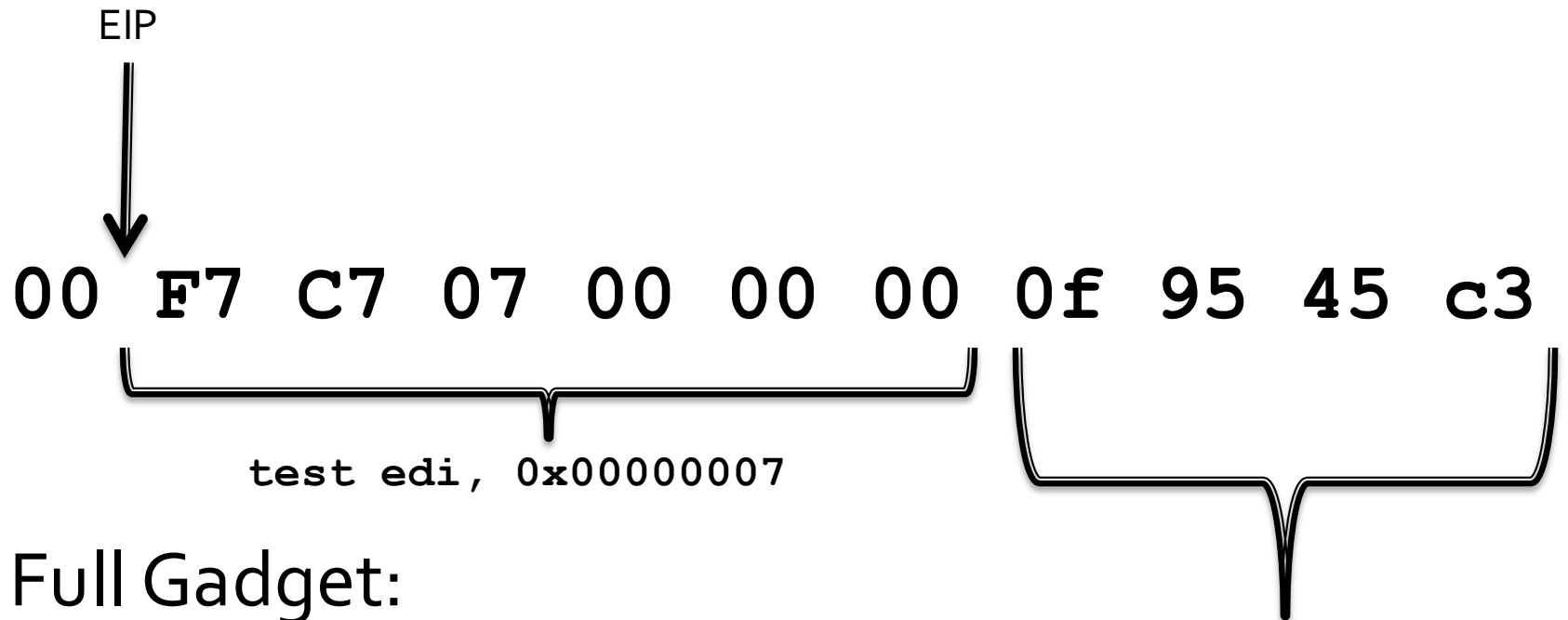
mov edi, 0x0F000000

xchg ebp, eax

inc ebp

ret

ROP



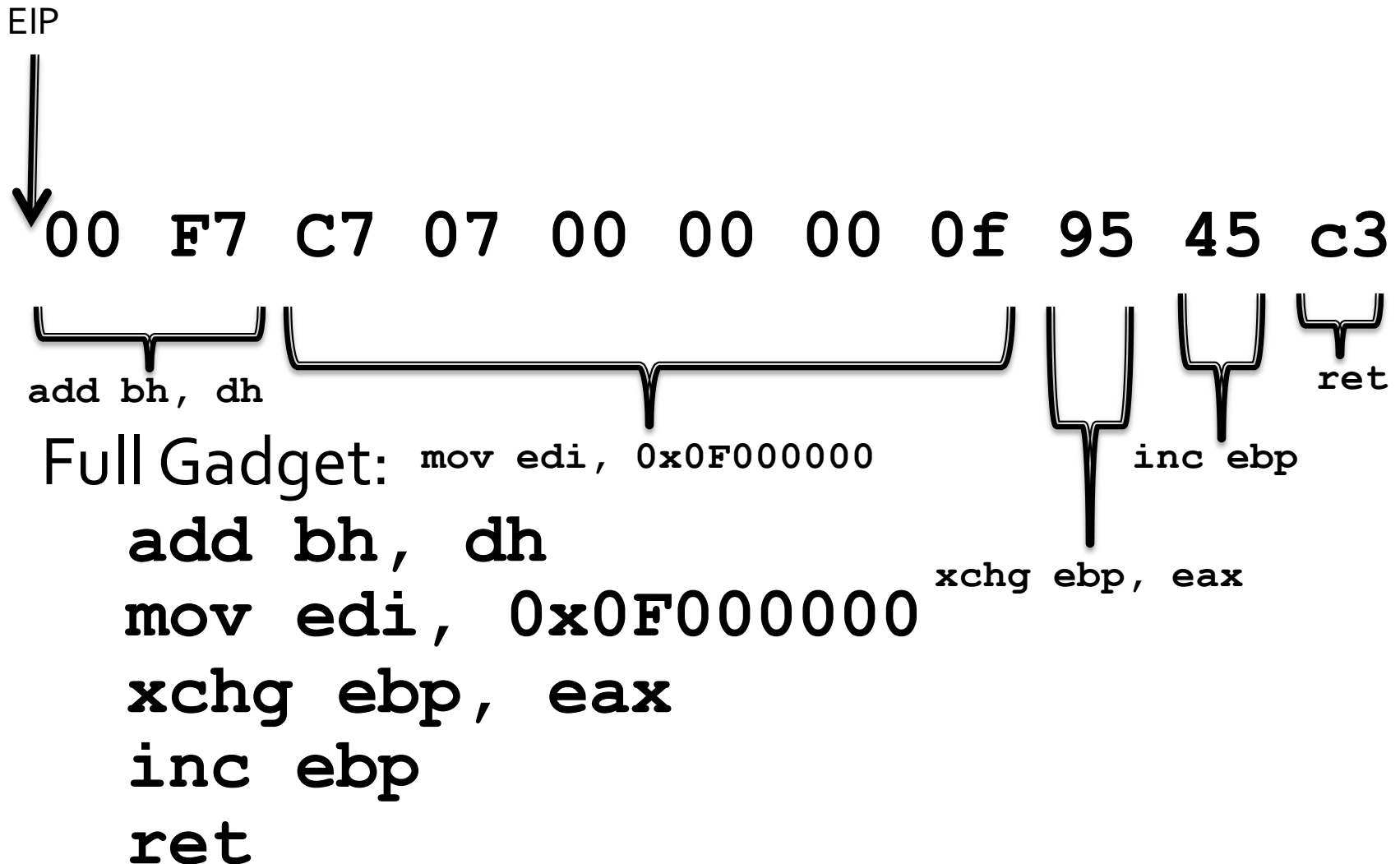
Full Gadget:

test edi, 0x00000007

setnzb [ebp-61]

<no return>

ROP



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
```

buffer

saved FP

ret

arg

arg

local var

prev FP

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*

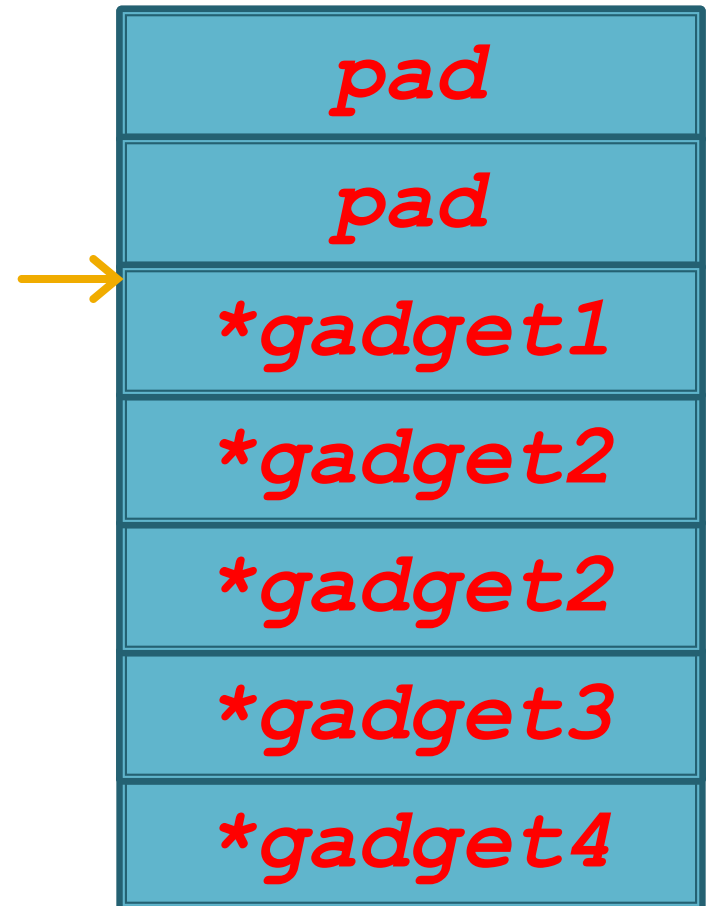
**gadget3*

**gadget4*

ROP-Chains

ROP Chain:

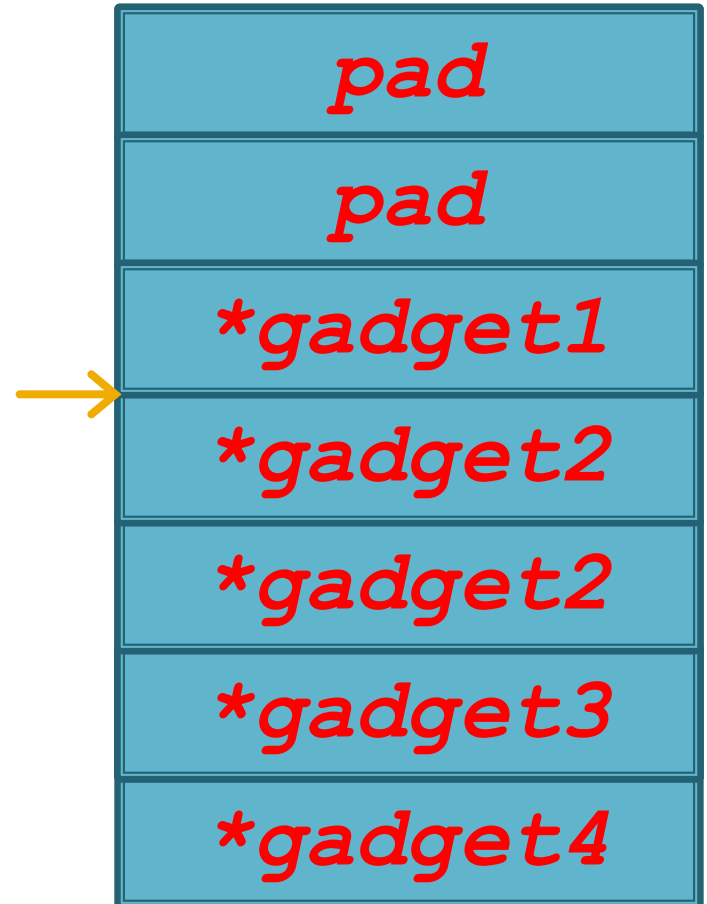
```
mov  eax, 0x10
add  eax, ebp
add  eax, ebp
mov  [eax+8], eax
mov  ebp, esp
ret
```



ROP-Chains

ROP Chain:

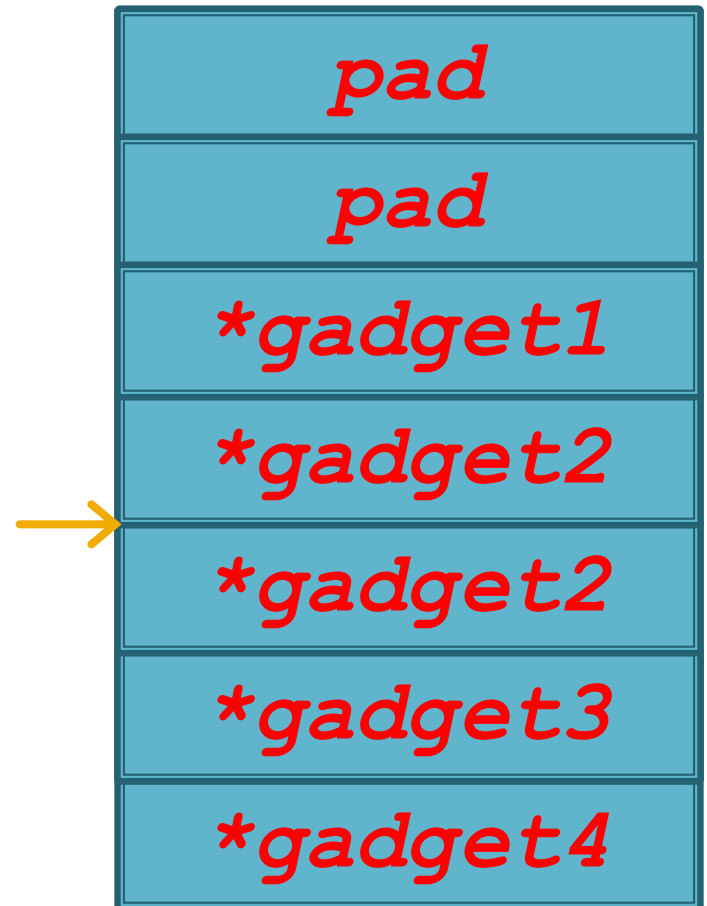
```
mov eax, 0x10  
add  eax, ebp  
add  eax, ebp  
mov [eax+8], eax  
mov ebp, esp  
ret
```



ROP-Chains

ROP Chain:

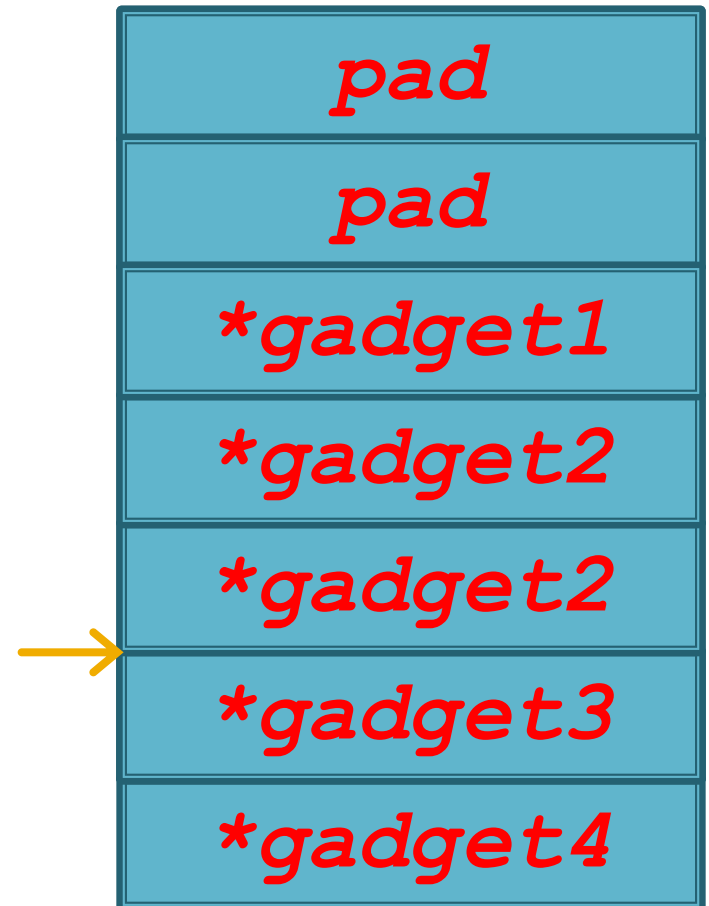
```
mov  eax, 0x10
add  eax, ebp
add  eax, ebp
mov  [eax+8], eax
mov  ebp, esp
ret
```



ROP-Chains

ROP Chain:

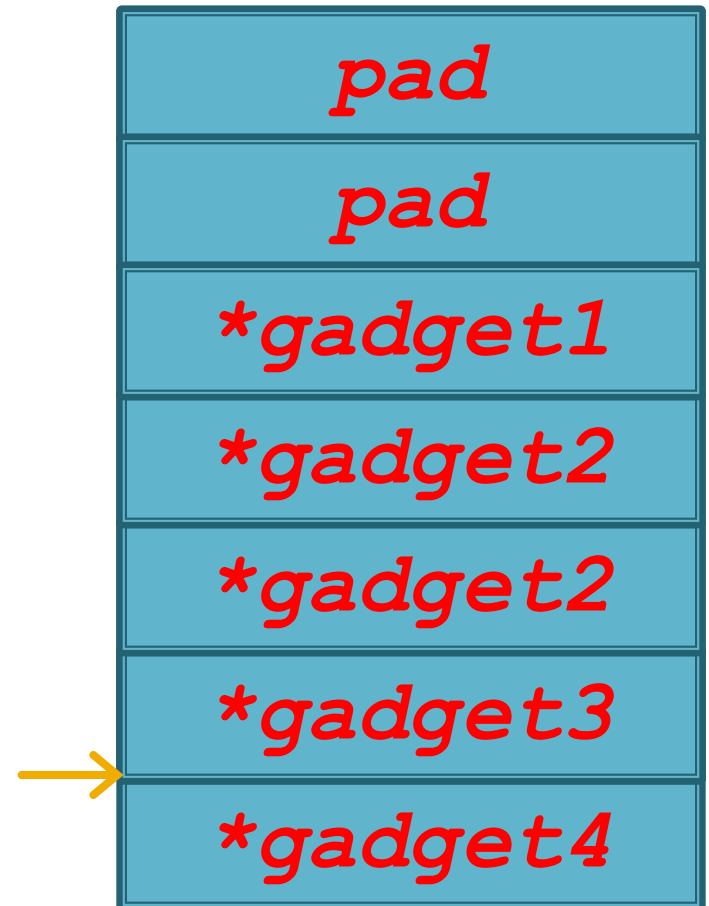
```
mov  eax, 0x10
add  eax, ebp
add  eax, ebp
mov  [eax+8], eax
mov  ebp, esp
ret
```



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

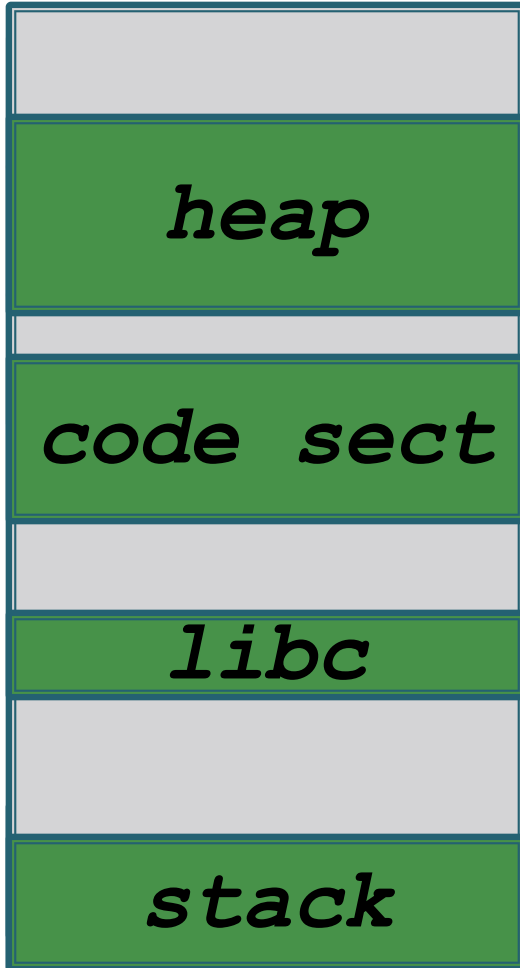
Make it extremely hard to predict references

Requires many changes to compilation and/or loading
Code must be “relocatable” or “position independent”

<Details are out-of-scope>

Memory Layout (no ASLR)

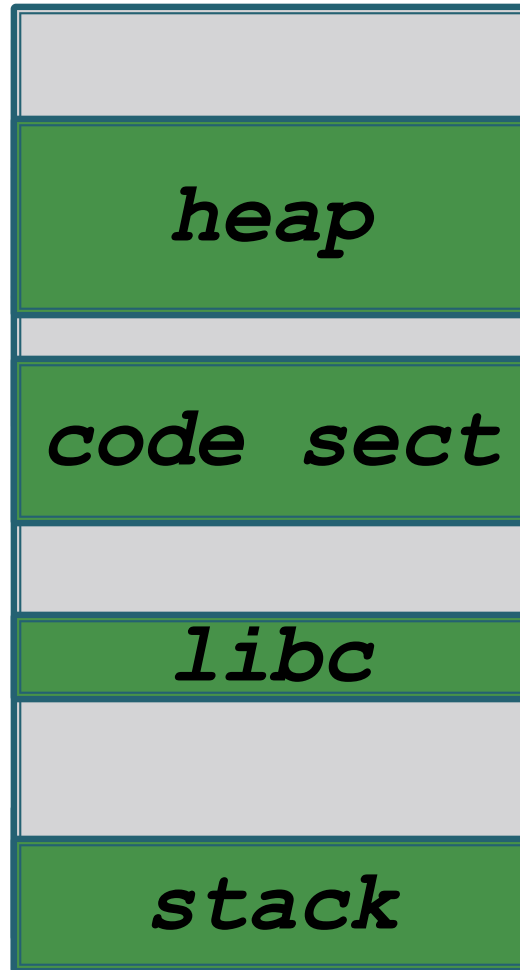
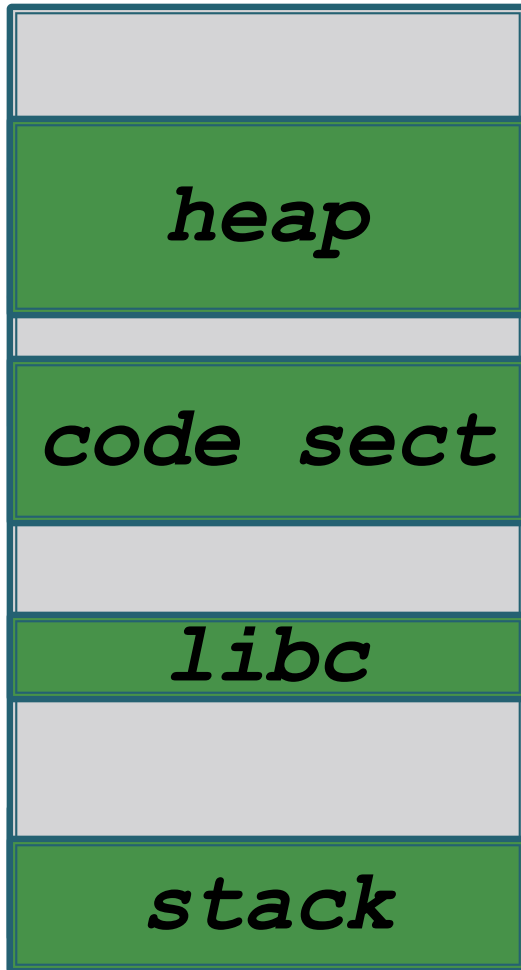
0x000000



0xFFFFFFFF

Memory Layout (no ASLR)

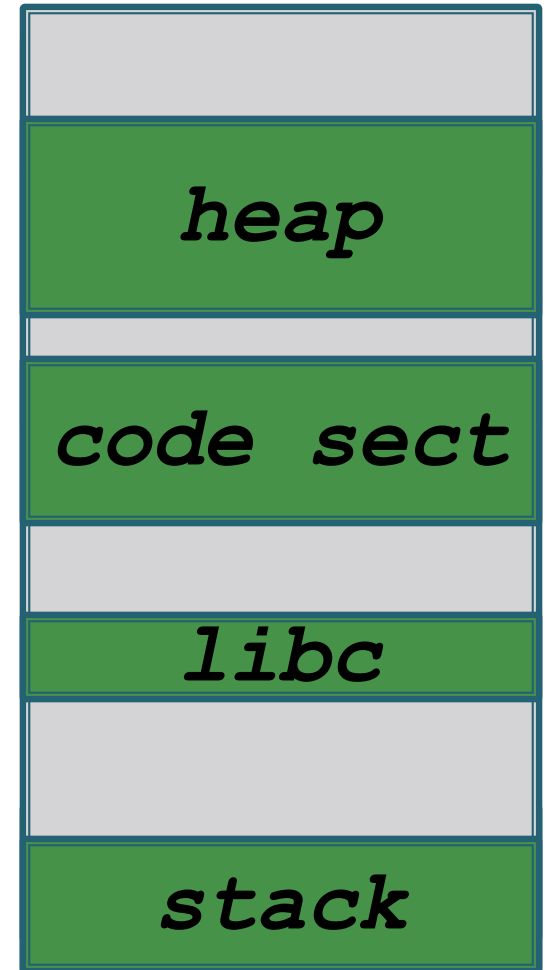
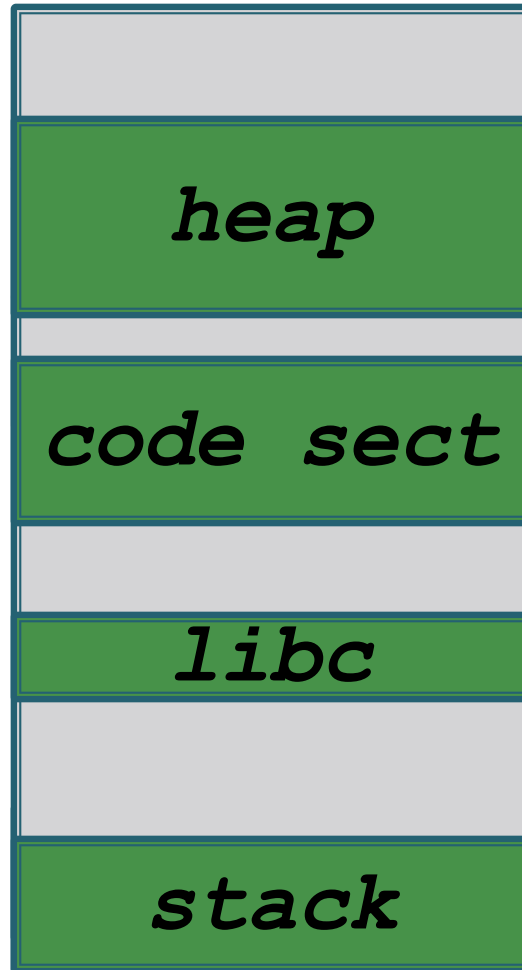
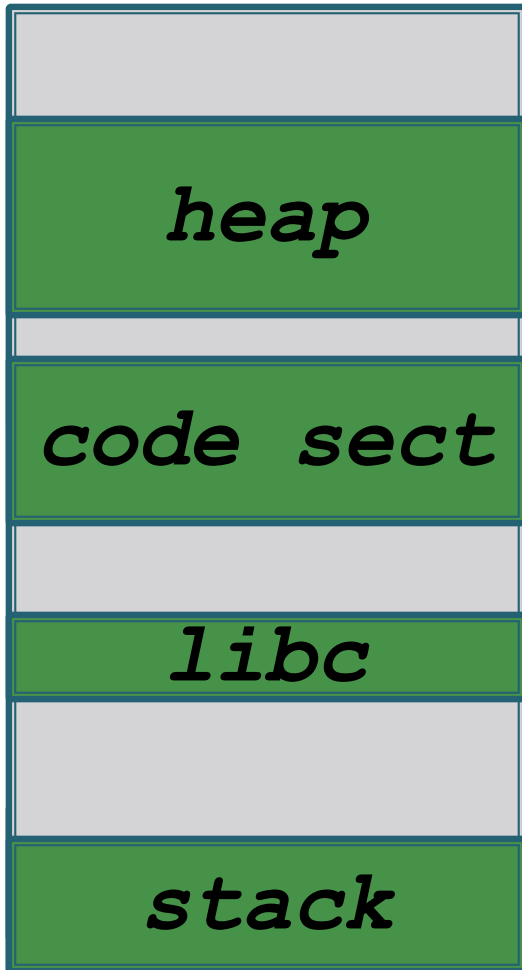
0x000000



0xFFFFFFFF

Memory Layout (no ASLR)

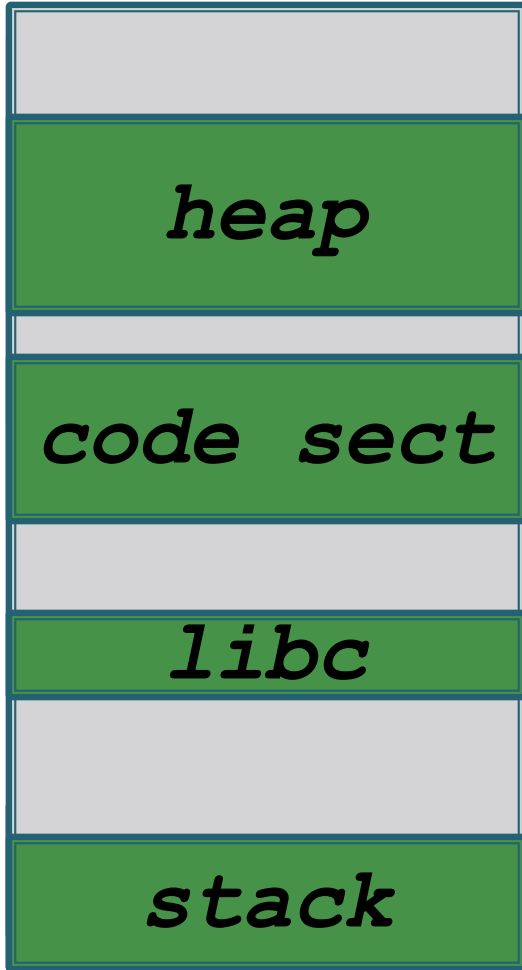
0x000000



0xFFFFFFFF

Memory Layout (with ASLR)

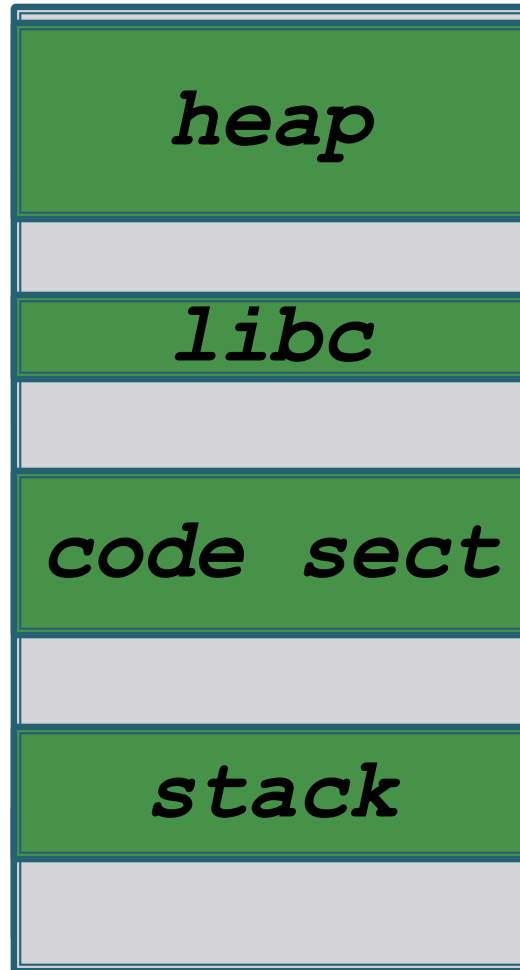
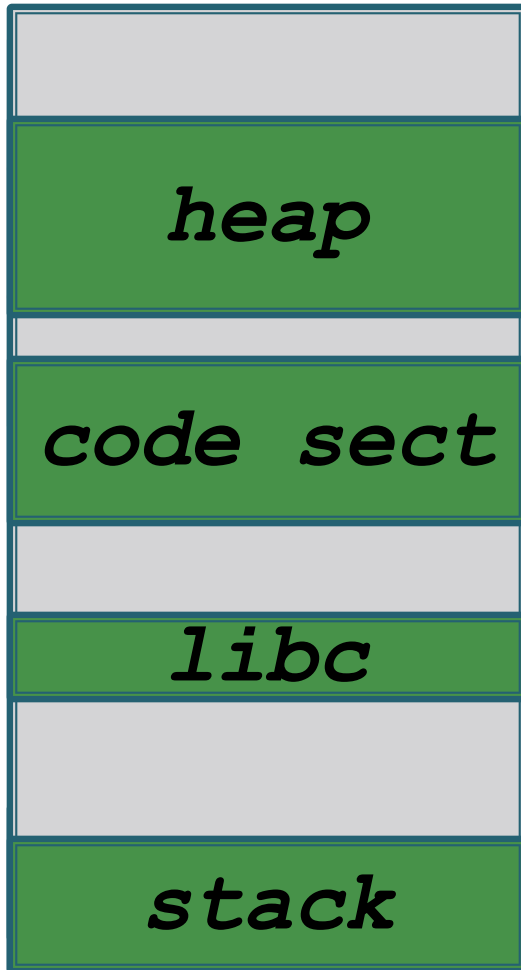
0x000000



0xFFFFFFFF

Memory Layout (with ASLR)

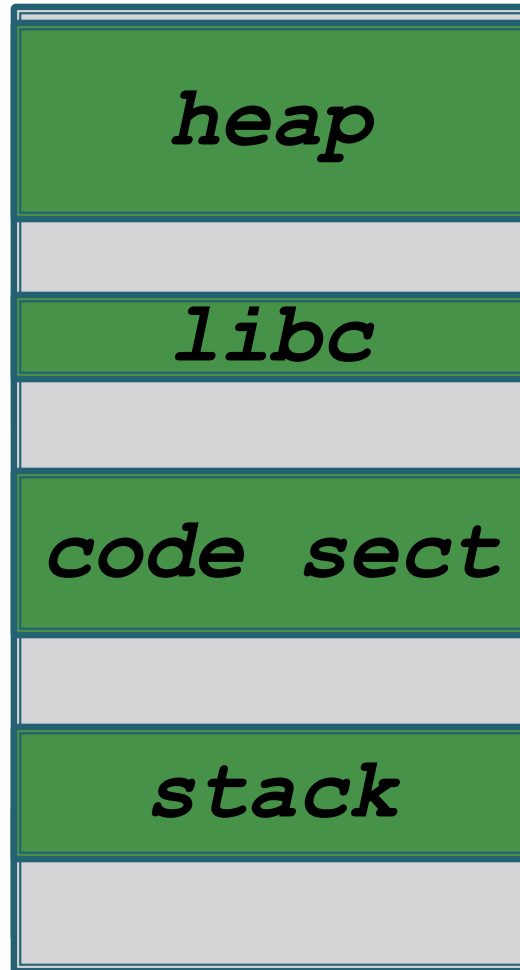
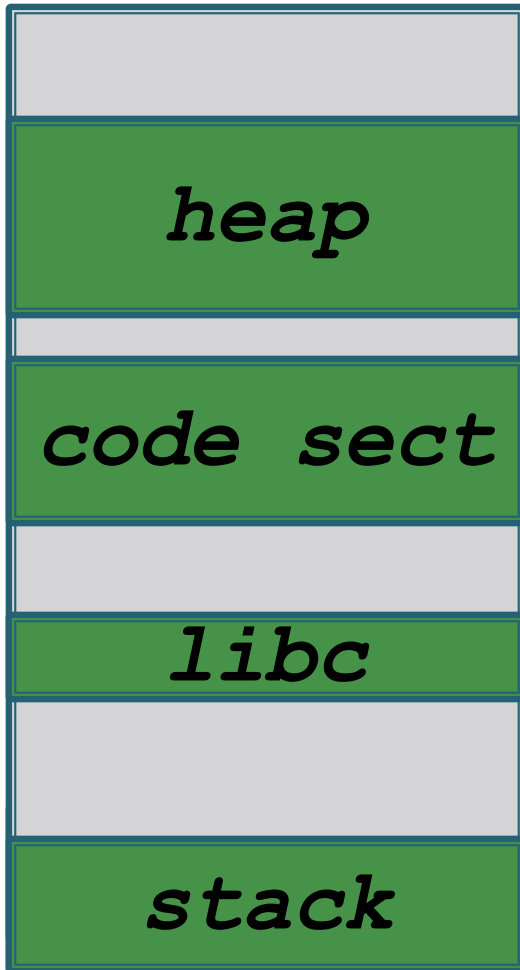
0x000000



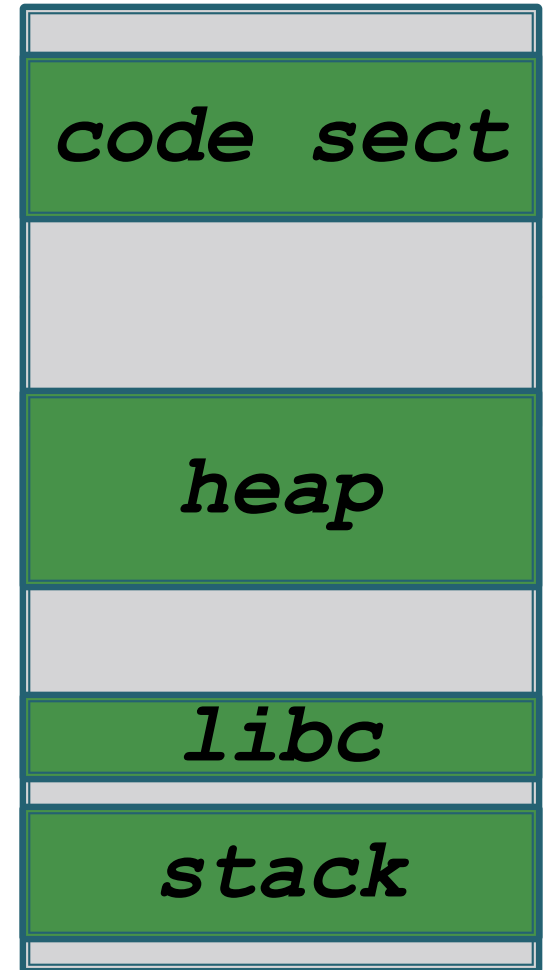
0xFFFFFFFF

Memory Layout (with ASLR)

0x000000



0xFFFFFFFF



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.

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

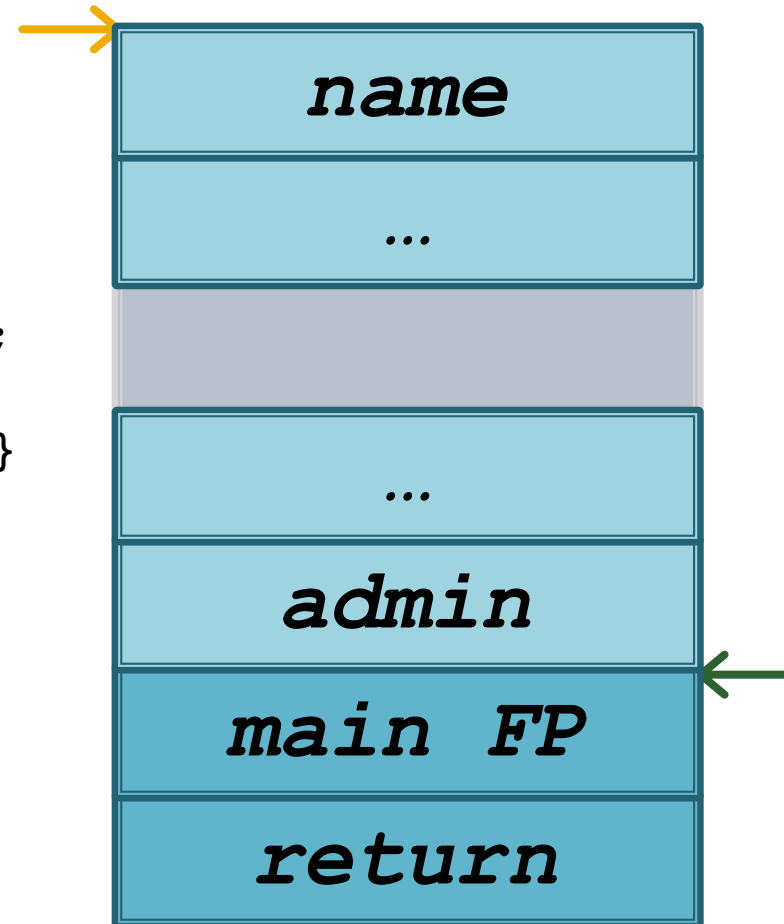
```
int delete_account(char* username,
    int length, VOID* creds) {
    int admin;
    char name[100];
    admin = check_admin(creds);
    strncpy(name, username, length);
    canonicalize_username(name);
    if (admin) {delete_user(name);}
    return (admin > 0);
}
```


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);
}
```

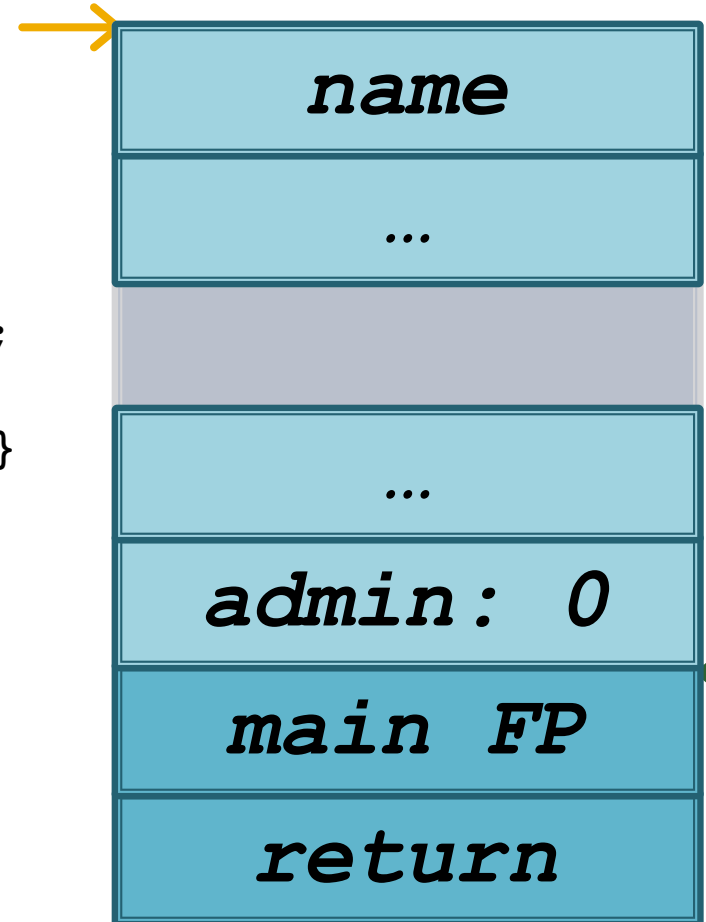
Data-only attacks

```
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);  
}
```



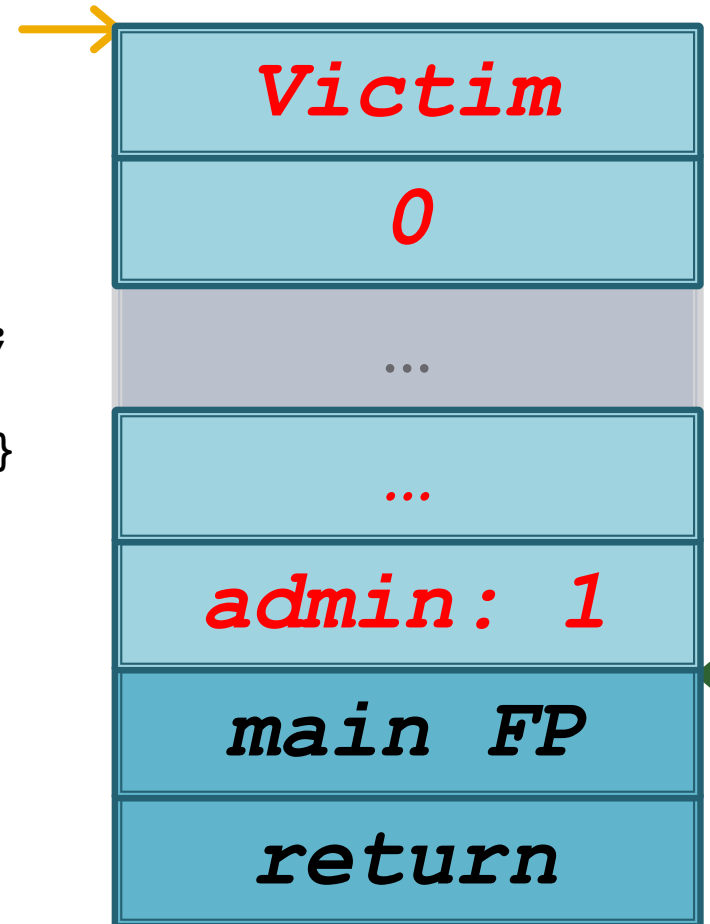
Data-only attacks

```
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);  
}
```



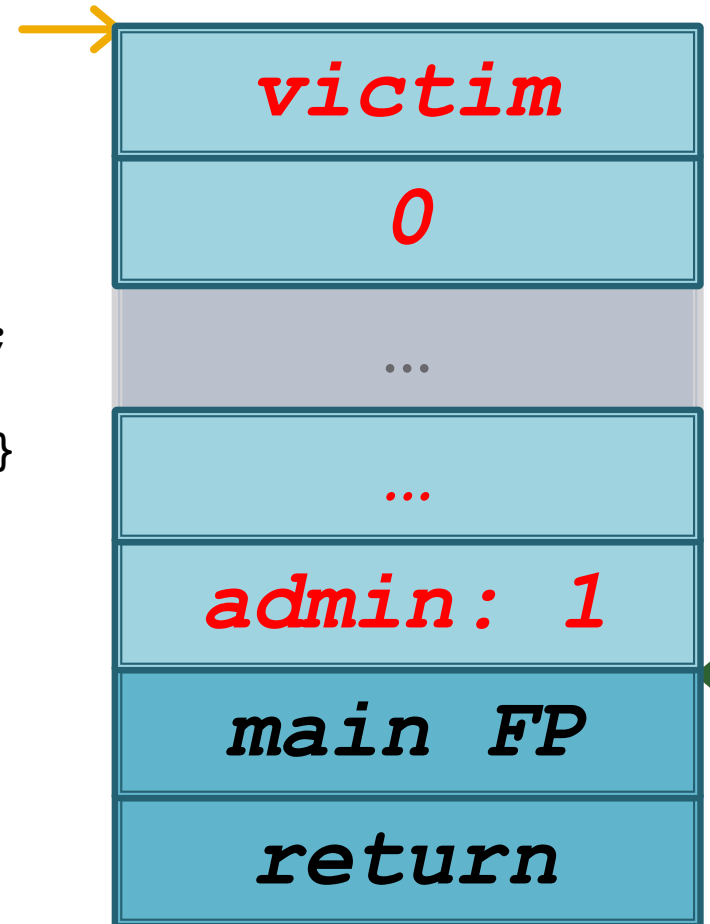
Data-only attacks

```
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);  
}
```



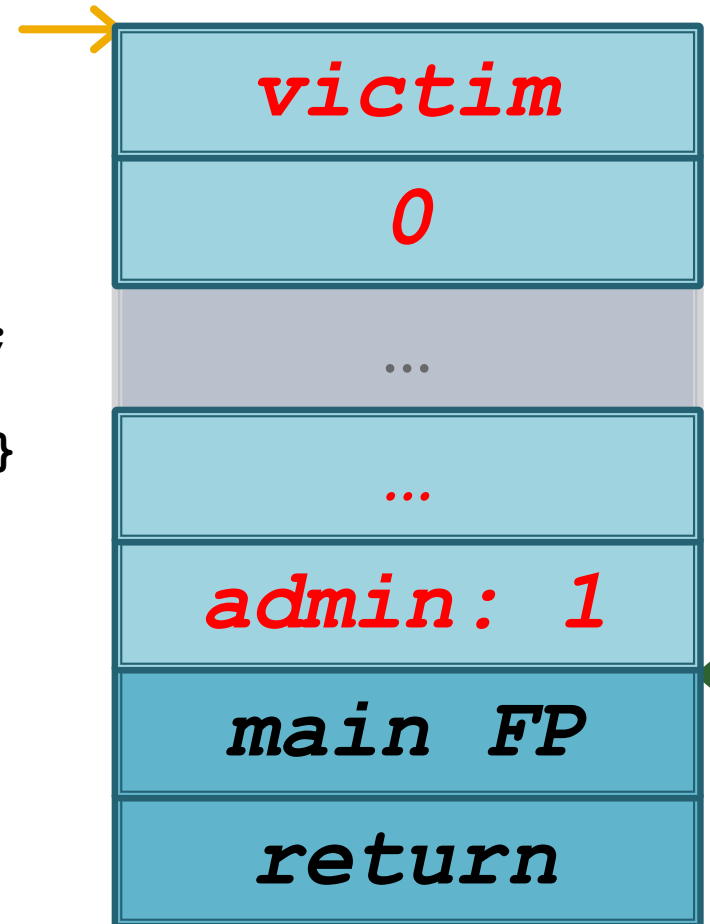
Data-only attacks

```
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

1. Buffer A is allocated
2. Pointers #1 and #2 are created
3. Buffer A is filled with data
4. Buffer is free'd via pointer #1
 - #2 still points to the buffer's previous location
5. Buffer B is allocated and its memory overlaps with Buffer A's previous allocation
6. Pointer #2 is dereferenced with the expectation that it still points to Buffer A

SEH Exploitation

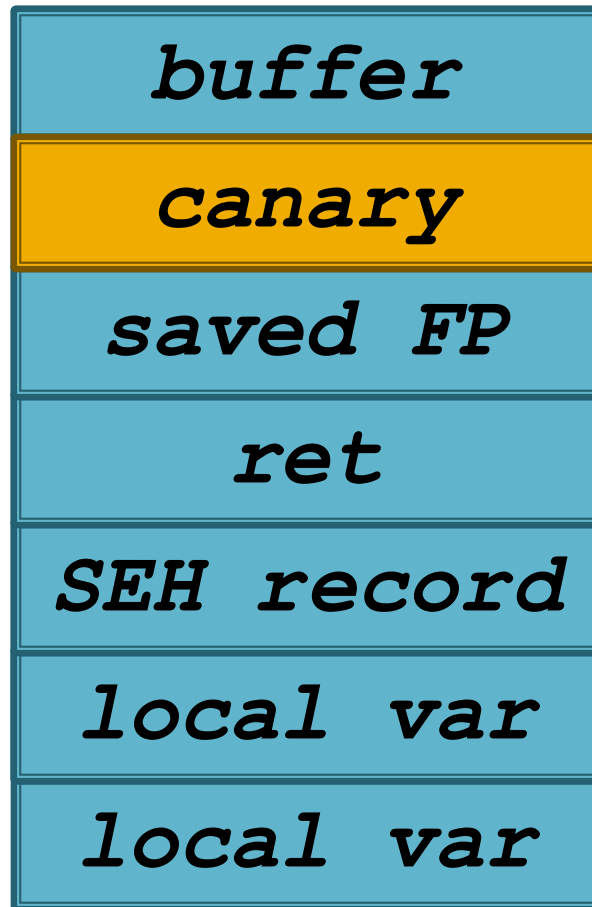
Structured Exception Handling

Redirect control flow via the exception handler 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