

EECS 388



Introduction to Computer Security

Lecture 15:

Control Hijacking (Part 1)

October 24, 2023
Marshall Stone



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)

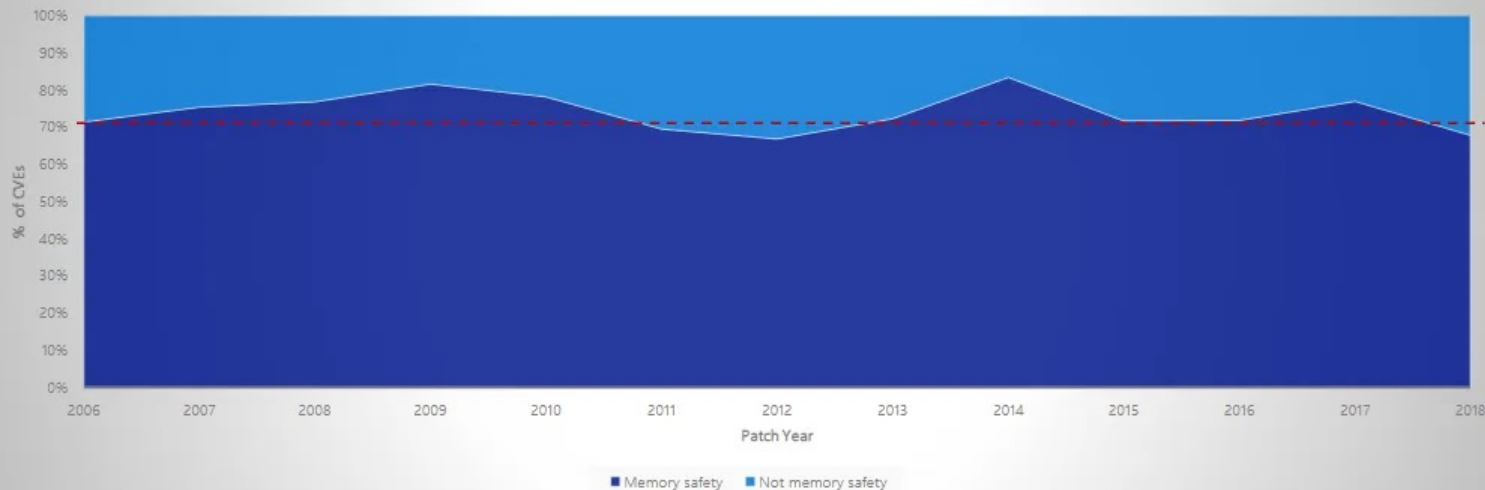
Also known as: remote/arbitrary code execution, pwning

Control Hijacking



We closely study the root cause trends of vulnerabilities & search for patterns

% of memory safety vs. non-memory safety CVEs by patch year





Forbes

Subscribe

Sign In



Nov 21, 2021, 06:00am EST | 5,105 views

Windows 10 Zero-Click Security Exploit Wanted. Reward: \$3 Million



Davey Winder

Senior Contributor

Cybersecurity

Co-founder, Straight Talking Cyber

Follow



Listen to article 4 minutes









0:00 / 19:54



Outline



Bits and pieces

- Memory: Address space

- CPU: Registers and Instructions

- Disassembly

Stacking things up

- Stack frames

- Stack in assembly

Blowing things up

- Buffer overflows

- Shellcode

- Simple defense (DEP)

Outline



Bits and pieces

- Memory: Address space

- CPU: Registers and Instructions

- Disassembly

Stacking things up

- Stack frames

- Stack in assembly

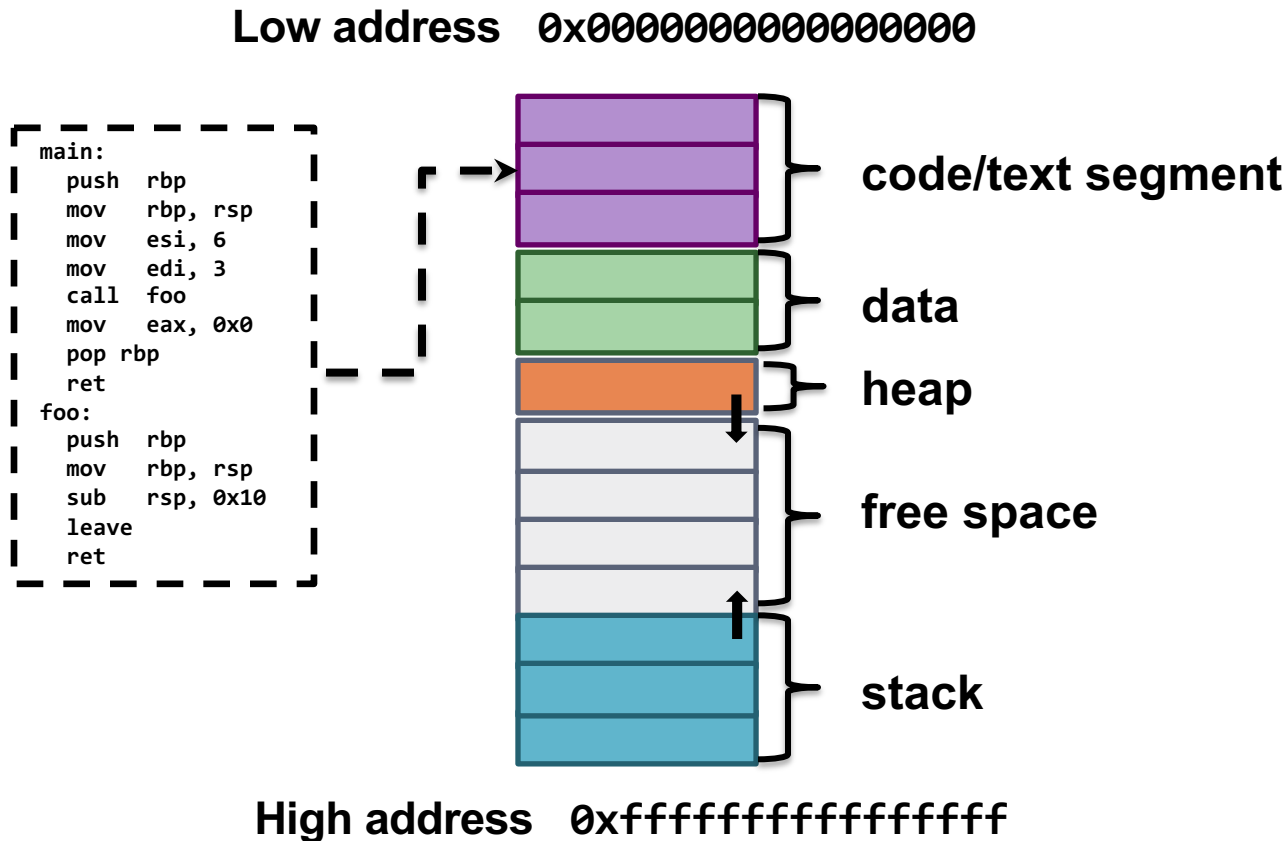
Blowing things up

- Buffer overflows

- Shellcode

- Simple defense (DEP)

Memory Organization



Assembly Language

x86_64 (x64, AMD64), x86, Arm, RISC-V, MIPS, PowerPC

General Purpose Registers

RAX: typically used to store return values

RBX, RCX, RDX, RDI, RSI

Special Purpose Registers

RIP: Instruction Pointer

→ RSP: Stack Pointer

RBP: Frame/Base Pointer ←


CPU: Instructions



Move a value to a register

```
mov rax, 0x34
```

This is x86_64 *Intel*
syntax. Be careful when
Googling!

A black arrow pointing from the explanatory text to the instruction 'mov rax, 0x34'.

Add a value to a register

```
add rax, 10
```

Change execution path

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

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

(Dis)assembly



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

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



```
$ gcc example.c -o example
```



		undefined foo()
00101129	55	PUSH RBP
0010112a	48 89 e5	MOV RBP, RSP
0010112d	48 83 ec	SUB RSP, 0x10
	10	
00101131	90	NOP
00101132	5d	POP RBP
00101133	c3	RET
		undefined main()
00101136	55	PUSH RBP
00101137	48 49 e5	MOV RBP, RSP
0010113a	be 06	MOV RSI, 0x6
0010113f	6a 03	MOV RDI, 0x3
	00 00 00	
00101144	e8 e0 ff ff	CALL foo
00101149	b8 00 00 00	MOV RAX, 0x0
0010114e	5d	POP RBP
0010114f	c3	RET



Outline



Bits and pieces

- Memory: Address space

- CPU: Registers and Instructions

- Disassembly

Stacking things up

- Stack frames

- Stack in assembly

Blowing things up

- Buffer overflows

- Shellcode

- Simple defense (DEP)

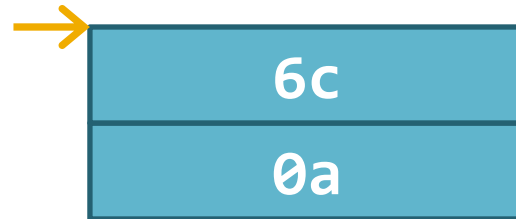
```
Stack myStack;  
myStack.push(1);  
myStack.push(2);  
myStack.pop();    // returns 2
```



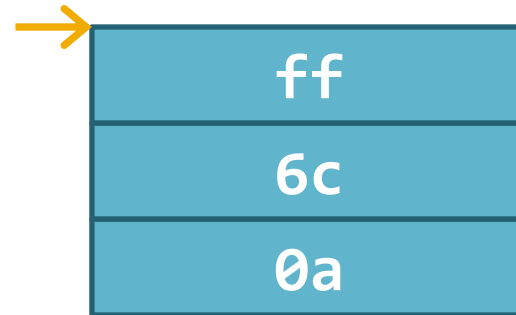
push 0x0a



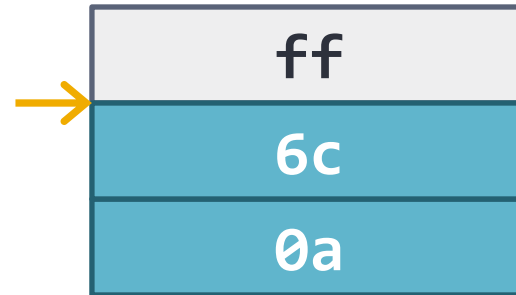
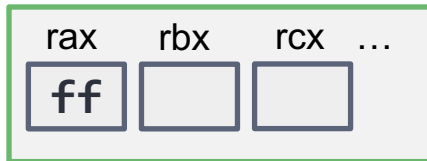
```
push 0x0a  
push 0x6c
```



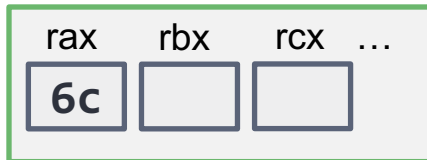
```
push 0x0a  
push 0x6c  
push 0xff
```



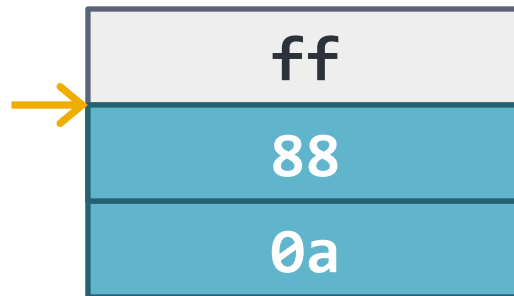
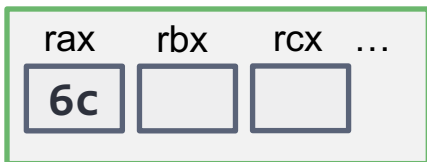

```
push 0x0a  
push 0x6c  
push 0xff  
pop  rax
```



```
push 0x0a  
push 0x6c  
push 0xff  
pop  rax  
pop  rax
```



```
push 0x0a  
push 0x6c  
push 0xff  
pop  rax  
pop  rax  
push 0x88
```



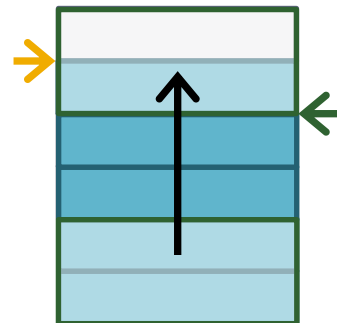
Stack frames



Starts at 0xfffffffffffffffffffff

Grows toward 0x0000000000000000
(x86 specific)

Low address
0x00000000



High address
0xffffffff

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

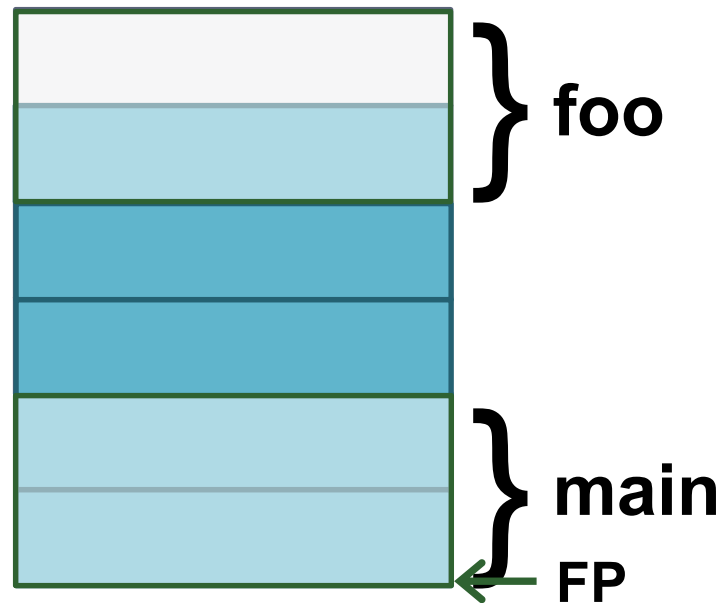
```
int main() {  
    foo(3,6);  
}
```

Stack frames



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

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

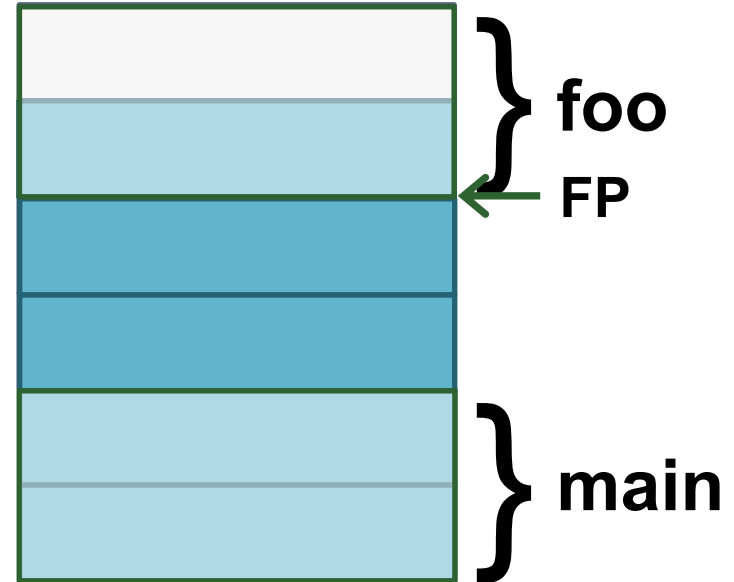


Stack frames



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

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



Stack frames



Starts at 0xfffffffff...

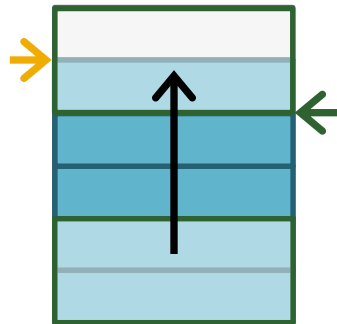
Grows toward 0x00000000...

x86 specific

Stack Pointer (RSP): →

Frame Pointer (RBP): ←

Low address
0x00000000...



High address
0xfffffffff...


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

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

Stack frames



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

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

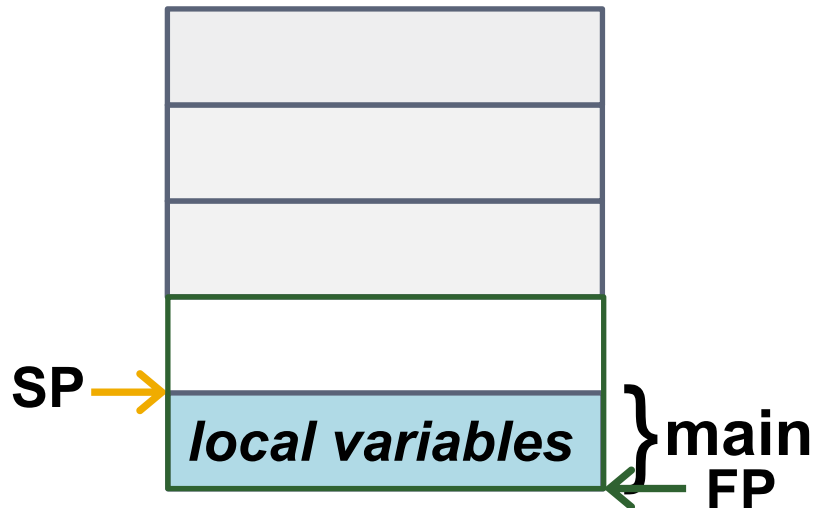


Stack frames



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

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

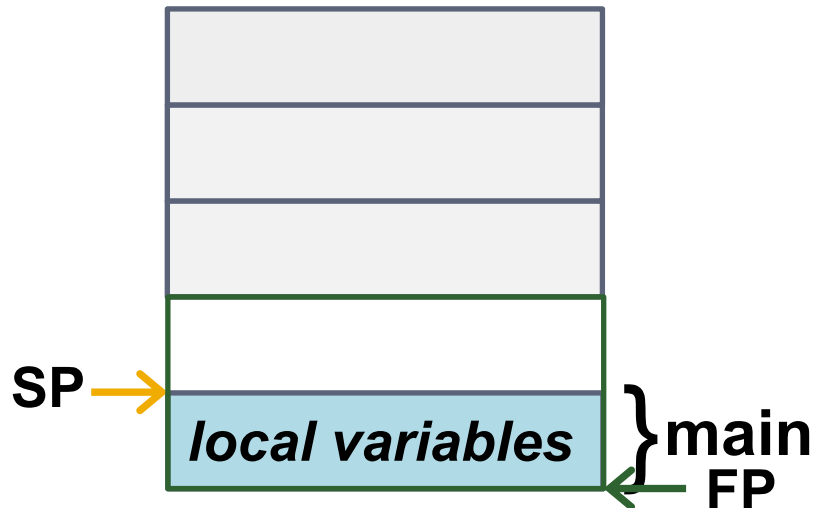


Stack frames



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

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

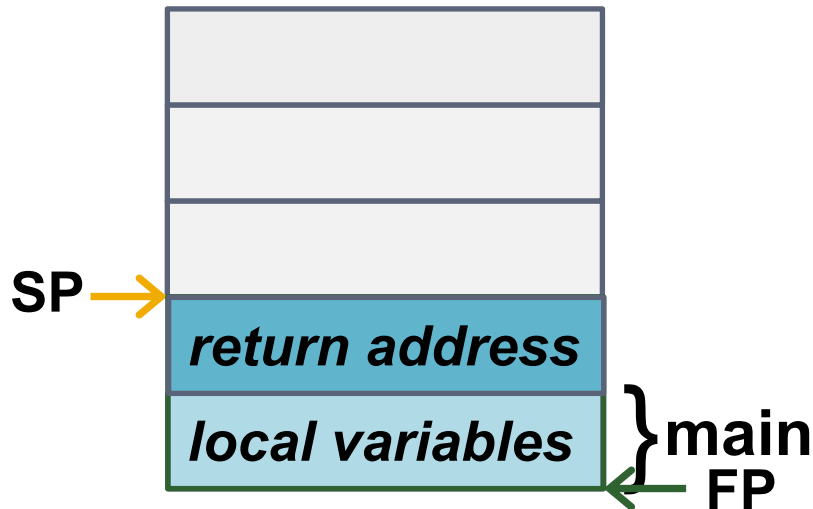


Stack frames



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

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

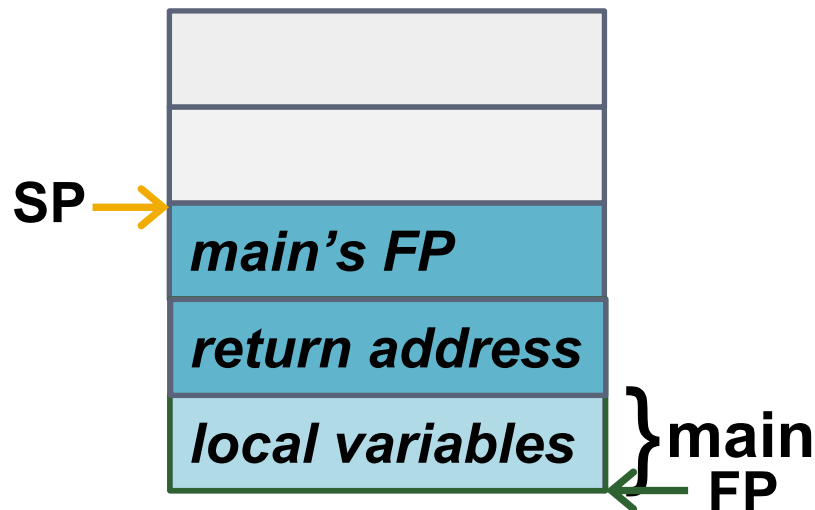


Stack frames



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

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

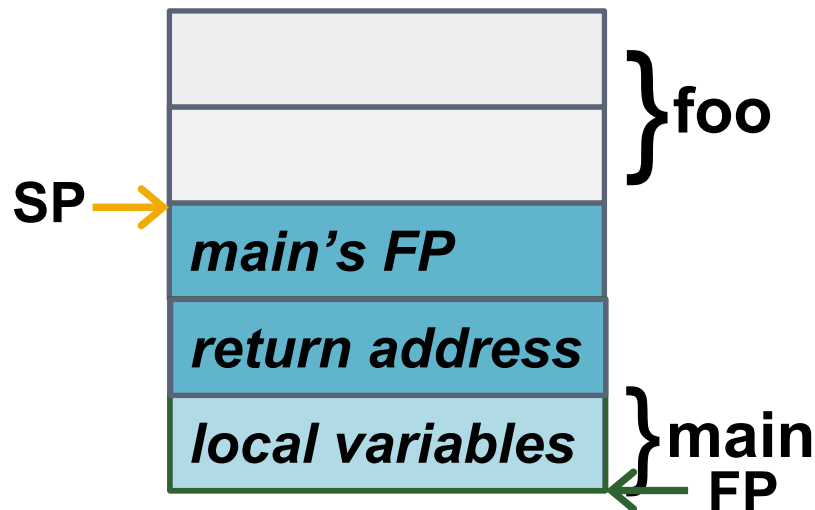
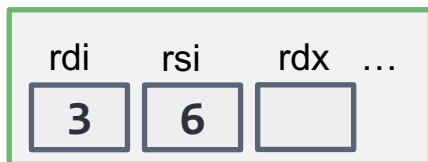


Stack frames



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

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

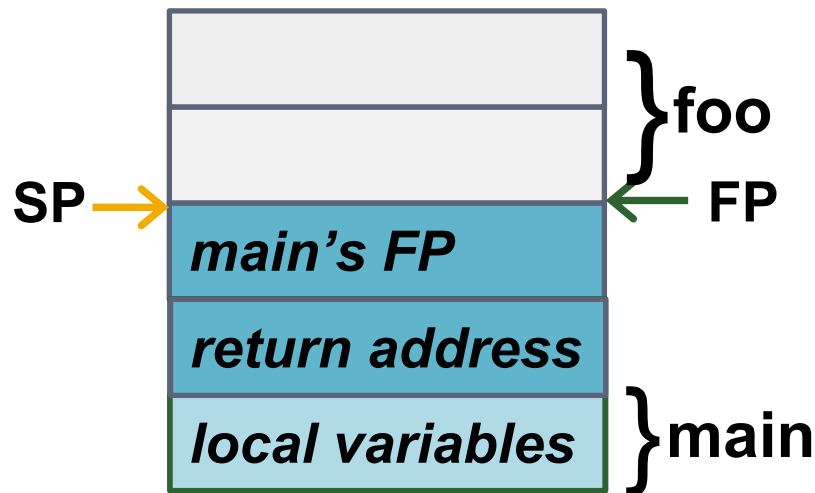
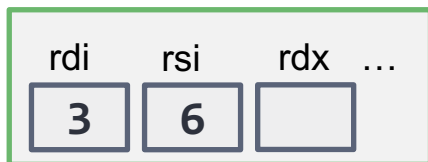


Stack frames



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

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

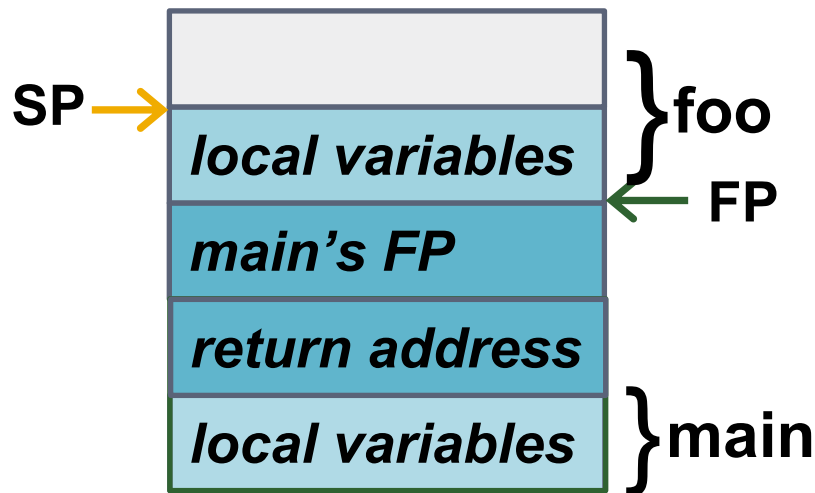


Stack frames



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

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



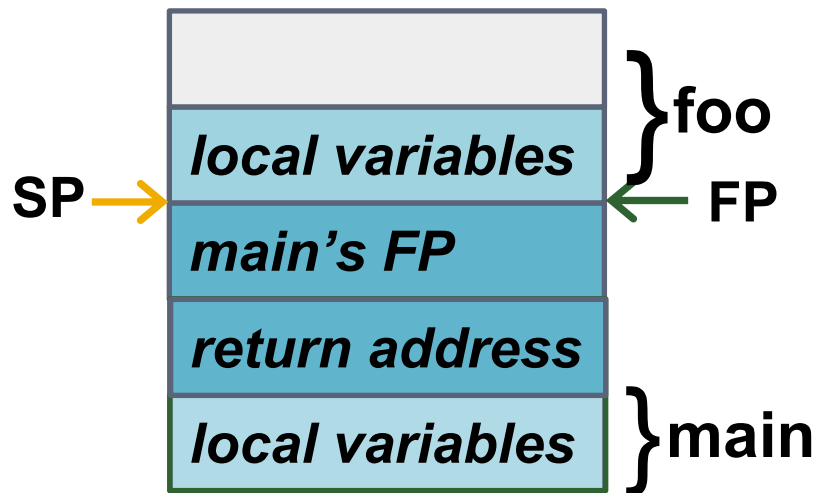
Stack frames



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

```
}  
}
```

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



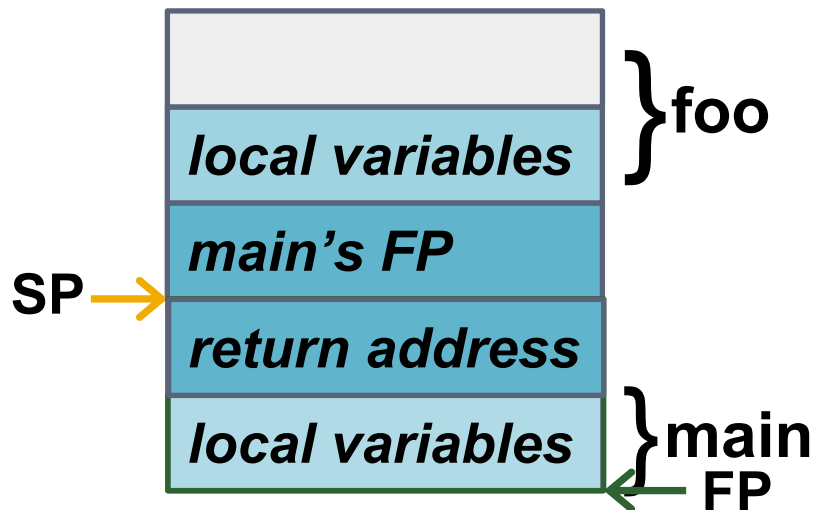
Stack frames



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

```
}  
}
```

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



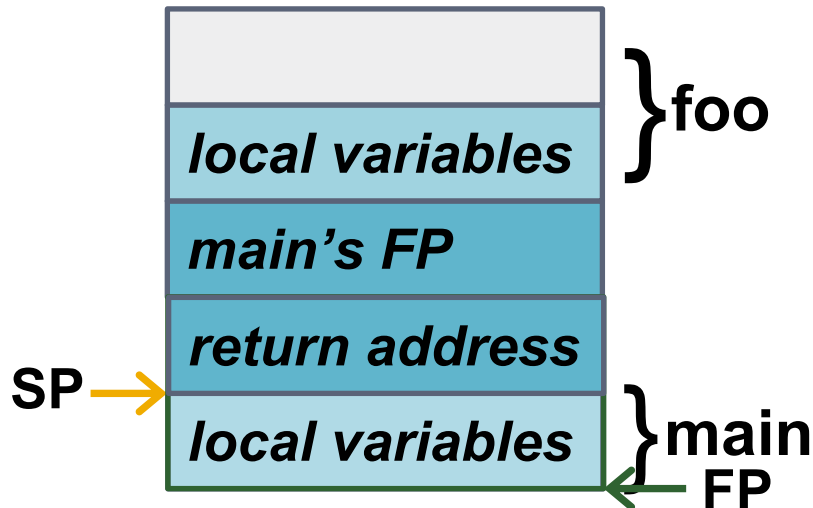
Stack frames



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

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

```
}
```



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

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

example.s (x64)



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

```
PUSH    RBP  
MOV     RBP,RSP  
MOV     RSI, 0x6  
MOV     RDI, 0x3  
CALL    foo  
MOV     RAX,0x0  
POP     RBP  
RET
```

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

```
PUSH    RBP  
MOV     RBP,RSP  
SUB     RSP,0x10  
NOP  
POP     RBP  
RET
```

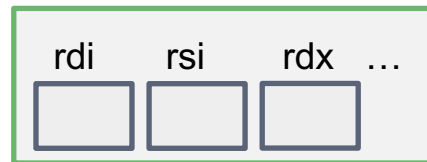
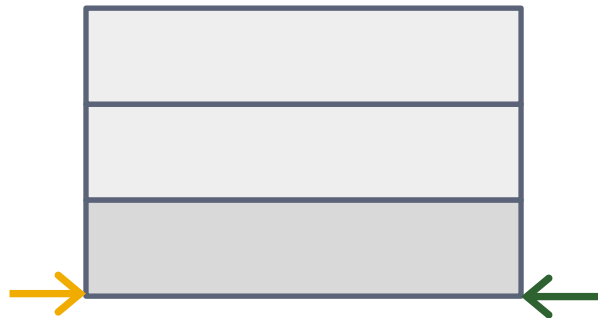
example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```

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



example.s (x64)



main:

push **rbp**

mov rbp, rsp

mov rsi, 0x6

mov rdi, 0x3

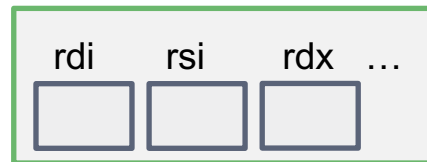
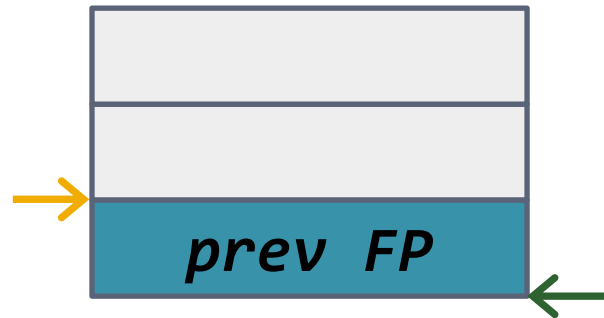
call foo

mov rax, 0x0

pop rbp

ret

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



example.s (x64)



main:

push rbp

mov rbp, rsp

mov rsi, 0x6

mov rdi, 0x3

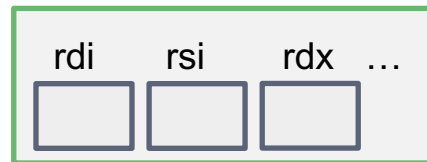
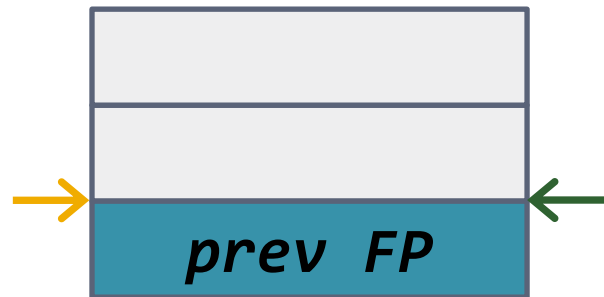
call foo

mov rax, 0x0

pop rbp

ret

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



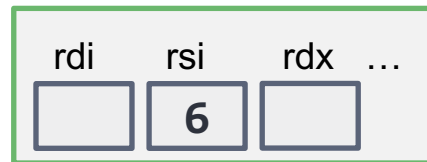
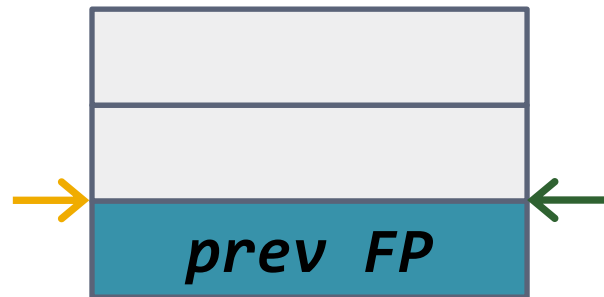
example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```

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



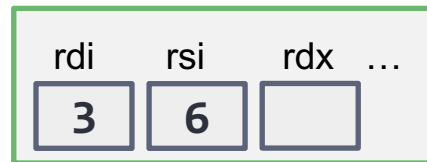
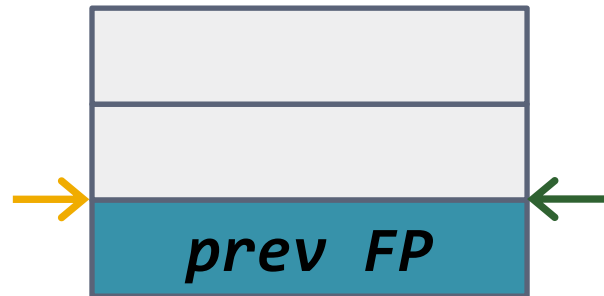
example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```

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



example.s (x64)

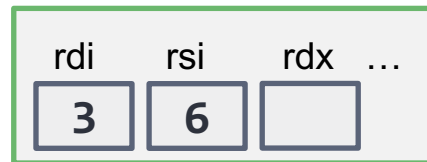
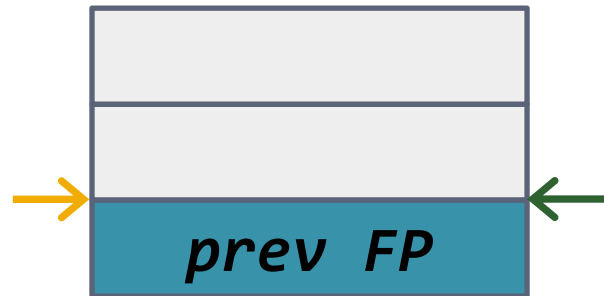


main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```

```
push    rip
jmp     foo
```

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



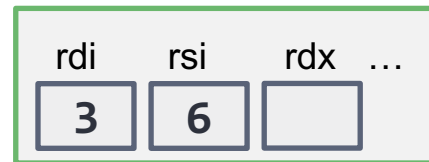
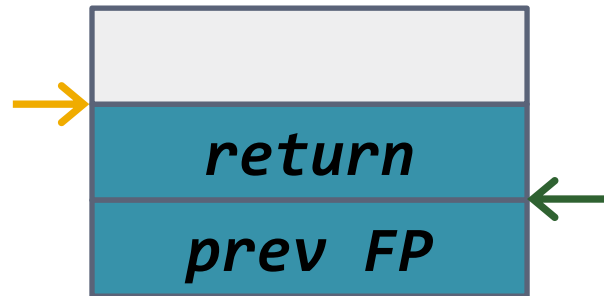
example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call   foo
mov     rax, 0x0
pop     rbp
ret
```

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



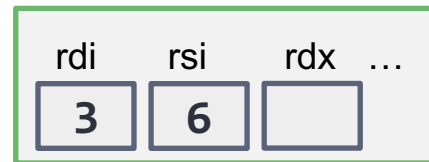
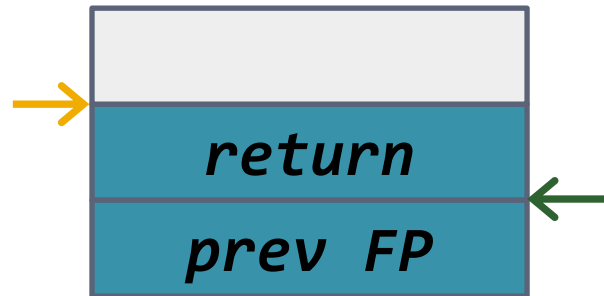
example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call   foo
mov     rax, 0x0
pop     rbp
ret
```

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



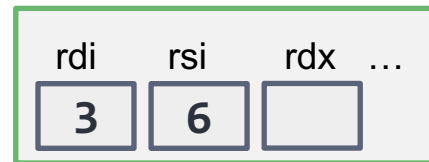
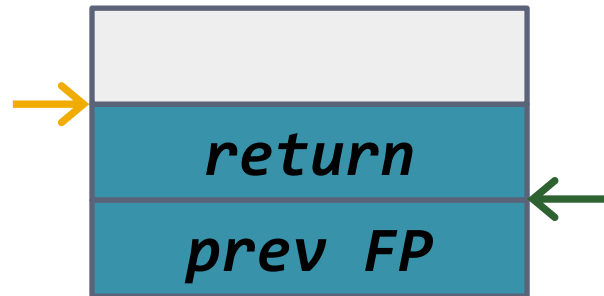
example.s (x64)



foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave
ret
```

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



foo:

push rbp

mov rbp, rsp

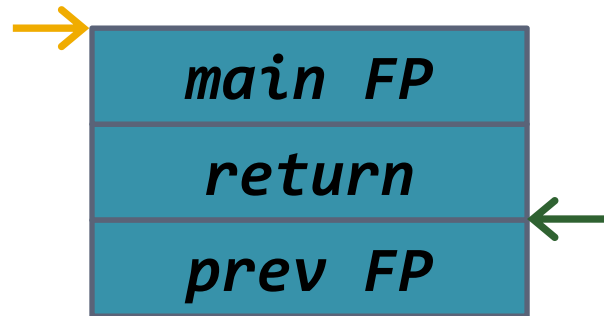
sub rsp, 0x10

nop

leave

ret

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



foo:

push rbp

mov rbp, rsp

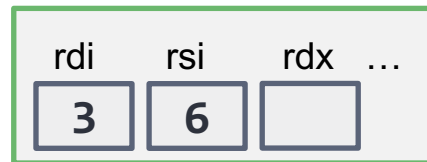
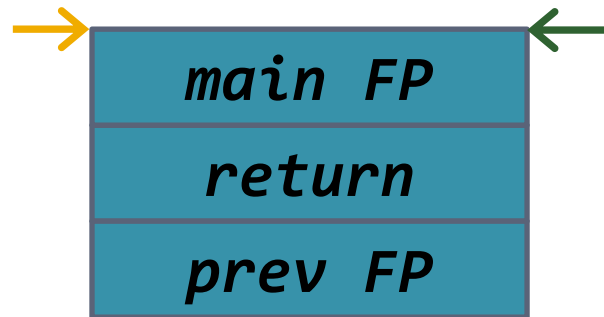
sub rsp, 0x10

nop

leave

ret

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



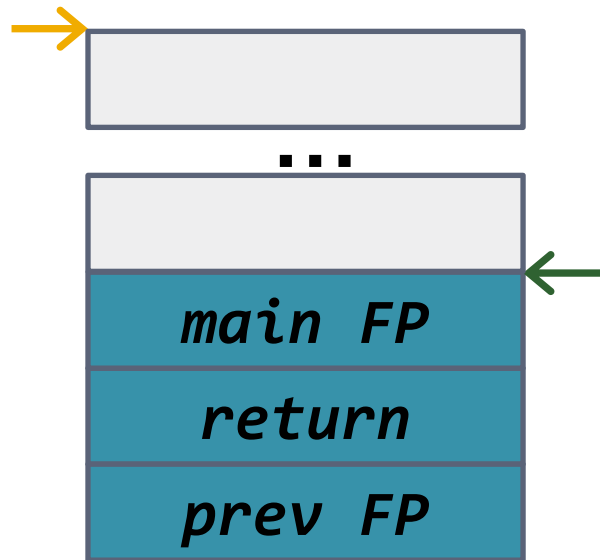
example.s (x64)



foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave
ret
```

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



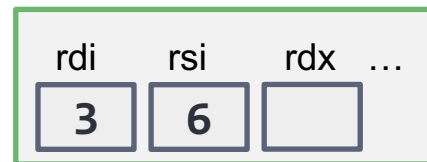
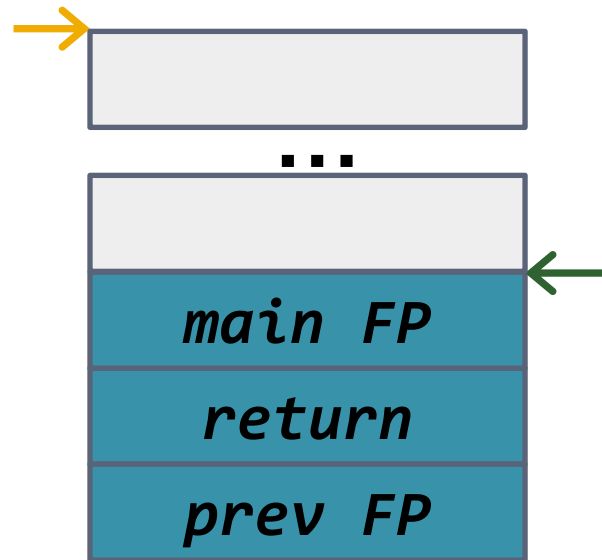
example.s (x64)



foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave
ret
```

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



example.s (x64)

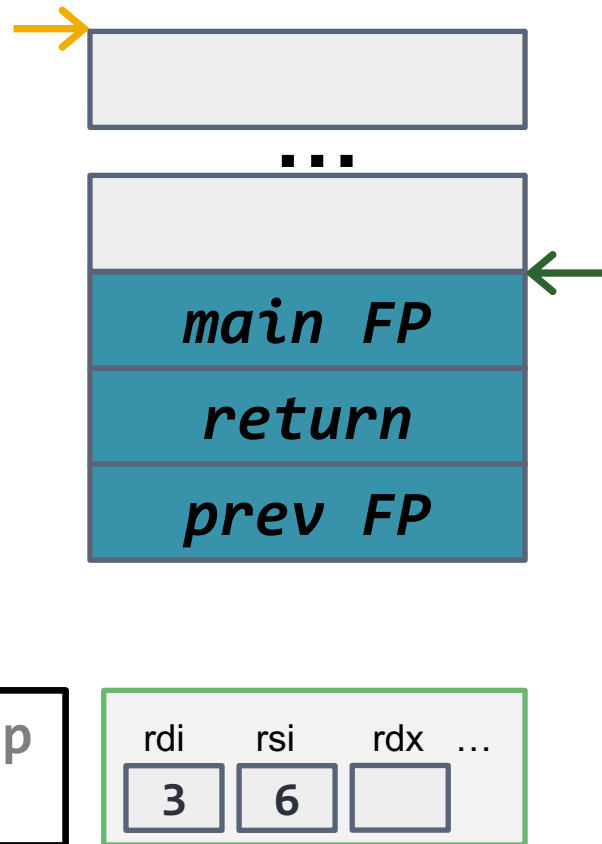


foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave
ret
```

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

```
mov     rsp, rbp
pop     rbp
```



example.s (x64)

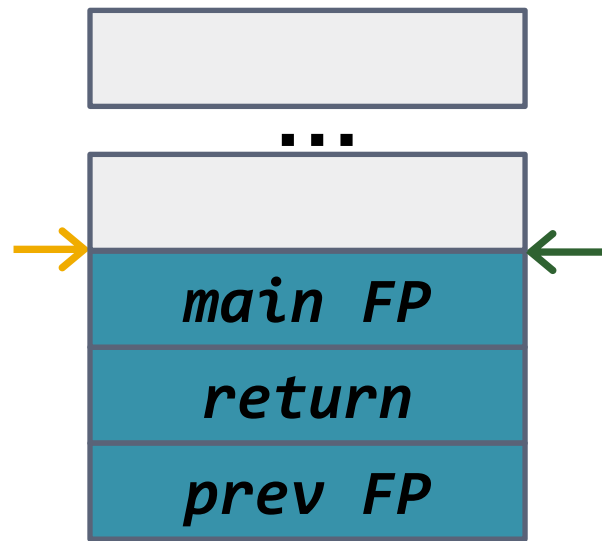


foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave    ←
ret
```

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

```
mov     rsp, rbp
pop     rbp
```



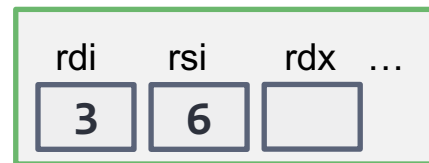
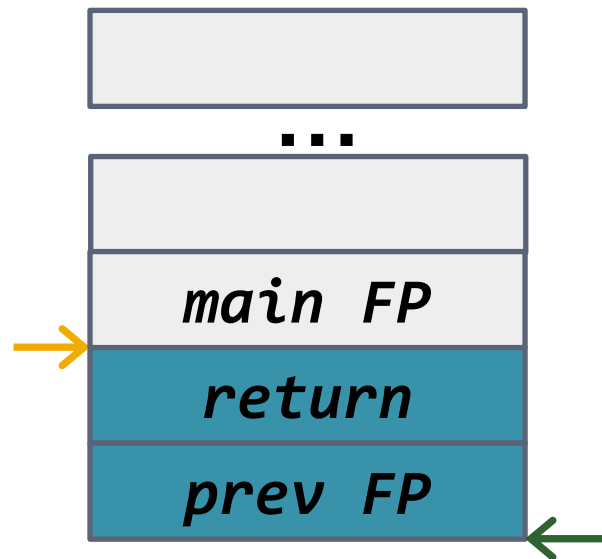
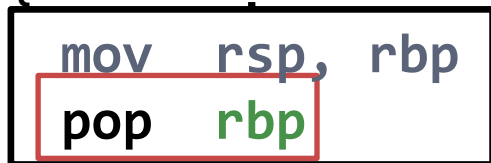
example.s (x64)



foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave    ←
ret
```

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



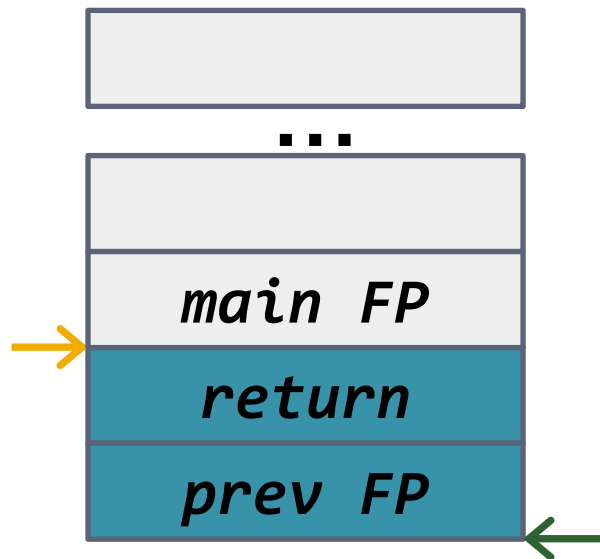
example.s (x64)



foo:

```
    push    rbp
    mov     rbp, rsp
    sub     rsp, 0x10
    nop
    leave
    ret
```

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



example.s (x64)



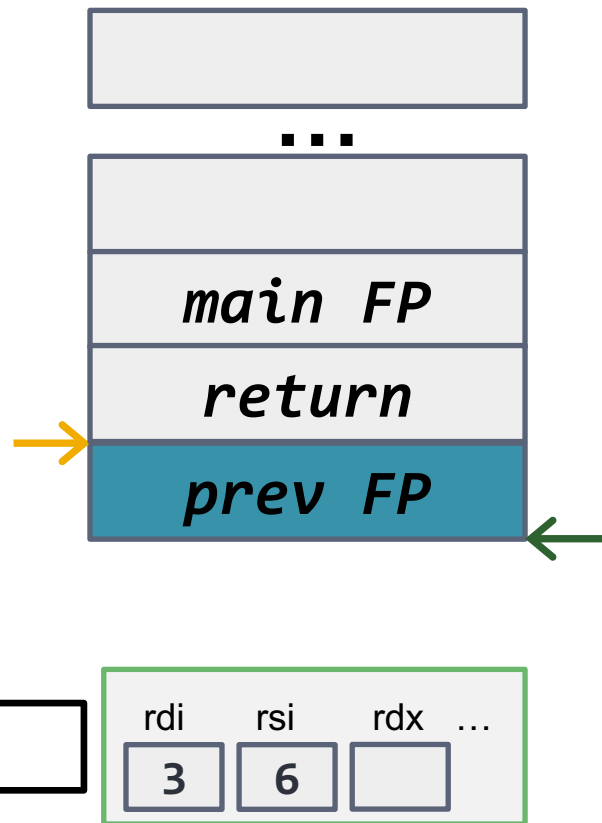
foo:

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x10
nop
leave
```

ret ←

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

pop **rip**

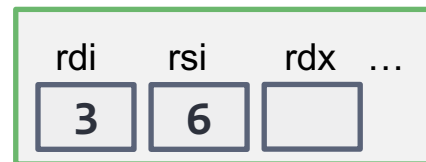
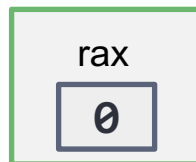


example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```

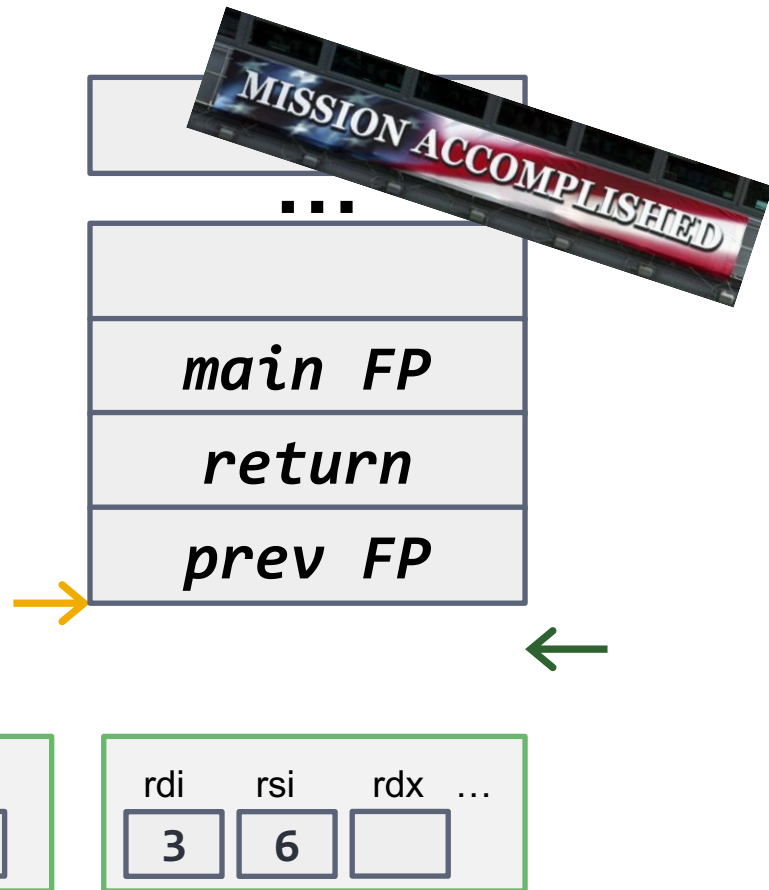


example.s (x64)



main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
ret
```



example.s (x64)

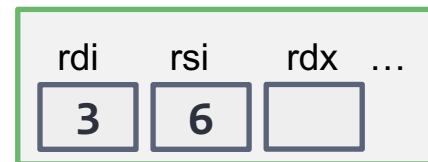
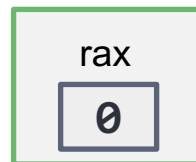


main:

```
push    rbp
mov     rbp, rsp
mov     rsi, 0x6
mov     rdi, 0x3
call    foo
mov     rax, 0x0
pop     rbp
```

ret

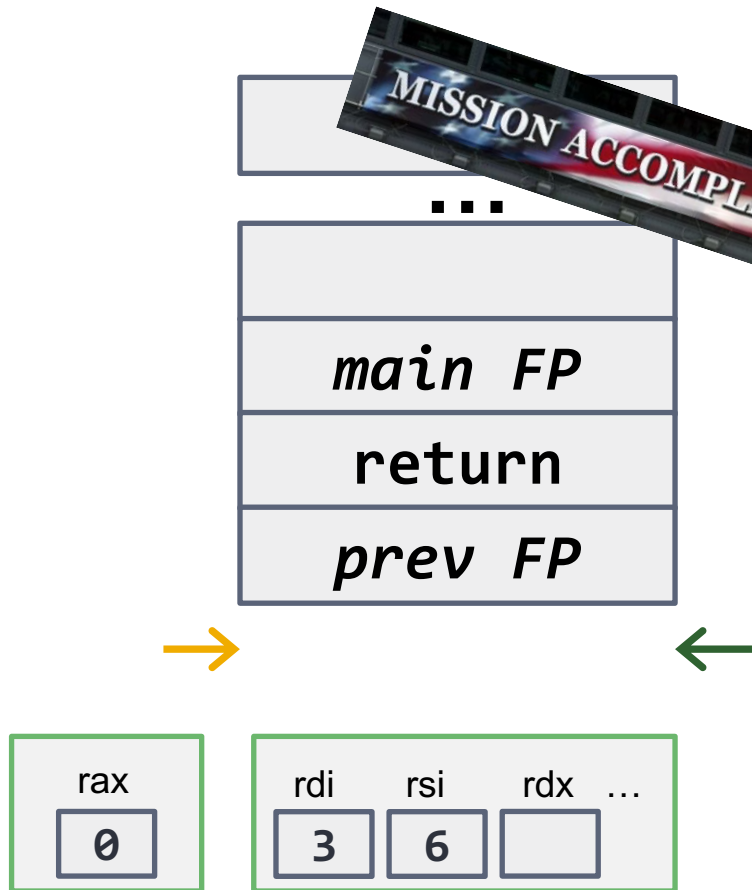
pop rip



example.s (x64)



```
main:
    push    rbp
    mov     rbp, rsp
    mov     rsi, 0x6
    mov     rdi, 0x3
    call    foo
    mov     rax, 0x0
    pop     rbp
    ret
```



Outline



Bits and pieces

- Memory: Address space

- CPU: Registers and Instructions

- Disassembly

Stacking things up

- Stack frames

- Stack in assembly

Blowing things up

- Buffer overflows

- Shellcode

- Simple defense (DEP)

Buffer overflow example



```
#include <string.h>

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

int main() {
    char *str = "AAAABBBBBBBBBB1234567";
    foo(str);          └─ 12 Bytes
}
```

example.s (x64)



```
int main() {  
    char *str = "AAAABBBBBBBBBB1234567";  
    foo(str);  
}
```

main:

```
    push    rbp  
    mov     rbp, rsp  
    lea     rdi,[rel str_ptr]  
    call    foo  
    pop     rbp  
    leave  
    ret
```

.rodata:

str_ptr: "AAAABBBBBBBBBB1234567"

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

foo:

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

example.s (x64)

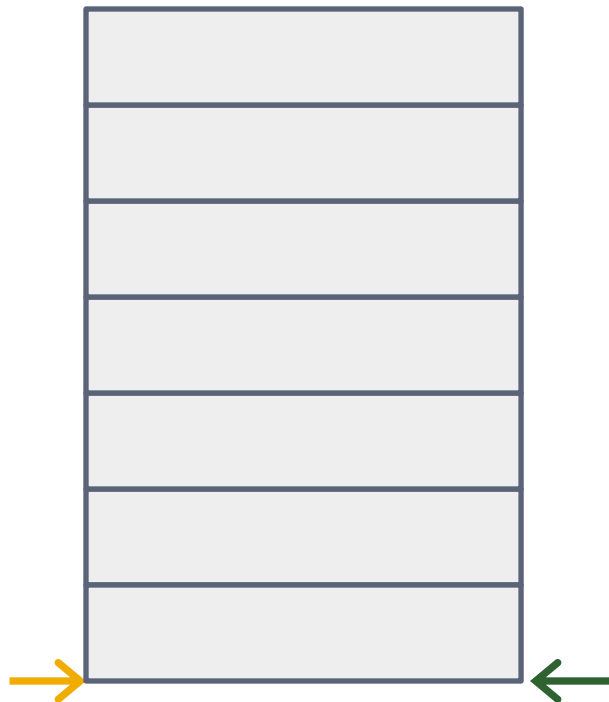


```
int main() {  
    char *str = "1234567890A";  
    foo(str);  
}
```

main:

```
    push    rbp  
    mov     rbp, rsp  
    lea     rdi, [rel str_ptr]  
    call    foo  
    pop     rbp  
    leave  
    ret
```

str_ptr: "AAAABBBBBBBBBB1234567"



example.s (x64)

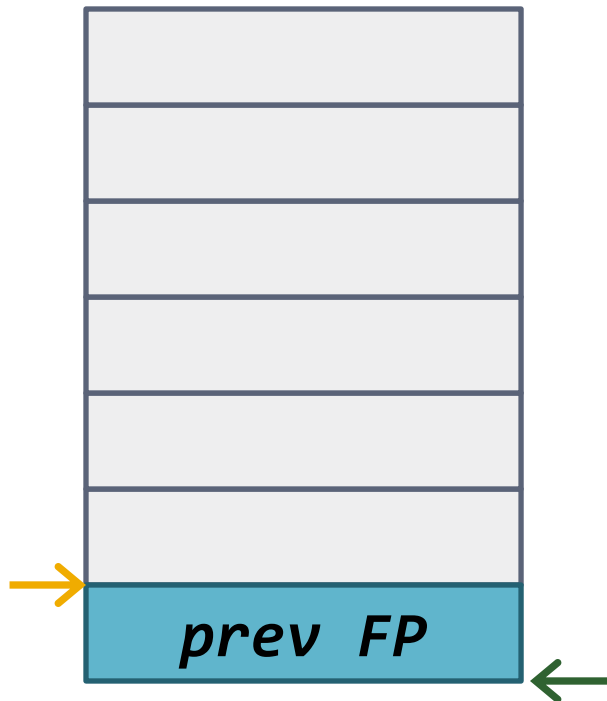


```
int main() {  
    char *str = "1234567890A";  
    foo(str);  
}
```

main:

```
push    rbp  
mov     rbp, rsp  
lea     rdi, [rel str_ptr]  
call    foo  
pop     rbp  
leave  
ret
```

str_ptr: "AAAABBBBBBBBBB1234567"



example.s (x64)

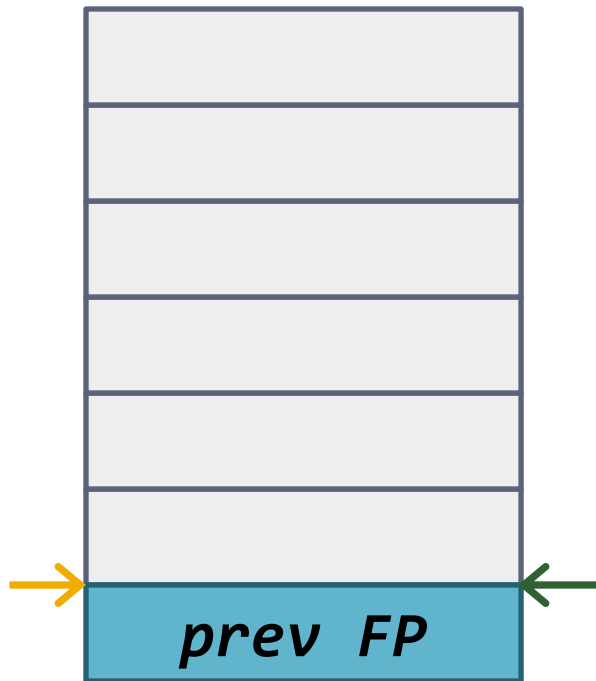


```
int main() {  
    char *str = "1234567890A";  
    foo(str);  
}
```

main:

```
    push    rbp  
    mov     rbp, rsp  
    lea     rdi, [rel str_ptr]  
    call    foo  
    pop     rbp  
    leave  
    ret
```

str_ptr: "AAAABBBBBBBBBB1234567"



example.s (x64)



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

main:

push rbp

mov rbp, rsp

lea rdi, [rel str_ptr]

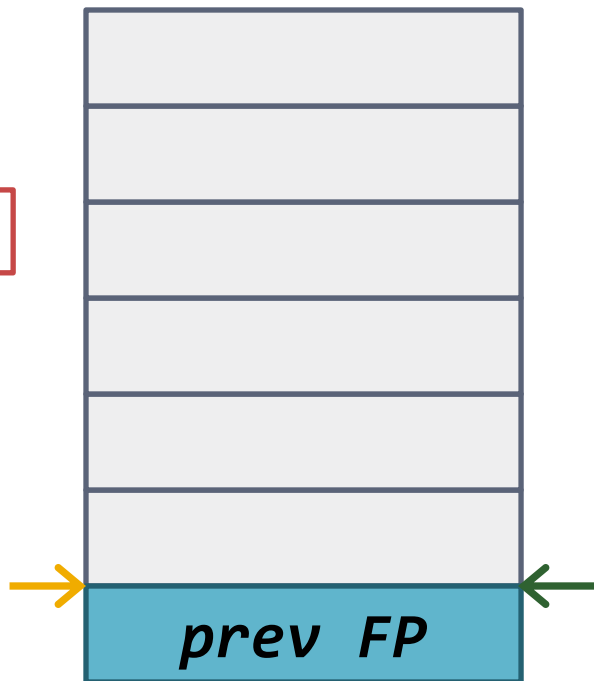
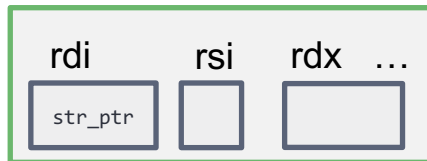
call foo

pop rbp

leave

ret

str_ptr: "AAAABBBBBBBB1234567"

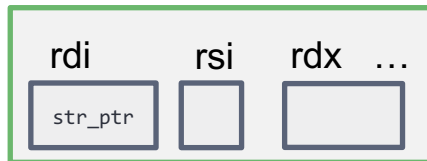


example.s (x64)

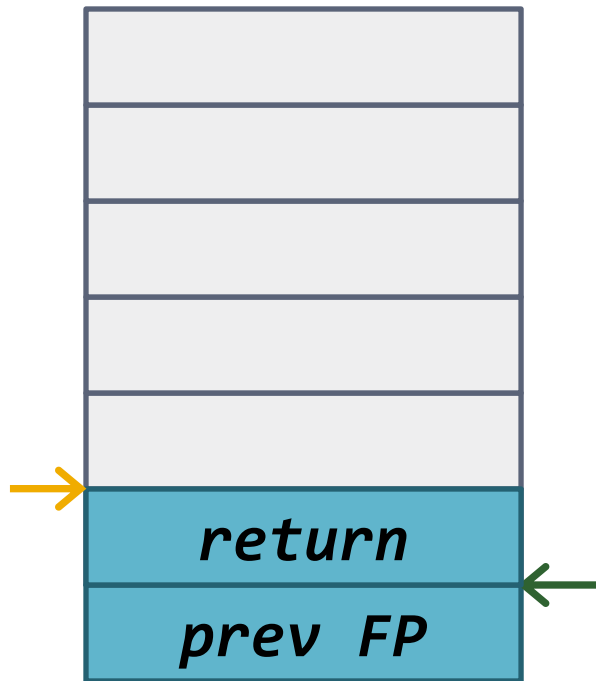


```
int main() {  
    char *str = "1234567890A";  
    foo(str);  
}
```

```
main:  
    push    rbp  
    mov     rbp, rsp  
    lea     rdi, [rel str_ptr]  
    call    foo  
    pop     rbp  
    leave  
    ret
```



str_ptr: "AAAABBBBBBBBBB1234567"



example.s (x64)



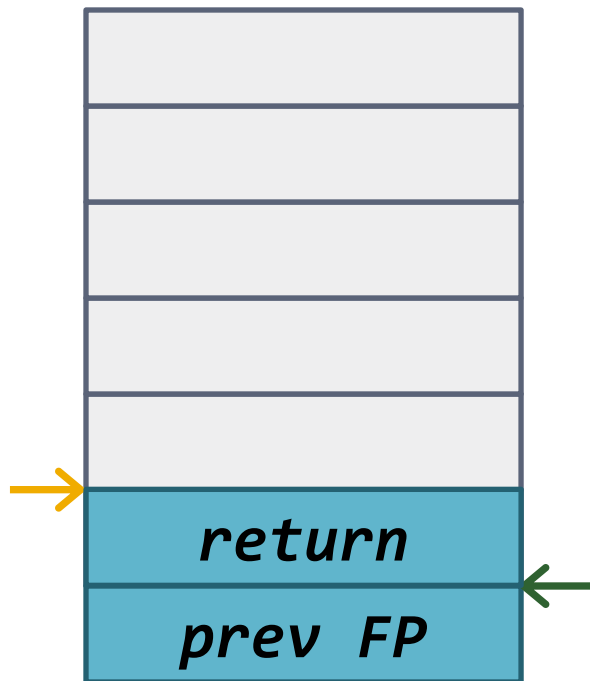
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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



str_ptr: "AAAABBBBBBBBBB1234567"



example.s (x64)



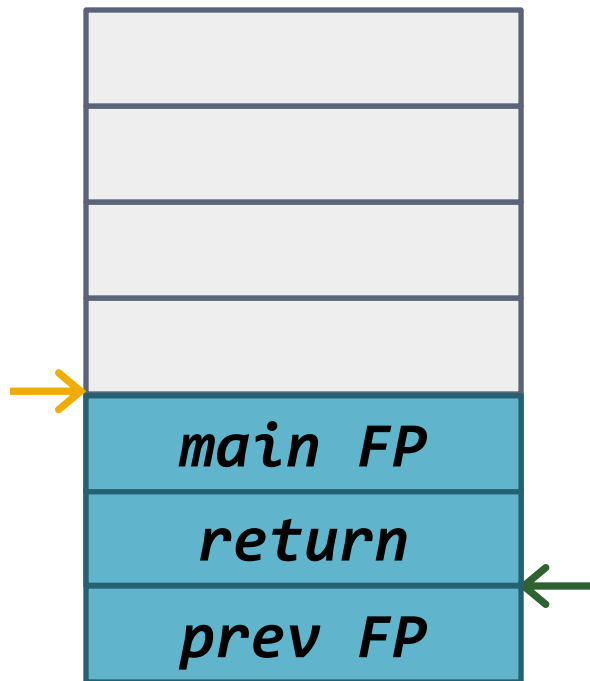
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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



str_ptr: "AAAABBBBBBBB1234567"



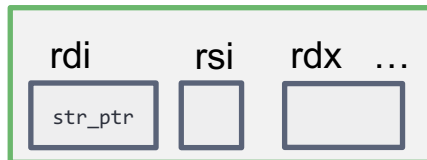
example.s (x64)



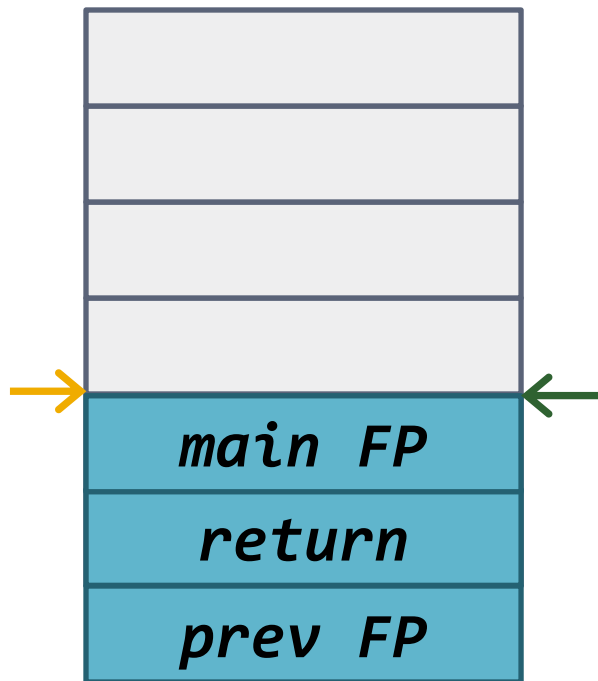
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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



str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)



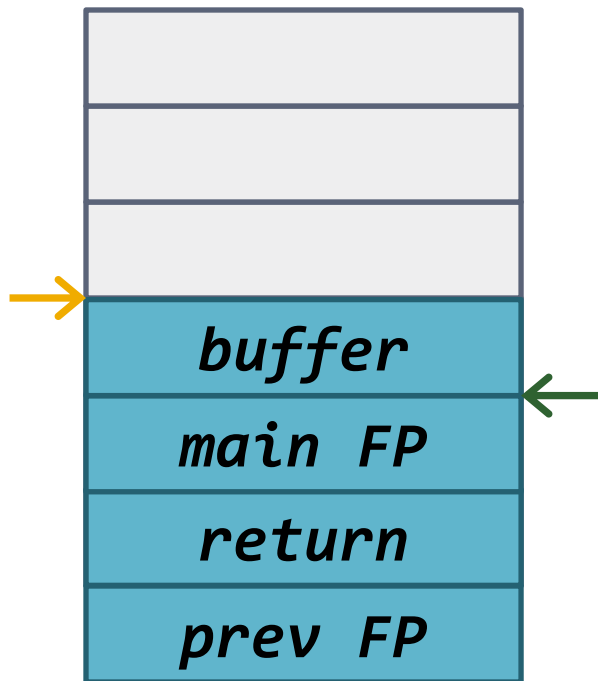
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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



str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)



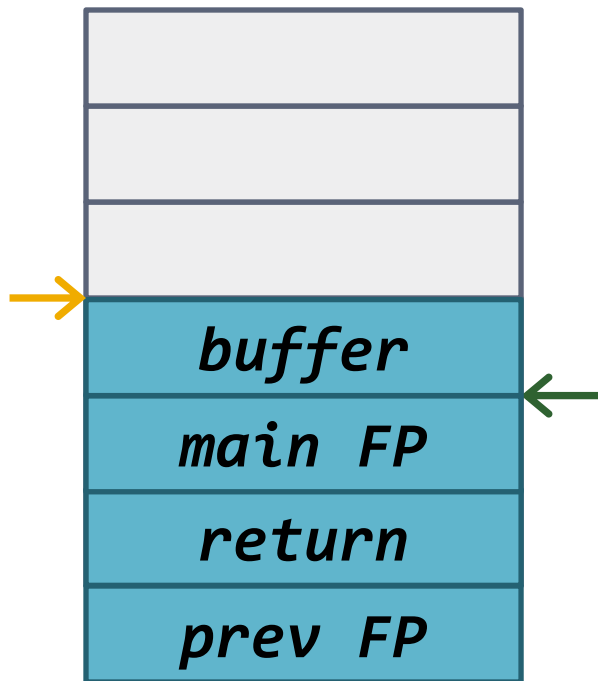
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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



str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)

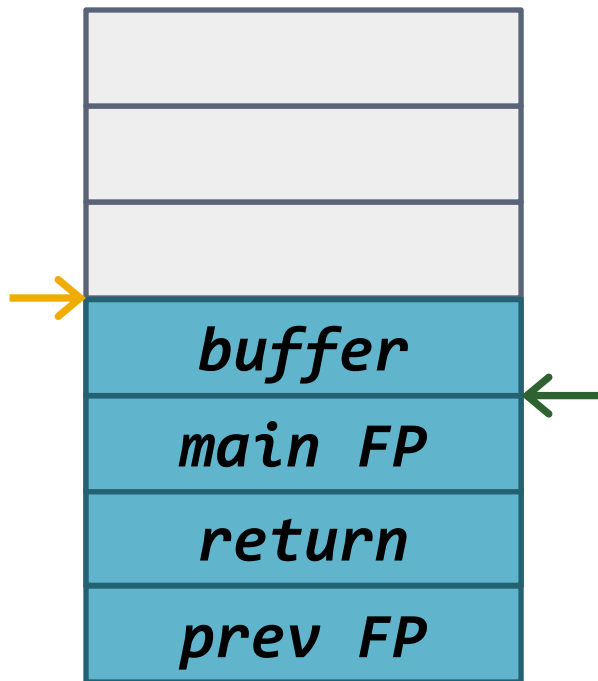


```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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

str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)

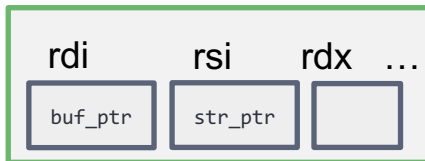
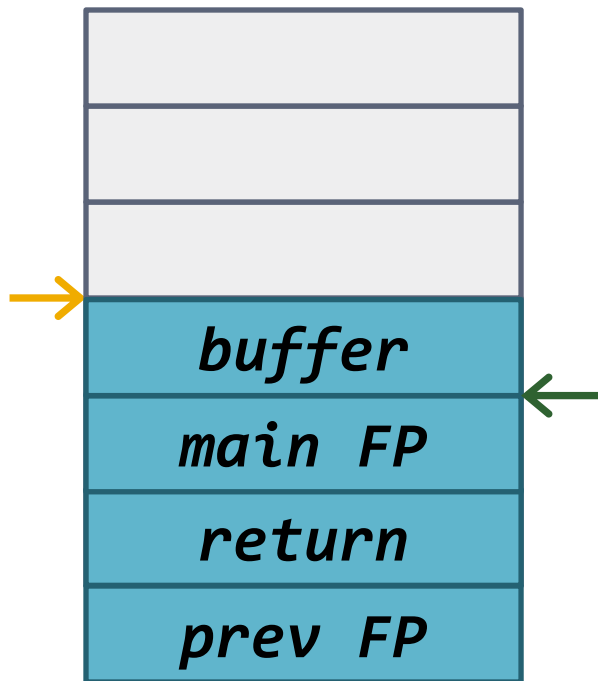


```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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

str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)



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

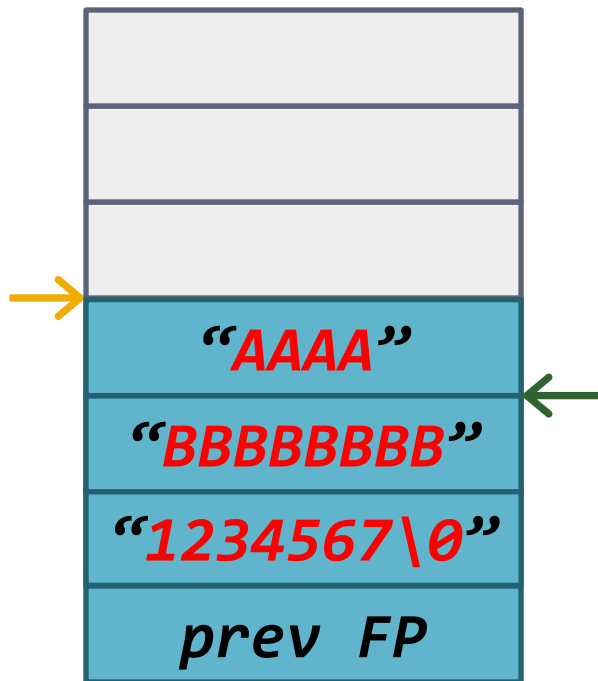
foo:

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

call strcpy



str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)



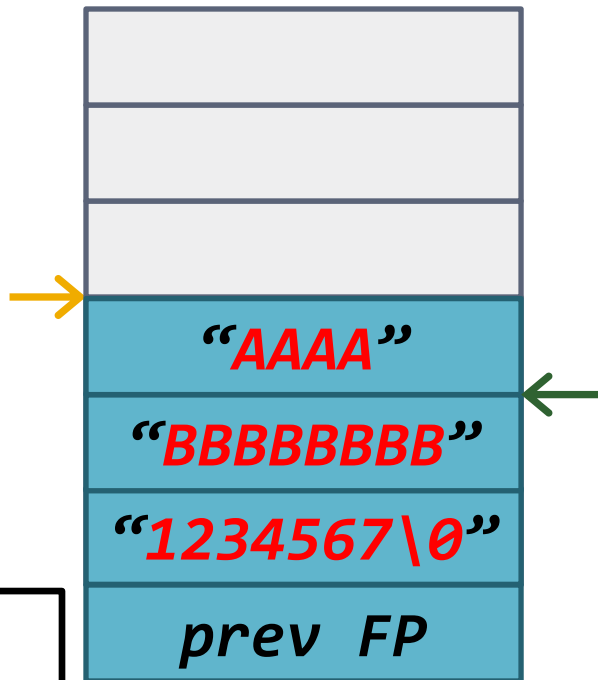
```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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

```
    mov     rsp, rbp  
    pop     rbp
```

str_ptr: "AAAABBBBBBBB1234567"



example.s (x64)

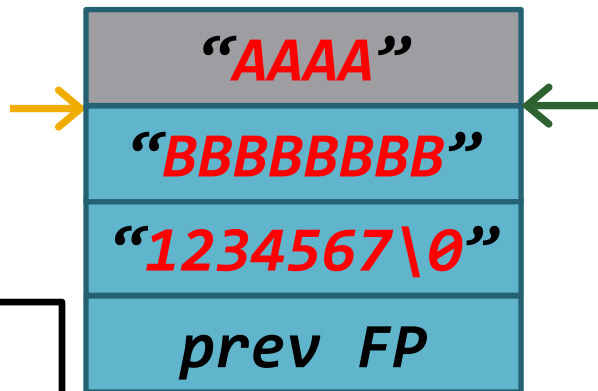


```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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

```
    mov     rsp, rbp  
    pop     rbp
```



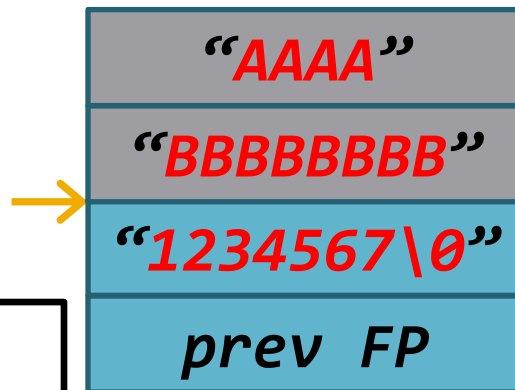
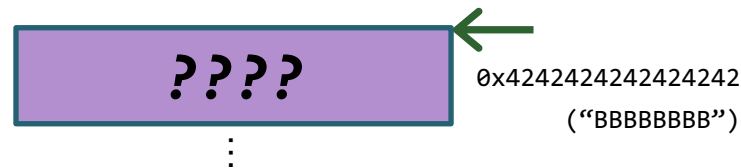
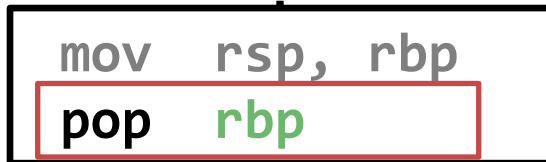
example.s (x64)



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

foo:

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



example.s (x64)

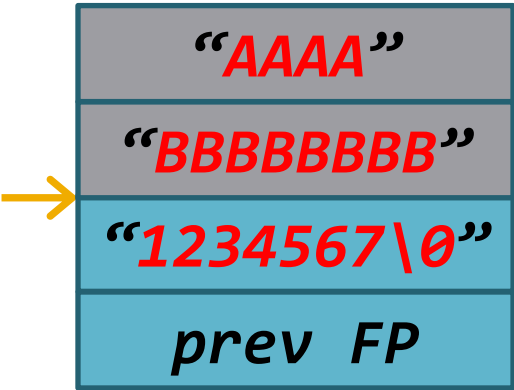
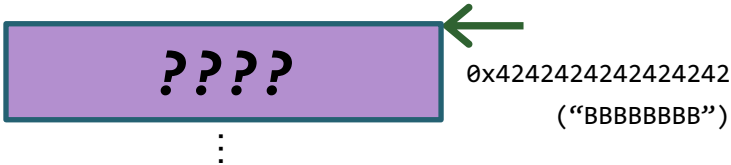


```
void foo(char *str) {  
    char buffer[4];  
    strcpy(buffer, str);  
}
```

foo:

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

```
pop    rip
```



example.s (x64)



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

foo:

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

pop rip

????

⋮

← 0x4242424242424242
("BBBBBBBB")

"AAAA"

"BBBBBBBBBB"

→ "1234567\0"

prev FP

example.s (x64)



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

foo:

```
push rbp  
mov rbp, rbp  
sub rbp, 0x10  
mov rdi, str  
lea rsi, buffer  
call strcpy@PLT  
leave rbp  
ret
```

rip = 0x3132333435363700
("1234567\0")

```
pop rip
```

????

0x4242424242424242
("BBBBBBBB")







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



```
#include <string.h>

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

int main() {
    char *str = "AAAABBBBBBBBBB1234567";
    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>
```

Let's do something more useful than crashing

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

Let's do something more useful than crashing

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

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

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

		undefined <code>main()</code>	
000111b1	55	PUSH	RBP
000111b3	48 89 e5	MOV	RBP, RSP
		LAB_000111b4	
000111b6	eb fe	JMP	LAB_000111b4

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

		undefined main()
000111b1	55	PUSH RBP
000111b3	48 89 e5	MOV RBP,RSP
		LAB_000111b4
000111b6	eb fe	JMP LAB_000111b4

Good

<i>buffer</i>
<i>main FP</i>
<i>return</i>
<i>prev FP</i>

Stack Shellcode



Good

<i>buffer</i>
<i>main FP</i>
<i>return</i>
<i>prev FP</i>

Evil

<i>“AAAA”</i>
<i>“BBBBBBBB”</i>
<i>“1234567\0”</i>
<i>prev FP</i>

Stack Shellcode



Good

<i>buffer</i>
<i>main FP</i>
<i>return</i>
<i>prev FP</i>

Evil

<i>“AAAA”</i>
<i>“BBBBBBBB”</i>
<i>“1234567\0”</i>
<i>prev FP</i>

DEADLY

<i>0xEBFE4141...</i>
<i>0x41414141...</i>
<i>0xFFFFFFFF88888888</i>
<i>prev FP</i>

0xffffffff88888888

Stack Shellcode



Good

Evil

DEADLY

buffer

“AAAA”

0xEBFE4141...

0xffffffff88888888

0xffffffff8888

000111b1 55

000111b3 48 89 e5

000111b6 eb fe

undefined *main()*

PUSH RBP

MOV RBP, RSP

LAB_000111b4

JMP LAB_000111b4

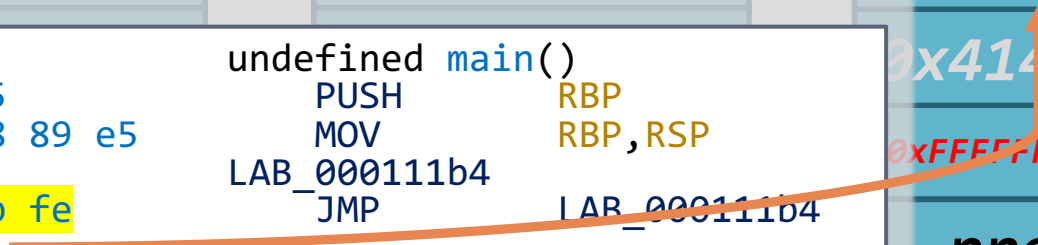
0x41414141...

0xFFFFFFFF88888888

prev FP

prev FP

prev FP

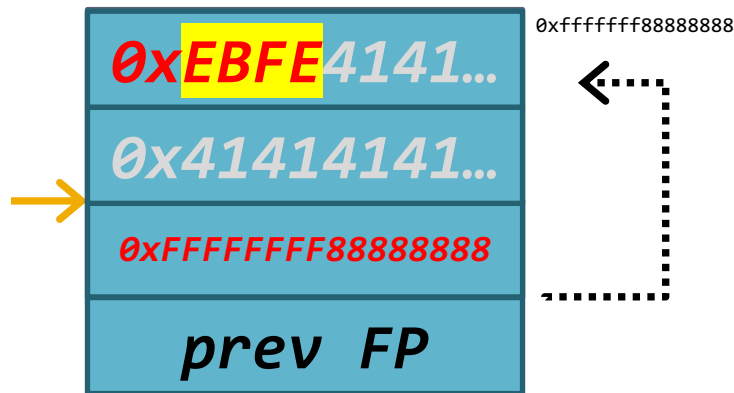


Stack Shellcode



foo:

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



str_ptr:

“\xEB\xFE\x41\x41...\x41\x41\xff\xff\xff\xff\x88\x88\x88\x88”

Stack Shellcode



shellcode:

`jmp shellcode`



“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?

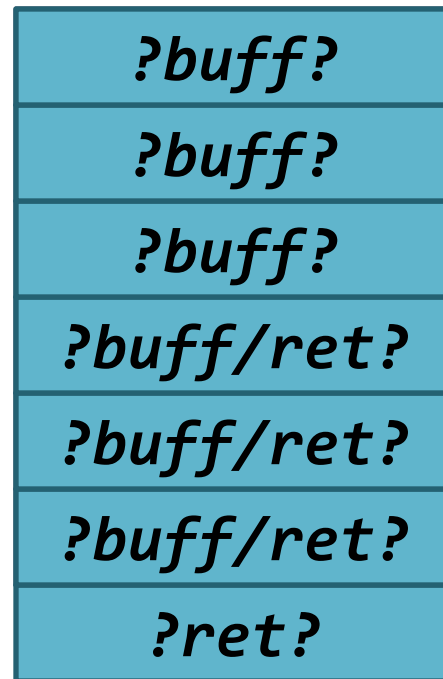
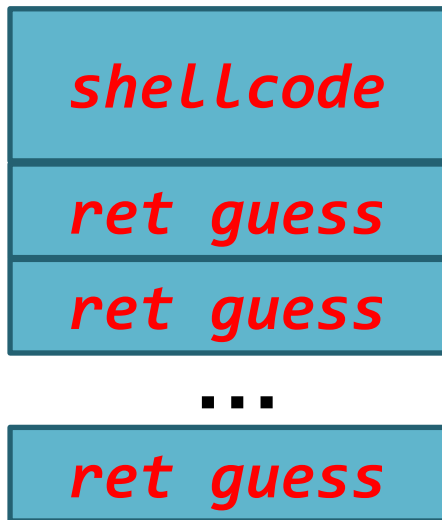
Hard to guess address



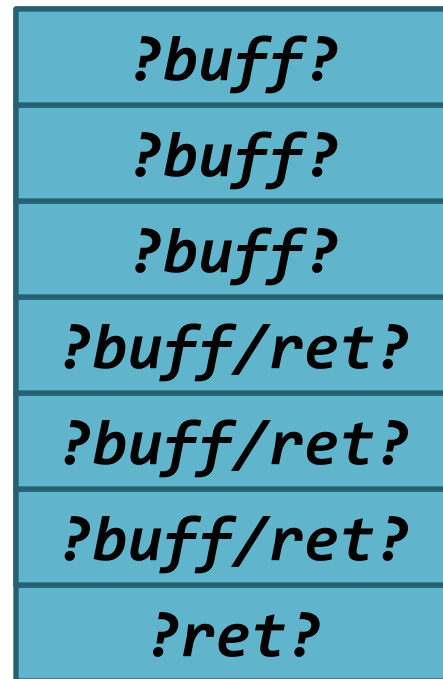
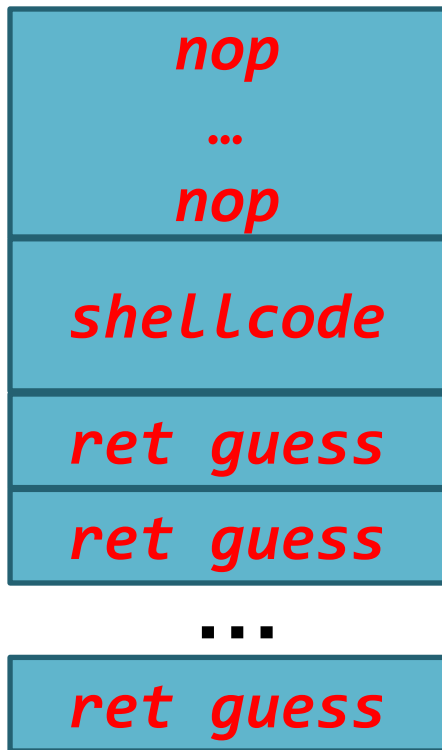
<i>shellcode</i>
<i>ret guess</i>

<i>?buff?</i>
<i>?buff?</i>
<i>?buff?</i>
<i>?buff/ret?</i>
<i>?buff/ret?</i>
<i>?buff/ret?</i>
<i>?ret?</i>

Hard to guess address



Hard to guess address



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

Cat-and-Mouse Exploitation



DEP

Stack Canaries

ASLR

Automated Testing

Buffer Overflow
Stack Shellcode

Data-only attacks
Return-to-libc

Buffer Over-read
Integer Overflow
ROP

Toolbox of Exploitation Techniques

Cat-and-Mouse Exploitation



Data Execution Prevention (DEP)

Stack Canaries

ASLR

Automated Testing

Buffer Overflow
Stack Shellcode

Data-only attacks
Return-to-libc

Buffer Over-read
Integer Overflow
ROP

Toolbox of Exploitation Techniques

Data Execution Prevention (DEP)



Defender's problem:
data and code are the same

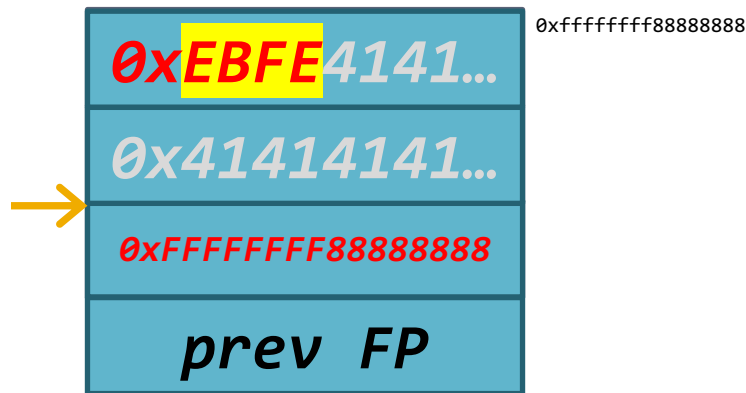
Solution:

Write \oplus Execute

(enforced by hardware or OS)

foo:

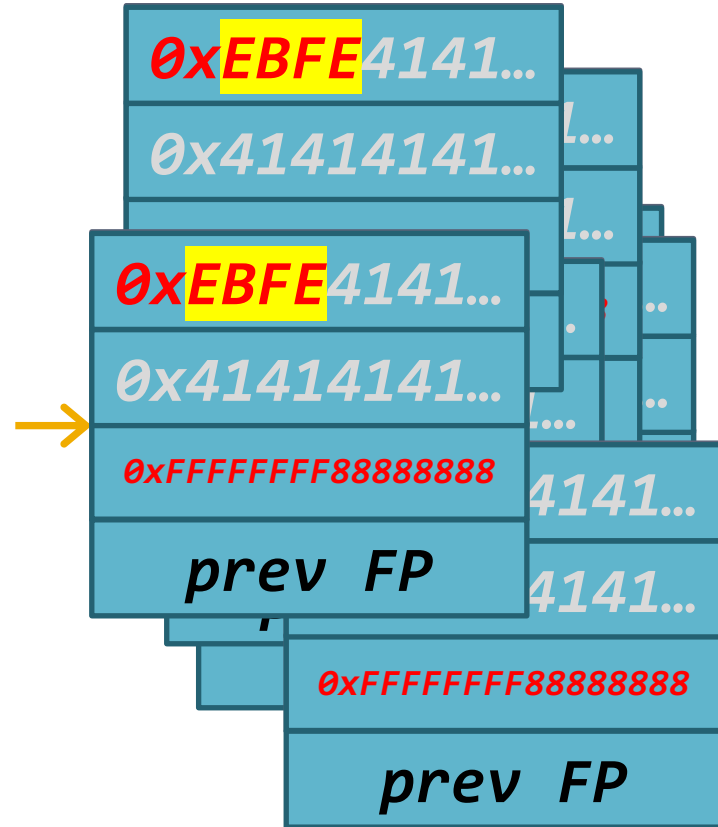
```
push    rbp
mov     rbp, rsp
sub     rsp, 4
mov     rsi, rdi
lea     rdi, [rbp-4]
call    strcpy
leave
ret
```



str_ptr:

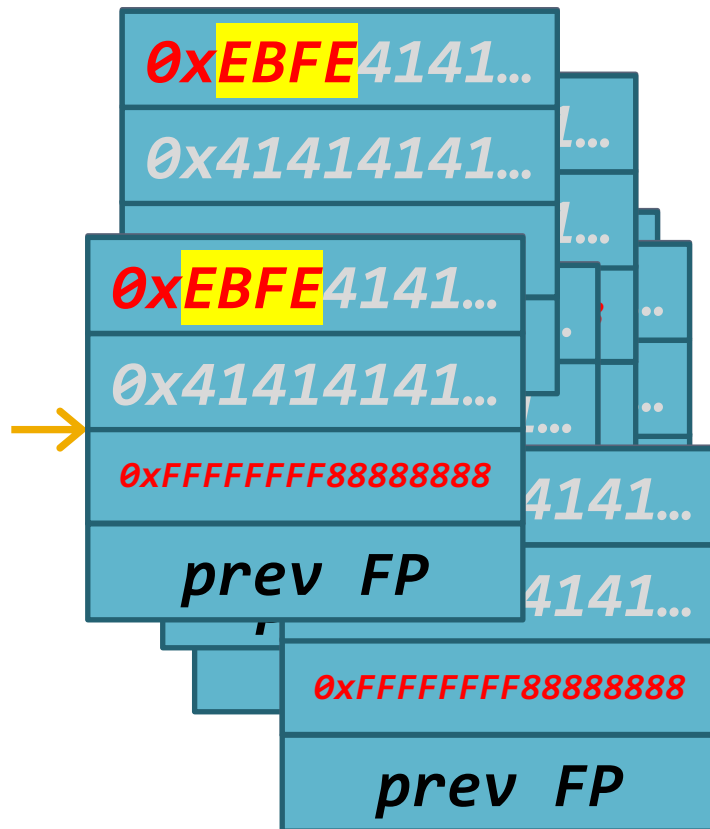
“\xEB\xFE\x41\x41...\x41\x41\xFF\xFF\xFF\xFF\x88\x88\x88\x88”

OS ERROR



OS ERROR

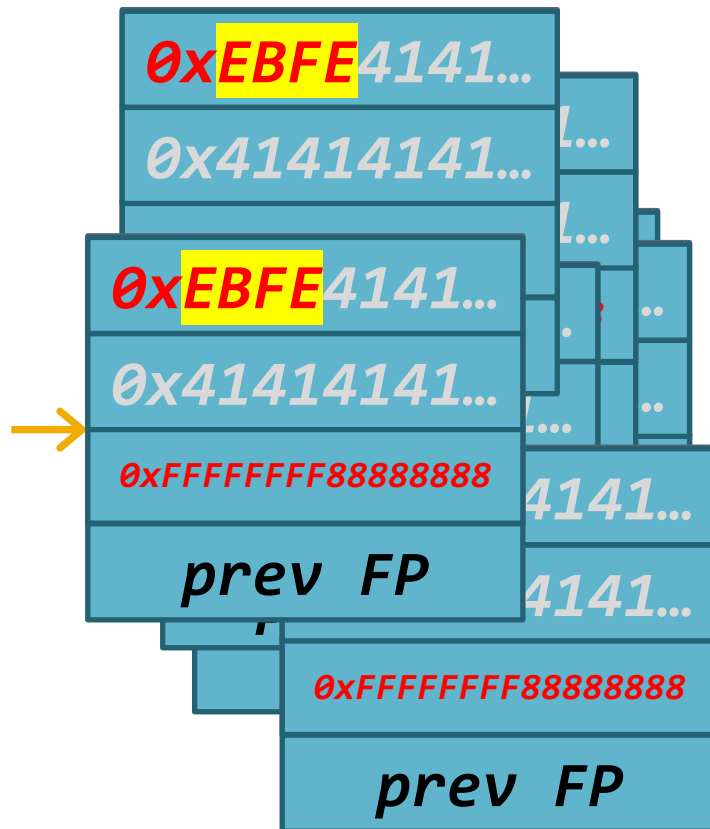
CONTROL FLOW
IS INCORRECT



OS ERROR

CONTROL FLOW
IS INCORRECT

IMMEDIATELY
END PROCESS



Cat-and-Mouse Exploitation



DEP

Stack Canaries

ASLR

Automated Testing

Buffer Overflow
Stack Shellcode

Data-only attacks
Return-to-libc

Buffer Over-read
Integer Overflow
ROP

Toolbox of Exploitation Techniques

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);
```

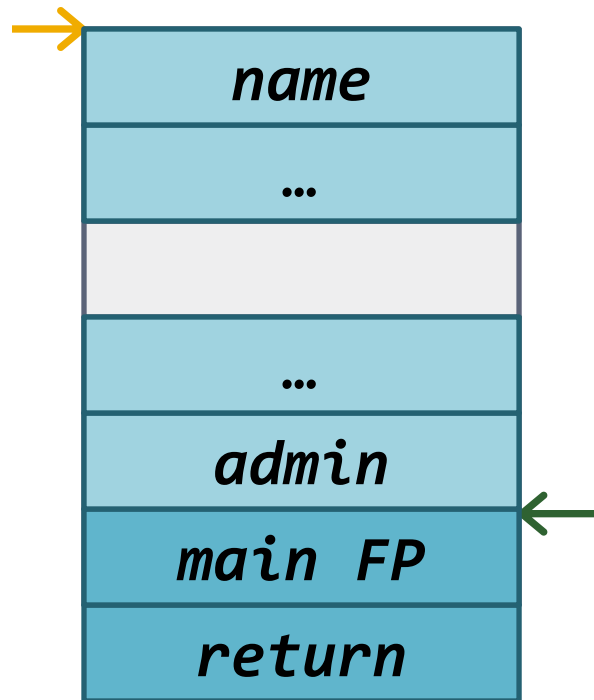
```
int delete_account(char* username,
    int length, VOID* creds) {
    int admin;
    char name[100];
    admin = check_admin(creds);
    strncpy(name, username, length);
    canonicalize_username(name);
    if (admin) { delete_user(name); }
    return (admin == 0);
}
```

```
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



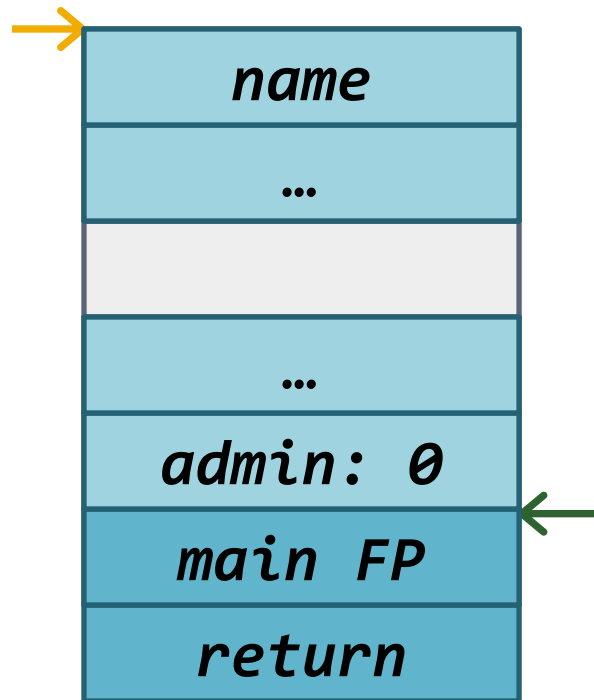
```
int delete_account(char* username,  
    int length, VOID* creds) {  
    int admin;  
    char name[100];  
    admin = check_admin(creds);  
    strncpy(name, username, length);  
    canonicalize_username(name);  
    if (admin) { delete_user(name); }  
    return (admin == 0);  
}
```



Data-only attacks



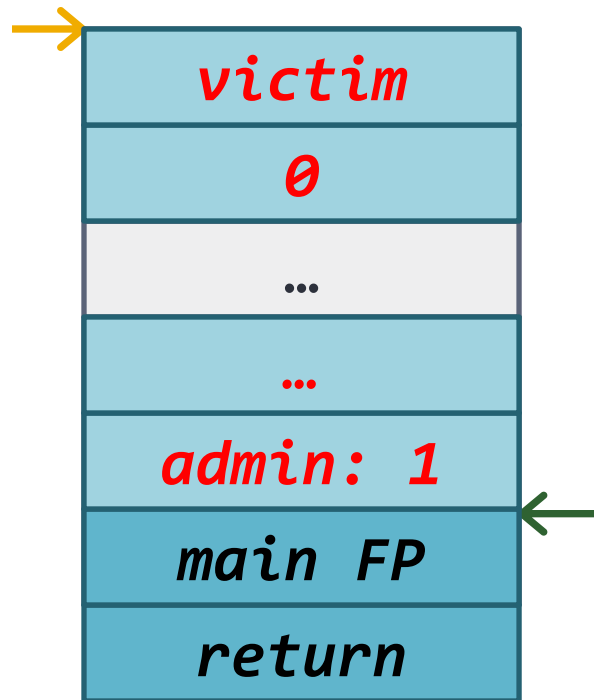
```
int delete_account(char* username,  
    int length, VOID* creds) {  
    int admin;  
    char name[100];  
    admin = check_admin(creds);  
    strncpy(name, username, length);  
    canonicalize_username(name);  
    if (admin) { delete_user(name); }  
    return (admin == 0);  
}
```



Data-only attacks



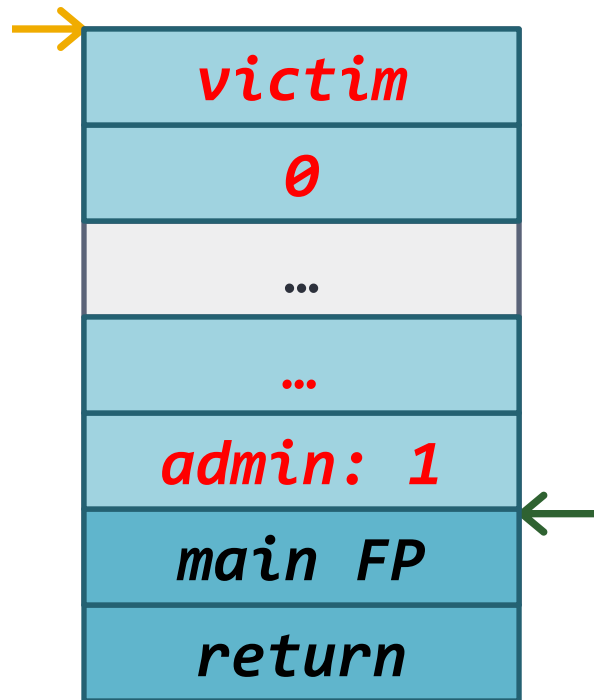
```
int delete_account(char* username,  
    int length, VOID* creds) {  
    int admin;  
    char name[100];  
    admin = check_admin(creds);  
    strncpy(name, username, length);  
    canonicalize_username(name);  
    if (admin) { delete_user(name); }  
    return (admin == 0);  
}
```



Data-only attacks



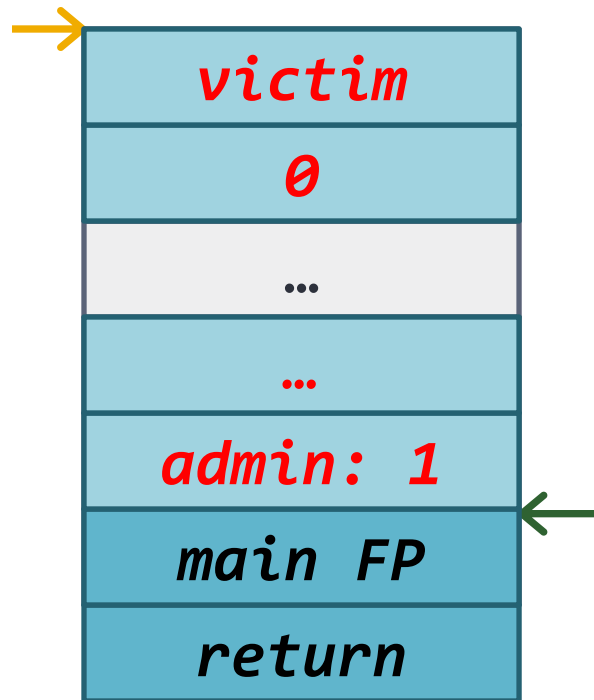
```
int delete_account(char* username,  
    int length, VOID* creds) {  
    int admin;  
    char name[100];  
    admin = check_admin(creds);  
    strncpy(name, username, length);  
    canonicalize_username(name);  
    if (admin) { delete_user(name); }  
    return (admin == 0);  
}
```



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);  
}
```



Can we do worse?
To be continued!

So far

- Computer organization background
- Basic control hijacking techniques

Upcoming...

More control hijacking



- Don't roll your own crypto!
- Avoid using C

Coming Up



Project 3 due on Thursday at 6 p.m.

Project 4 out Thursday, due Nov. 16 (three weeks from Thursday)

Thursday, Oct. 26

Control Hijacking, Part 2

Tuesday, October. 31

Malware