CSE 127: Computer Security

Stack Buffer Overflows

Deian Stefan

Adopted from Kirill Levchenko, Stefan Savage, and Hovav Shacham

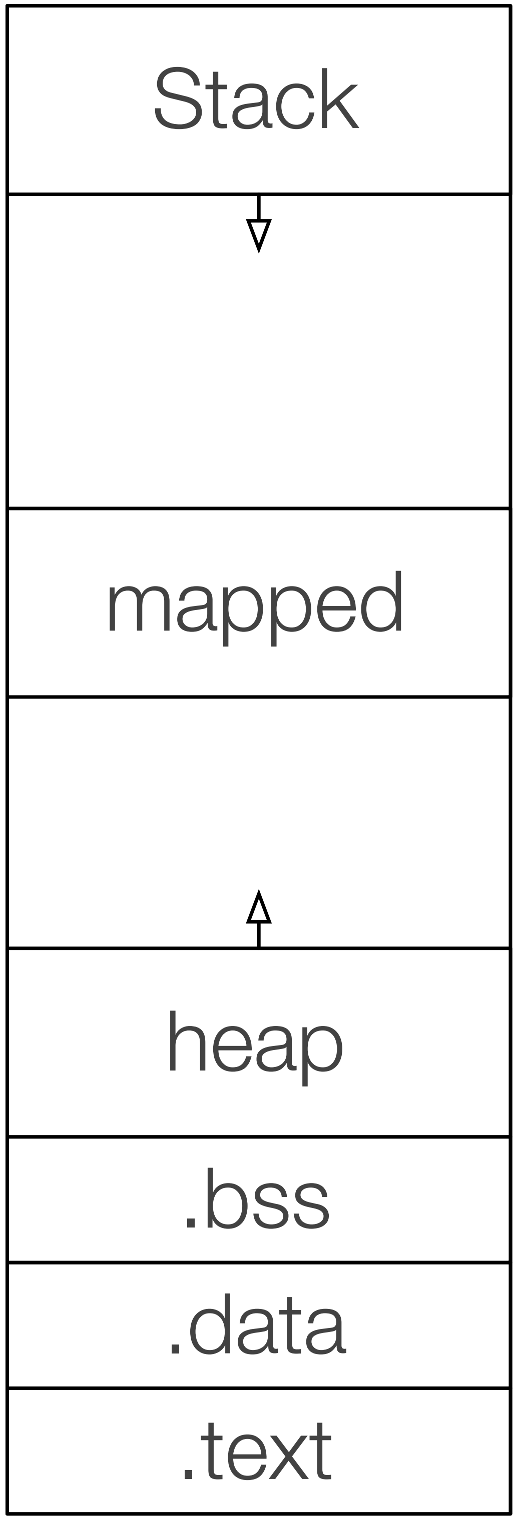
Control Flow Hijacking Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both (W^X)
- Control flow integrity (CFI)

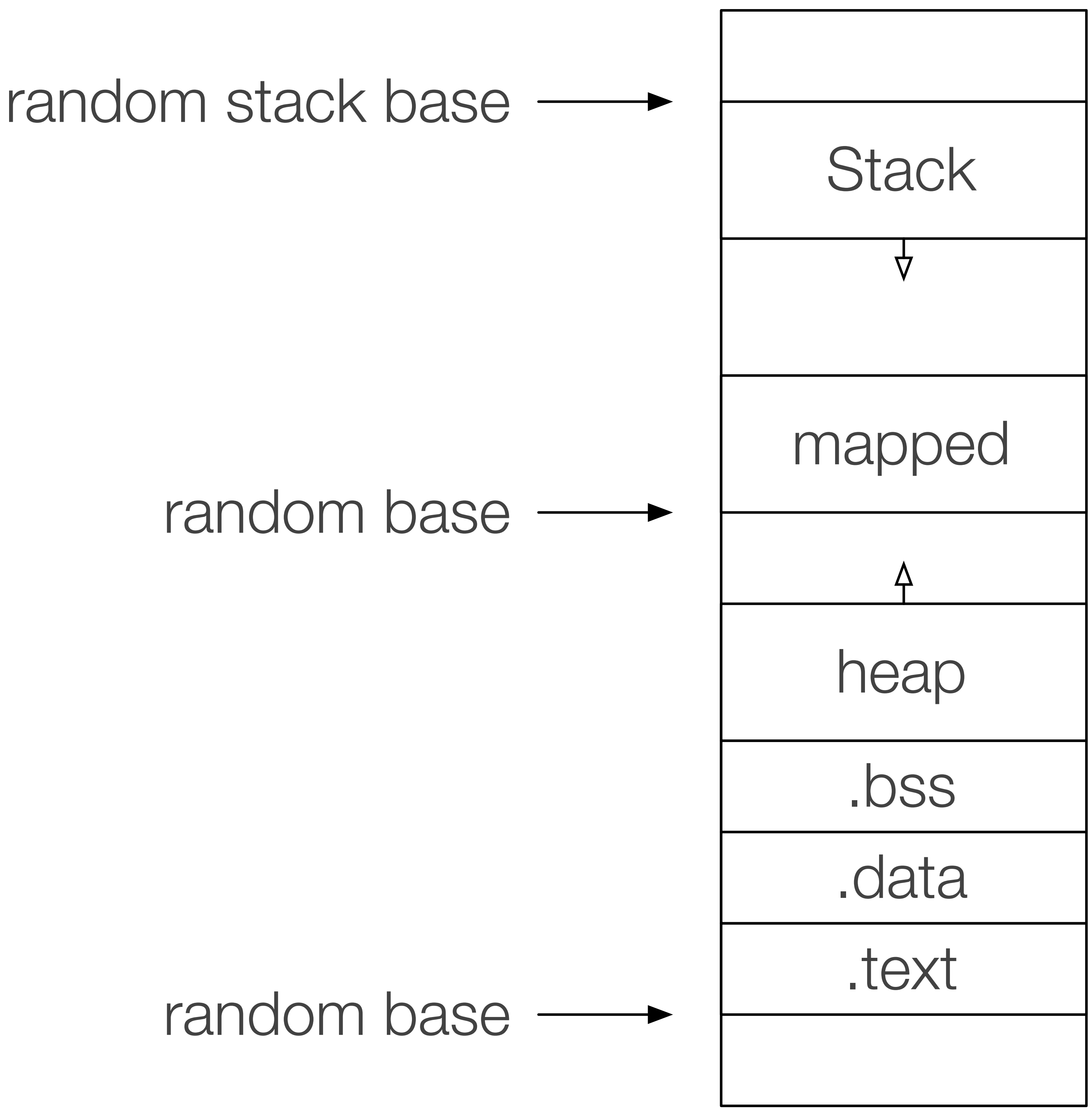
Address Space Layout Randomization

- Change location of stack, heap, code, static vars
- Works because attacker needs address of shellcode
- Layout must be unknown to attacker
  - Randomize on every launch (best)
  - Randomize at compile time
- Implemented on most modern OSes in some form

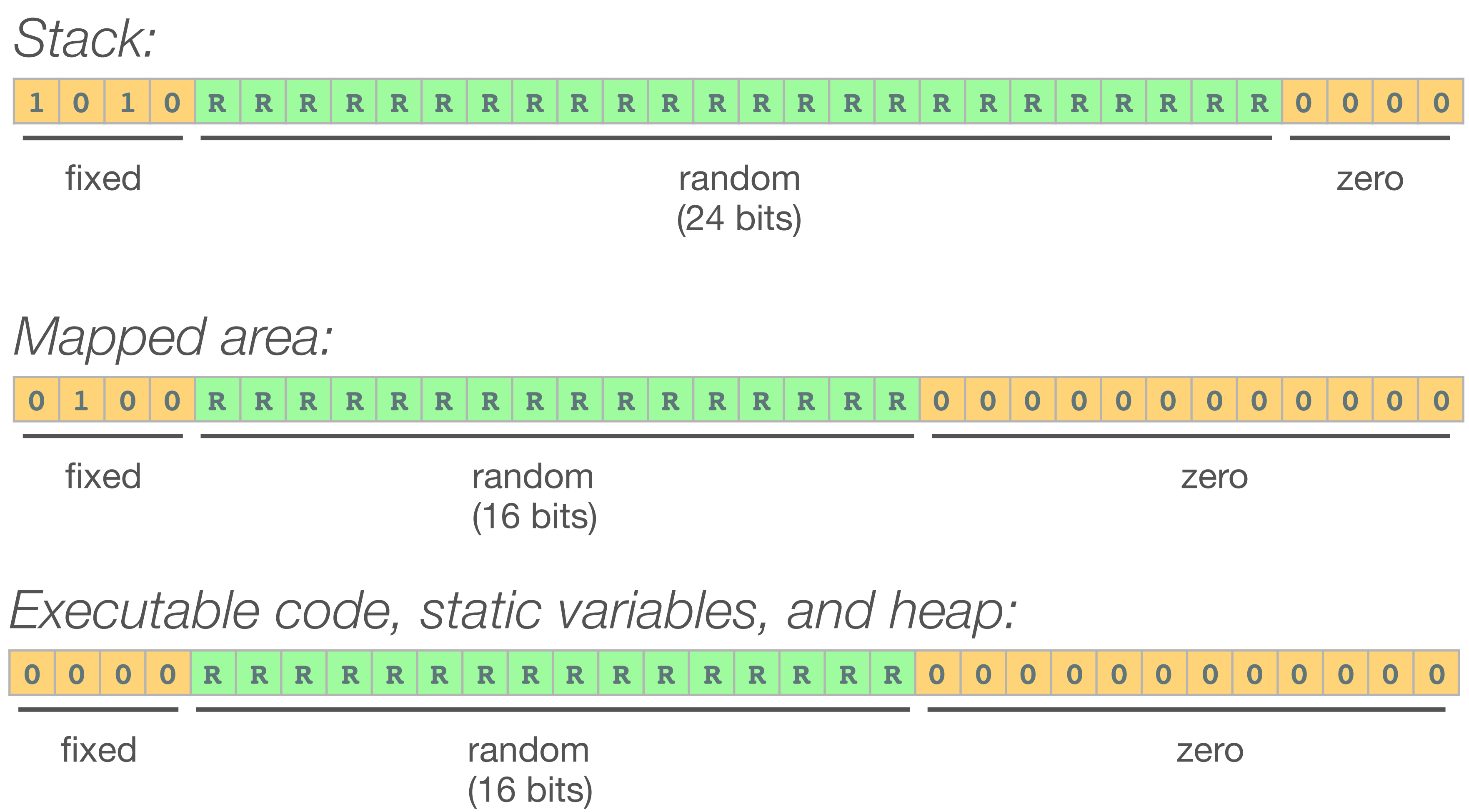
Traditional Memory Layout



PaX Memory Layout



32-bit PaX ASLR (x86)



On the Effectiveness of Address-Space Randomization

Hovav Shacham
Stanford University
hovav@cs.stanford.edu

Matthew Page
Stanford University
mpage@stanford.edu

Ben Pfaff
Stanford University
blp@cs.stanford.edu

Eu-Jin Goh
Stanford University
eujin@cs.stanford.edu

Nagendra Modadugu
Stanford University
nagendra@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Derandomizing ALSR

- **Attack goal:** call system() with attacker arg
- **Target:** Apache daemon
  - **Vulnerability:** buffer overflow in ap_getline()

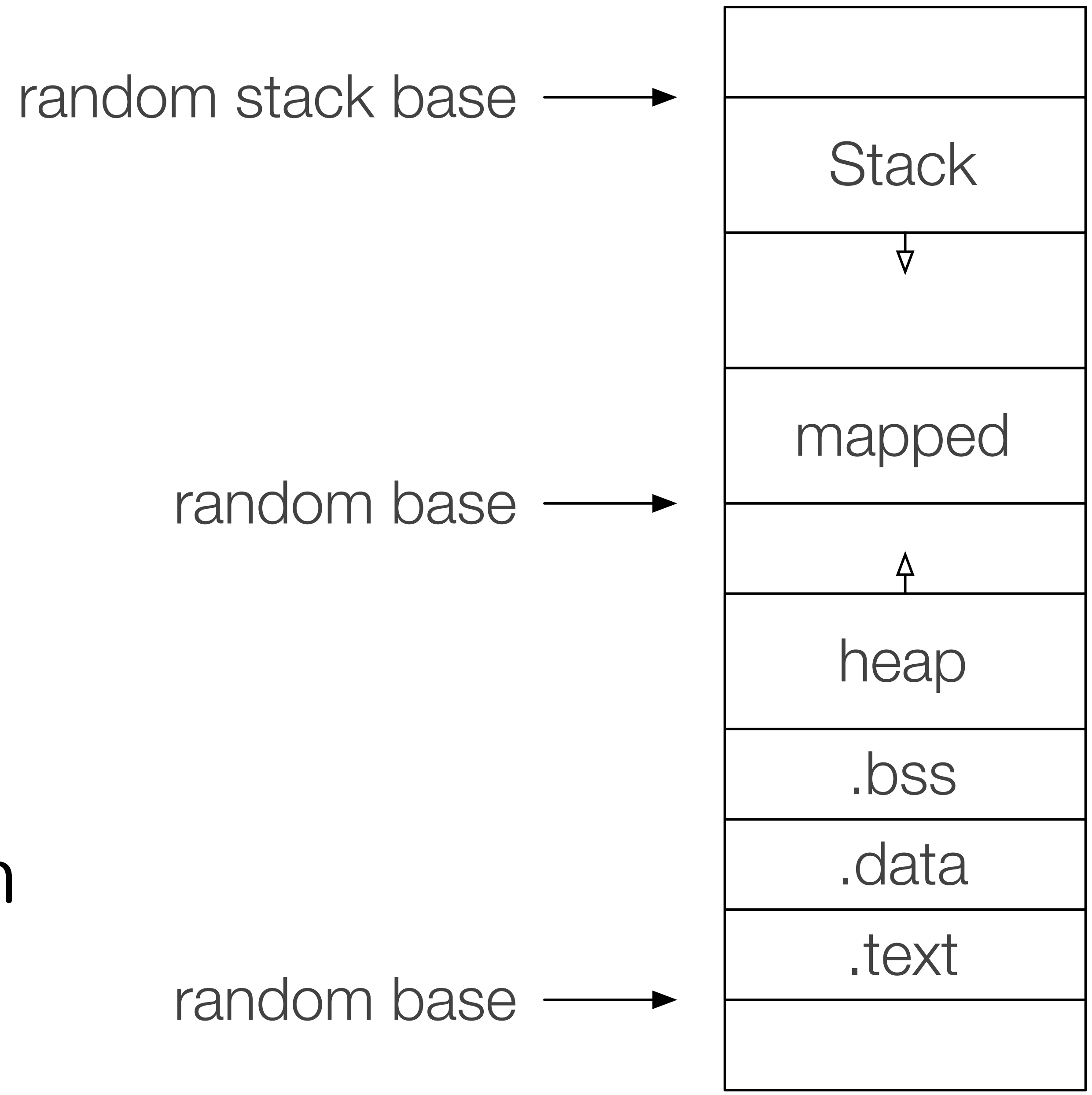
```
char buf[64];
…
strcpy(buf, s); // overflow
```

Defense assumptions

- W^X enabled
- PaX ASLR enabled
  - Apache forks child processes to handle client interaction
  - How does re-randomization work?

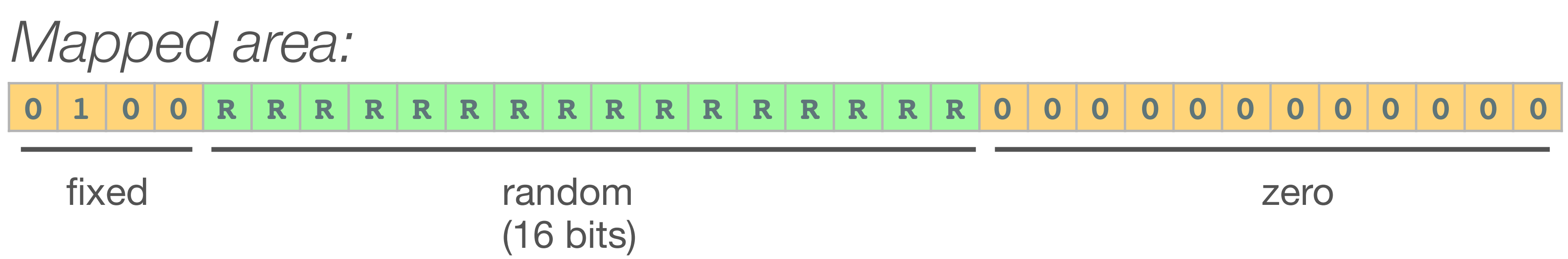
Planning the Attack

- Can we inject shell code on the stack?
  - A: yes, **B: no**
- Call system in libc
  - Located in mapped region



Derandomizing ASLR

- **Stage 1:** Find base of mapped region



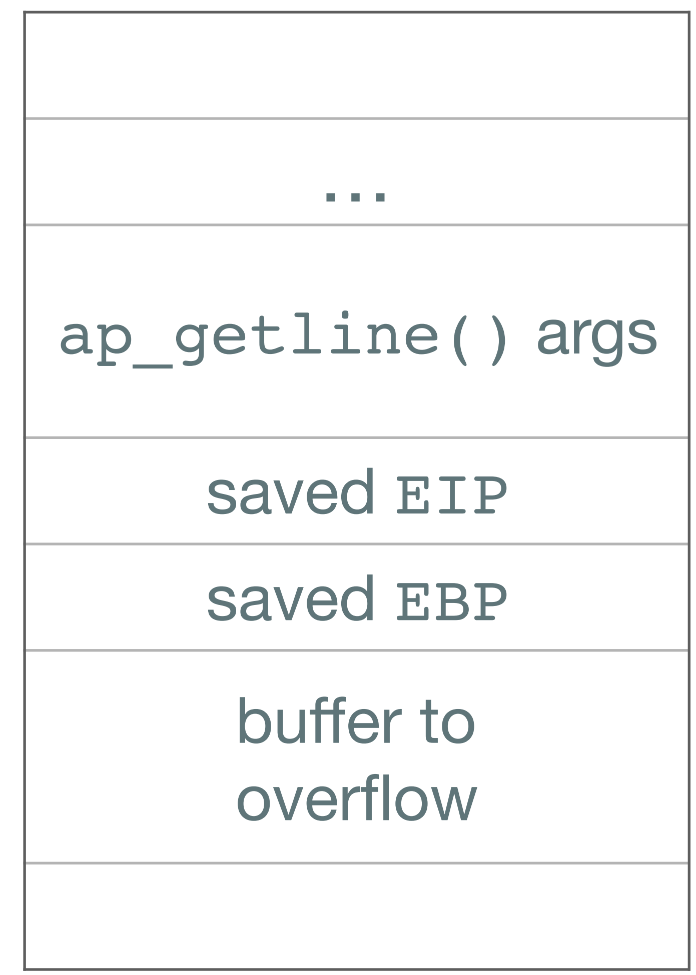
- **Stage 2:** Call system() with command string

Finding base of mapped region

- Overflow buffer in ap_getline()
- Overwrite saved EIP with guessed location of usleep

Finding base of mapped region

- Overflow buffer in ap_getline()
- Overwrite saved EIP with guessed location of usleep
  - Base + offset of usleep in mapped region
- Provide non-zero byte argument to usleep()



Finding base of mapped region

- If we guessed usleep() address right
  -
- If we guessed usleep() address wrong
  -
- Use this to tell if we guessed base of mapped region correctly

Finding base of mapped region

- If we guessed usleep() address right
  - Server will freeze for 16 seconds, then crash
- If we guessed usleep() address wrong
  -
- Use this to tell if we guessed base of mapped region correctly

Finding base of mapped region

- If we guessed usleep() address right
  - Server will freeze for 16 seconds, then crash
- If we guessed usleep() address wrong
  - Server will (likely) crash immediately
- Use this to tell if we guessed base of mapped region correctly

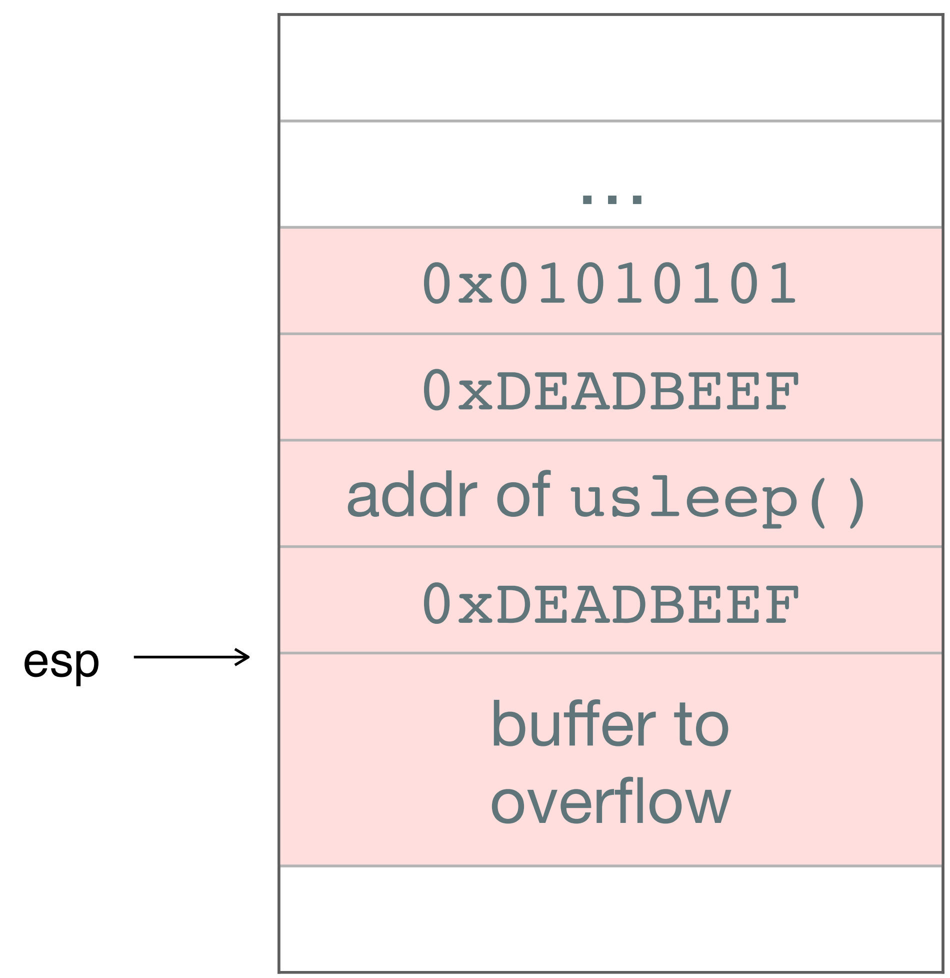
Finding base of mapped region

```
ap_getline:
…
pop ebp
ret

usleep:
…
ret
```

Finding base of mapped region

eip

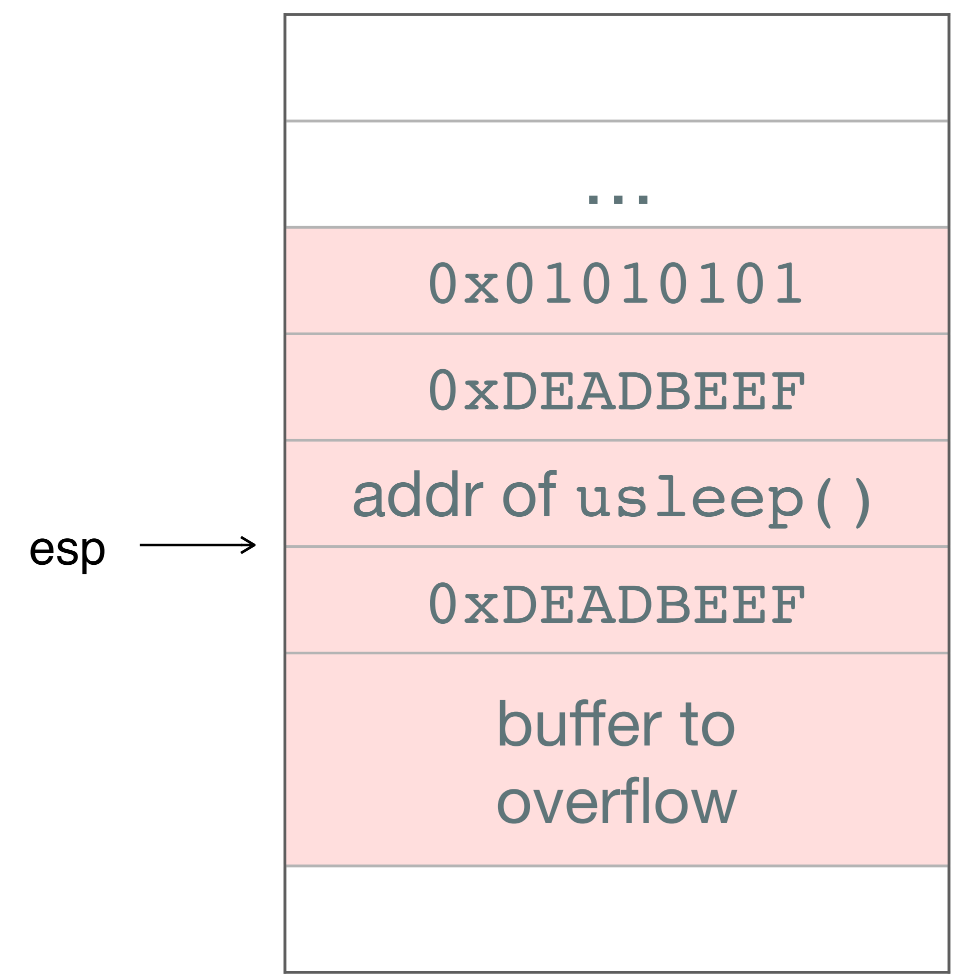
```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```



Finding base of mapped region

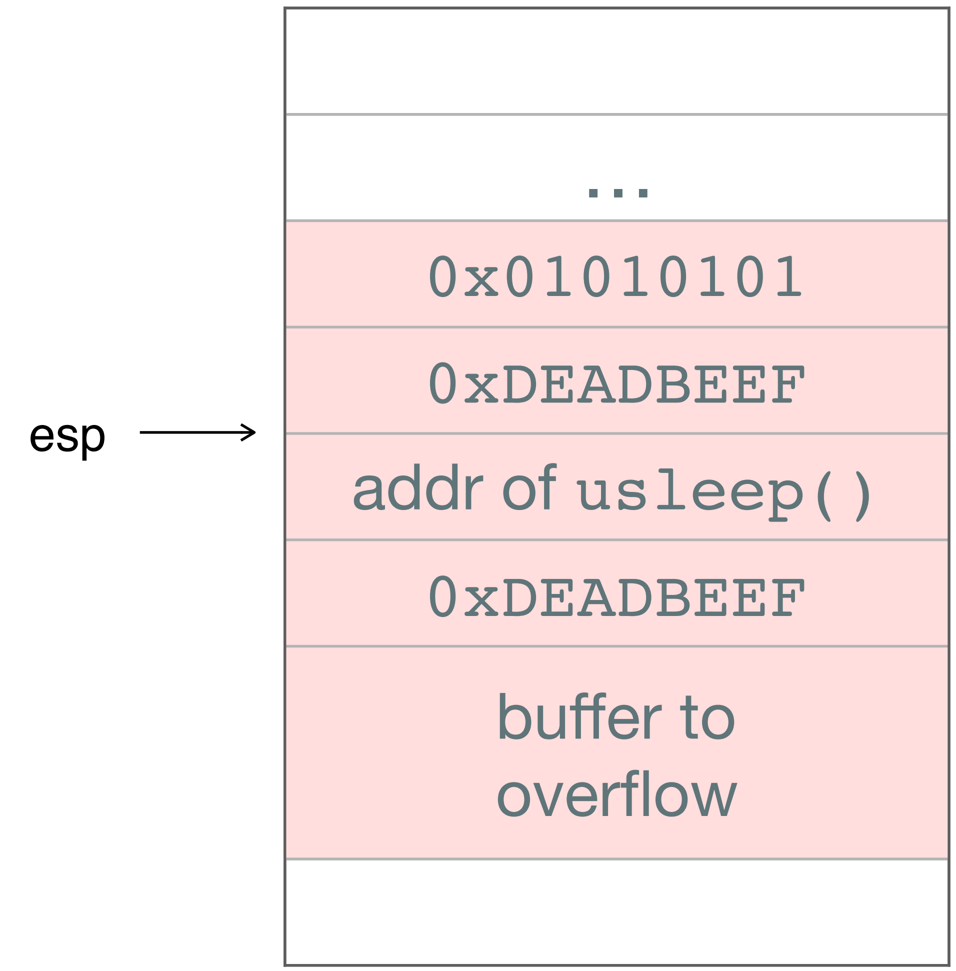
eip

```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```



Finding base of mapped region

```
ap_getline:
...
pop ebp
eip ⟶ ret

usleep:
...
ret
```

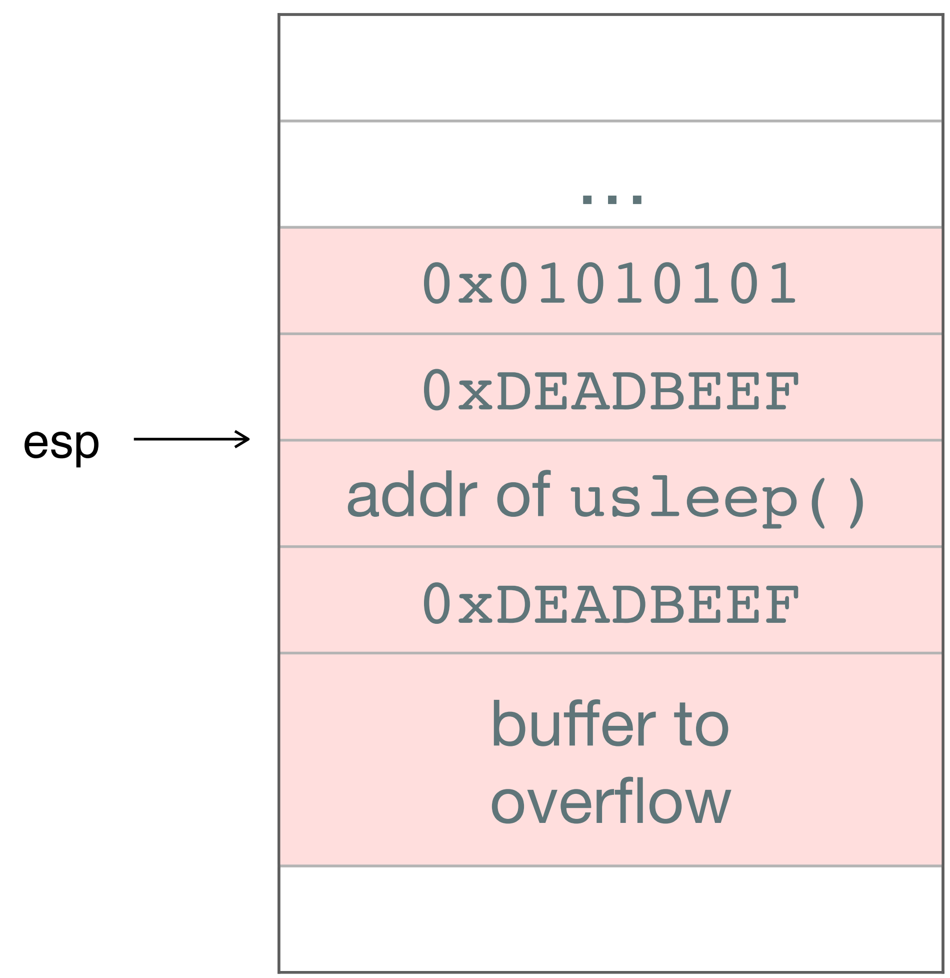


Finding base of mapped region

```
ap_getline:
...
pop ebp
ret
```

eip

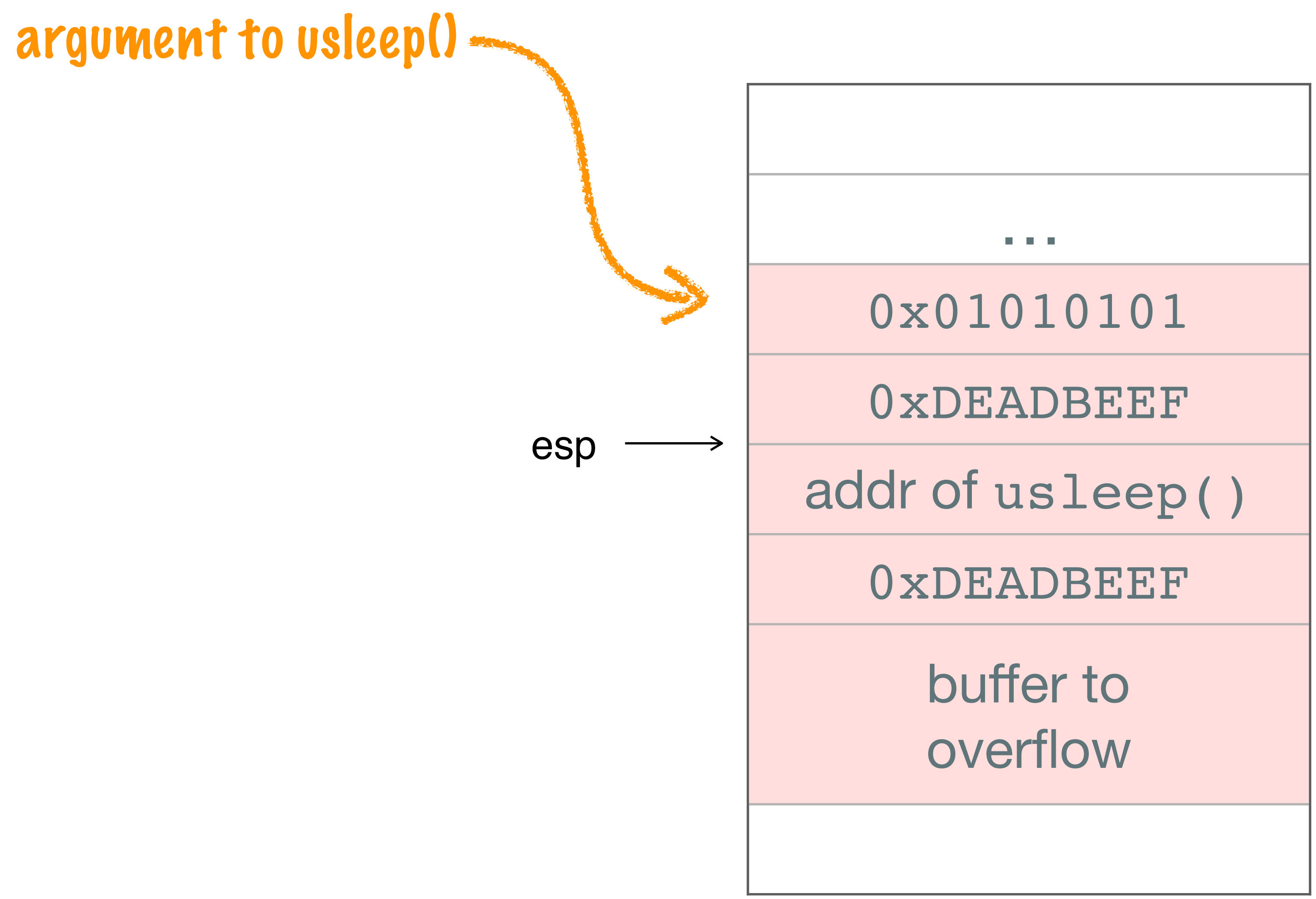
```
usleep:
...
ret
```



Finding base of mapped region

```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```

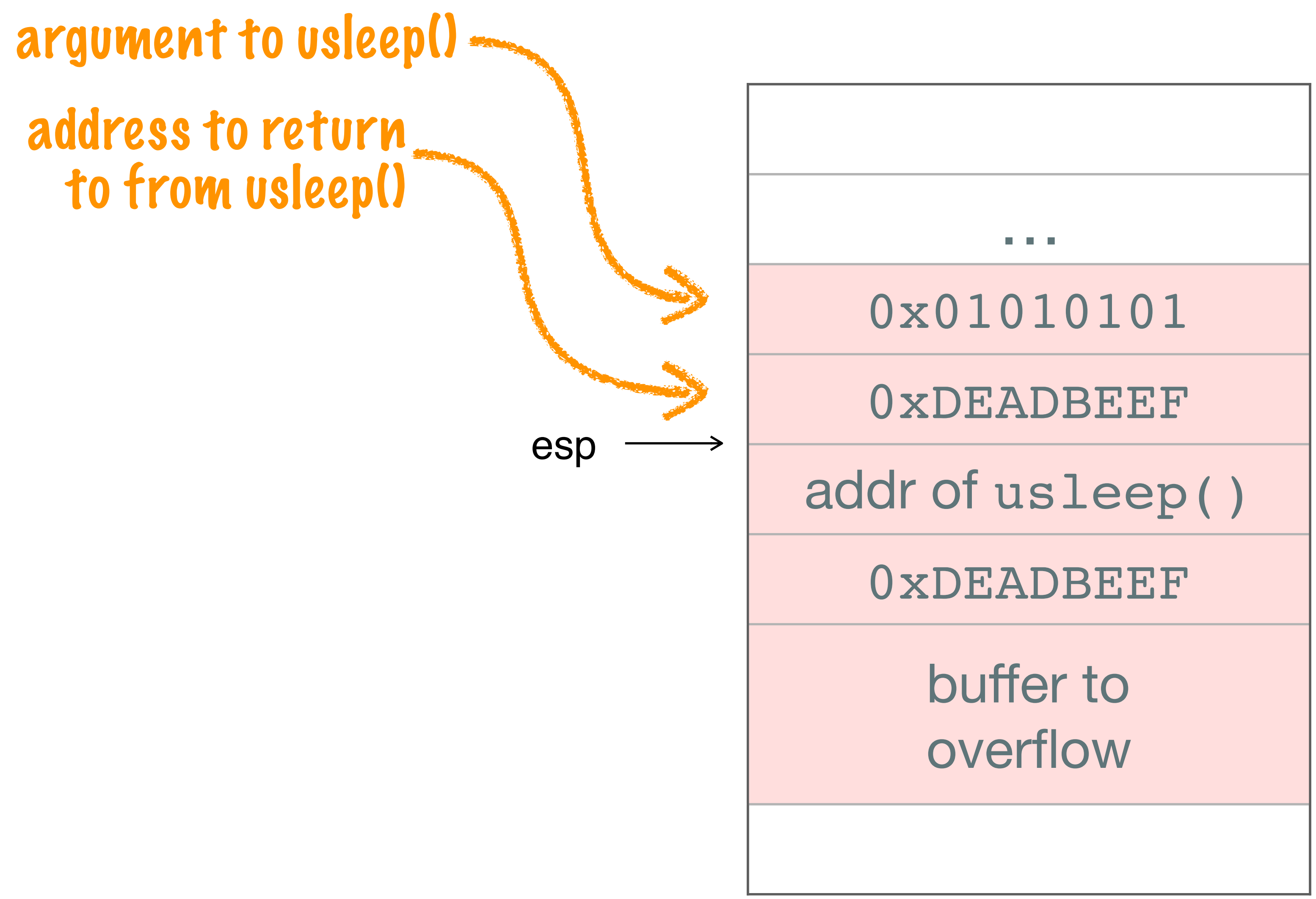
eip →



Finding base of mapped region

```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```

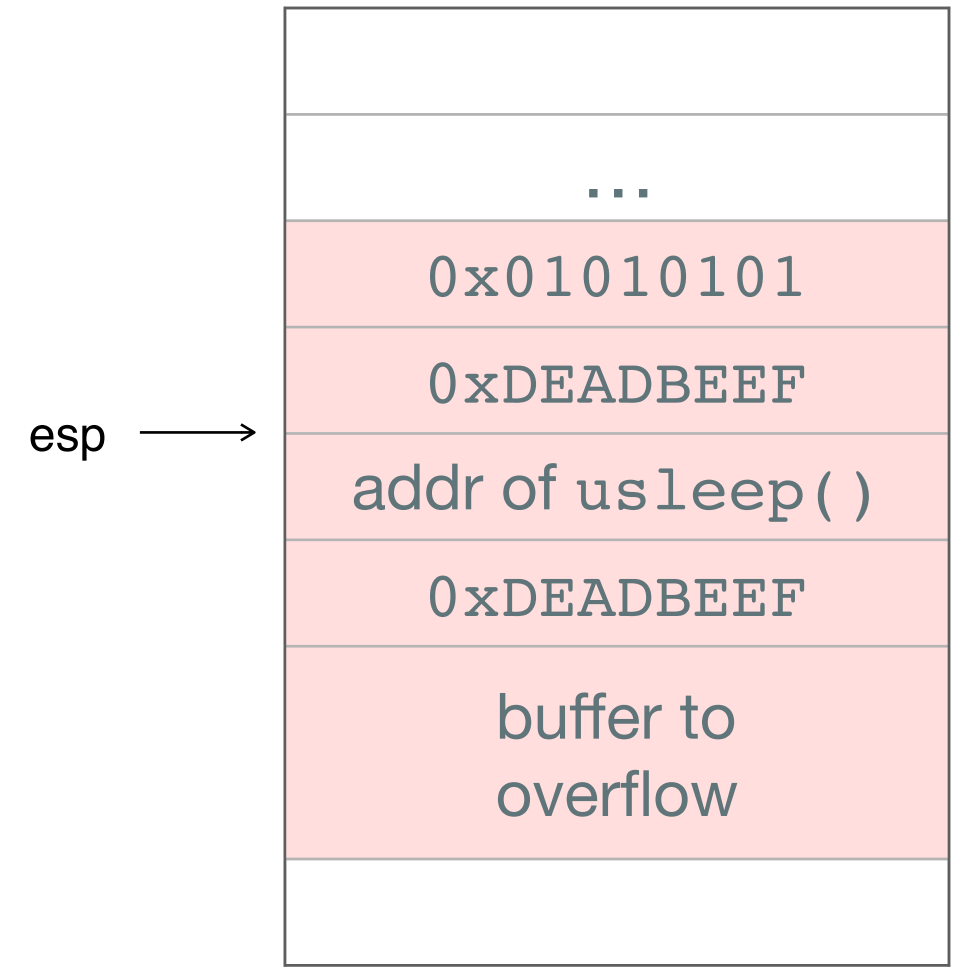
eip → usleep:



Finding base of mapped region

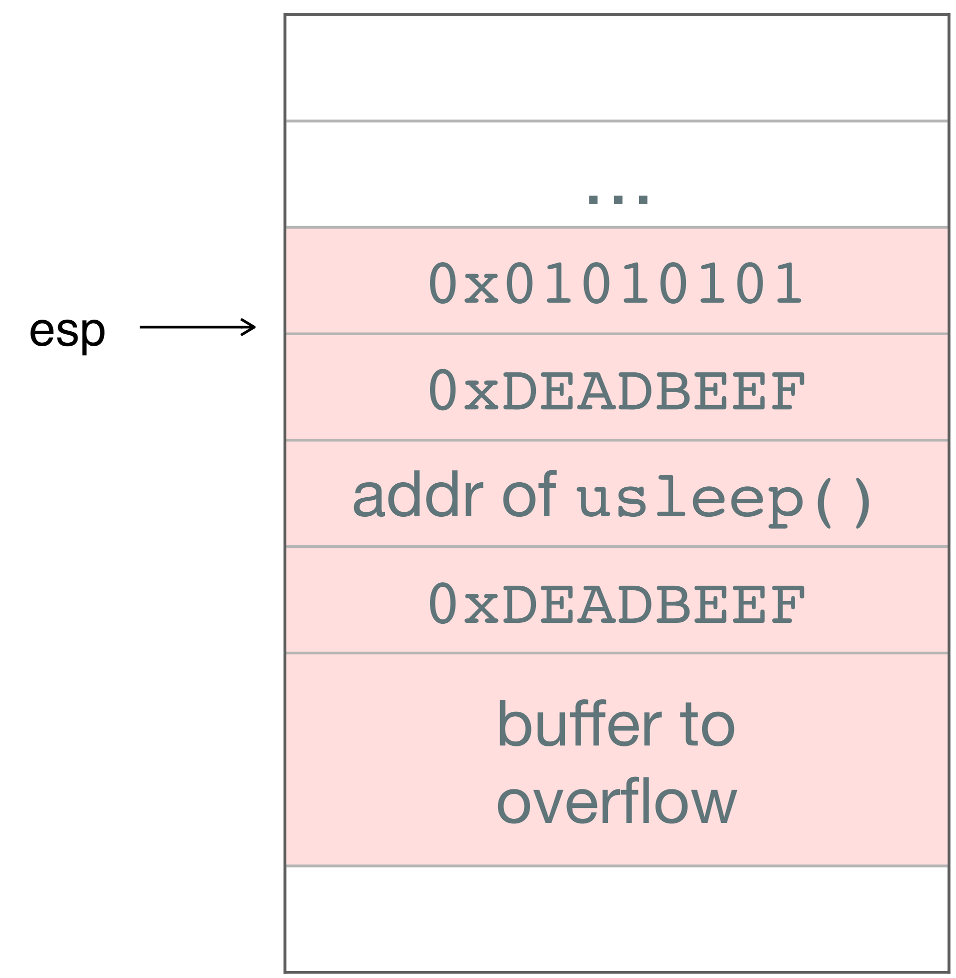
```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```

eip



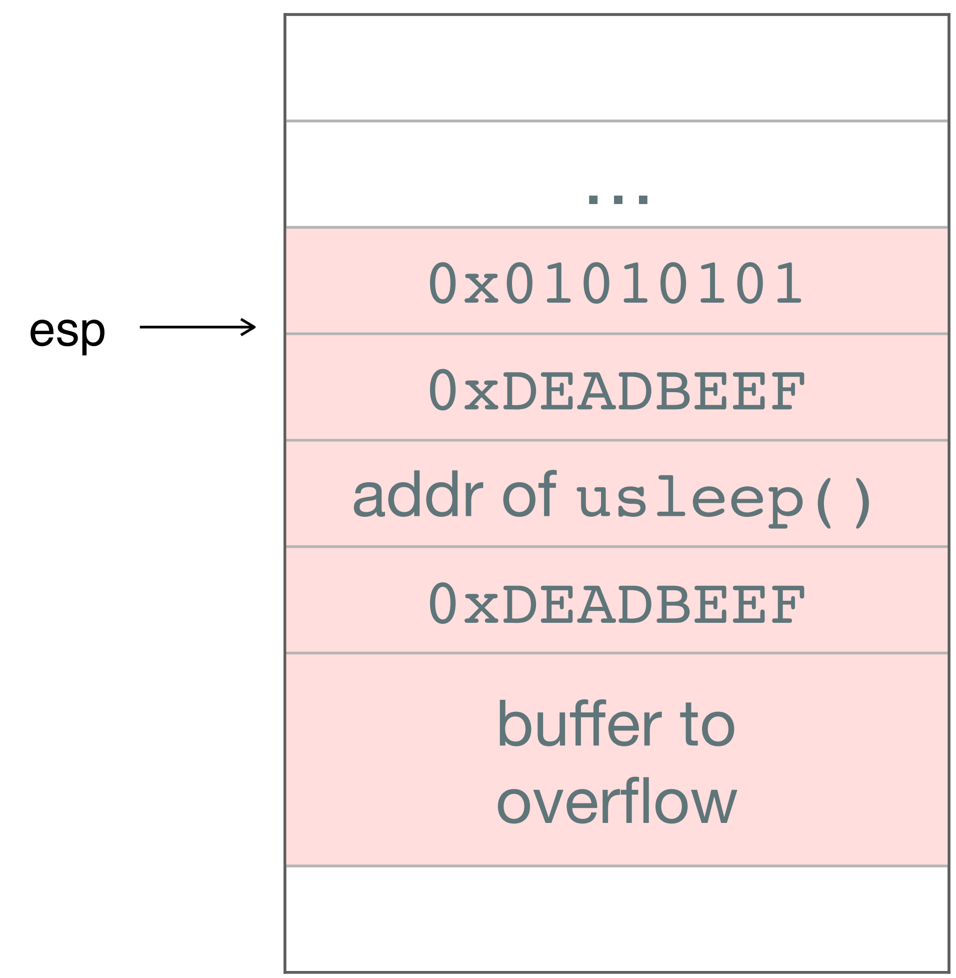
Finding base of mapped region

```
      ap_getline:
      ...
      pop ebp
      ret

      usleep:
      ...
eip → ret
```



Finding base of mapped region

```
ap_getline:
...
pop ebp
ret

usleep:
...
ret
```

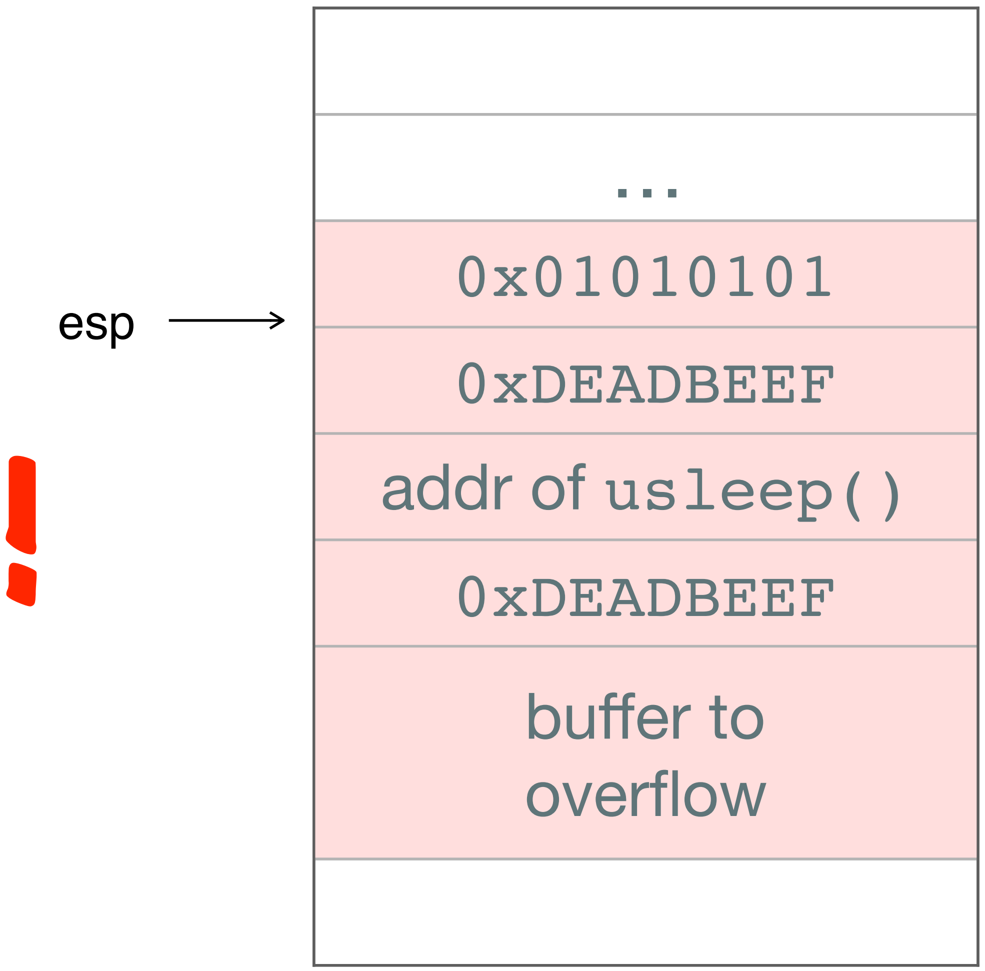


eip → `0xDEADBEEF`

Finding base of mapped region

```
ap_getline:
…
pop ebp
ret

usleep:
…
ret
```

SEGFAULT!



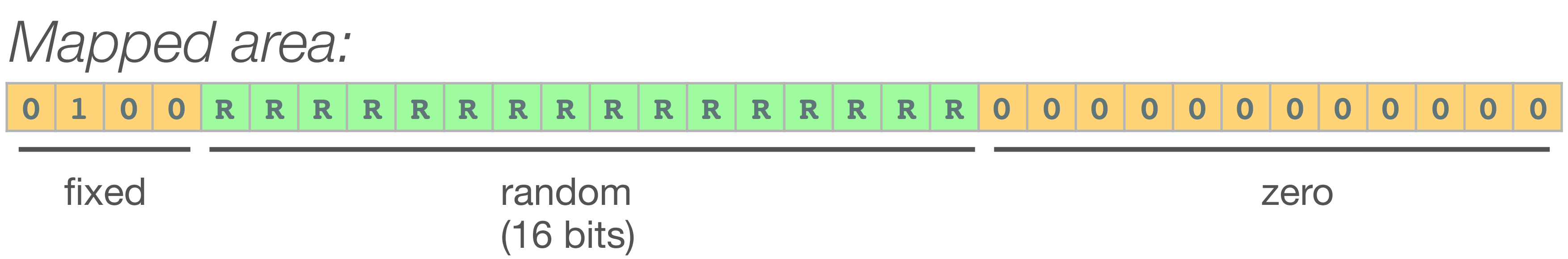
eip → 0xDEADBEEF

Derandomizing ASLR

- What is the success probability?
  -
- Do we need to derandomize the stack base?
  - A: yes, **B: no**
- Attack works even with PaX ASLR and DEP

Derandomizing ASLR

- **Stage 1:** Find base of mapped region



- **Stage 2:** Call system() with command string

Stage 2

- Overflow buffer in ap_getline()
- Pointer to buffer is a local in ap_getline
  - Overwrite saved EIP with address of (any) ret instruction in libc
  - Repeat until address of attack command string on the stack
- Append address of system()

Stage 2

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

Stage 2

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

Stage 2

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

top of stack (higher addresses)
⋮
pointer into 64 byte buffer
`0xdeadbeef`
address of `system()`
address of `ret` instruction
⋮
address of `ret` instruction
`0xdeadbeef`
64 byte buffer (contains shell commands)
⋮
bottom of stack (lower addresses)

Stage 2

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

already on stack, adjust esp (w/ rets) to make it look like arg to system()

top of stack (higher addresses)
⋮
pointer into 64 byte buffer
`0xdeadbeef`
address of `system()`
address of `ret` instruction
⋮
address of `ret` instruction
`0xdeadbeef`
64 byte buffer (contains shell commands)
⋮
bottom of stack (lower addresses)

Summary: return to libc

- If stack not executable, what can we do?
  - Use existing program code! return-to-libc
- Search executable for code
  - E.g. if executable calls exec("/bin/sh"), jump there
- Need known executable
  - Usually not a problem, can work around this


Employees must
wash hands before
returning to libc

What if there is no code that does what we want?

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

Hovav Shacham*
hovav@cs.ucsd.edu

Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved EIP on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  -
  -

Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved EIP on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function (inserted by compiler)
  -

Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved EIP on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function (inserted by compiler)
  - Any sequence of executable memory ending in 0xc3

x86 instructions

- Variable length!
- Can begin on any byte boundary!

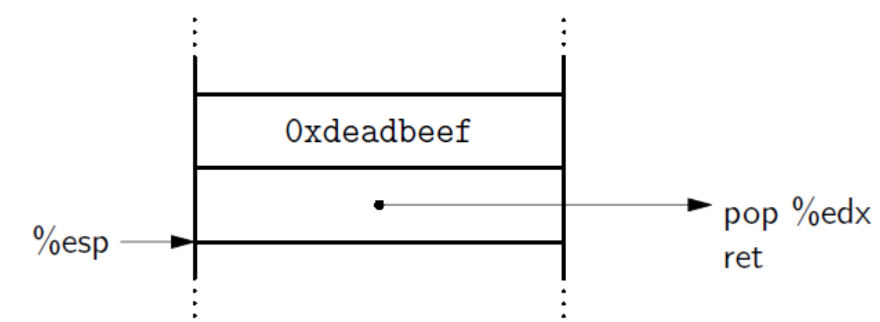
So?

- b8 01 00 00 00 5b c9 c3
  - mov eax,0x1 → pop ebx → leave → ret
- 00 00 5b c9 c3
  - add BYTE PTR [eax],al → pop ebx → leave → ret
- 00 5b c9 c3
  - add BYTE PTR [eax-0x37],bl → ret

```
compy% otool -t /bin/ls
/bin/ls:
(__TEXT,__text) section
0000000100001478 6a 00 48 89 e5 48 83 e4 f0 48 8b 7d 08 48 8d 75
0000000100001488 10 89 fa 83 c2 01 c1 e2 03 48 01 f2 48 89 d1 eb
0000000100001498 04 48 83 c1 08 48 83 39 00 75 f6 48 83 c1 08 e8
00000001000014a8 58 0f 00 00 89 c7 e8 1b 39 00 00 f4 55 48 89 e5
00000001000014b8 48 8d 47 68 48 8d 7e 68 48 89 c6 c9 e9 01 3a 00
00000001000014c8 00 55 48 89 e5 48 83 c6 68 48 83 c7 68 c9 e9 ef
00000001000014d8 39 00 00 55 48 89 e5 53 48 89 f1 48 8b 56 60 48
00000001000014e8 8b 47 60 48 8b 58 30 48 39 5a 30 7f 1d 7c 22 48
00000001000014f8 8b 58 38 48 39 5a 38 7f 11 7c 16 48 8d 77 68 48
0000000100001508 8d 79 68 5b c9 e9 b8 39 00 00 b8 ff ff ff ff eb
0000000100001518 05 b8 01 00 00 00 5b c9 c3 55 48 89 e5 48 8b 56
0000000100001528 60 48 8b 47 60 48 8b 48 50 48 39 4a 50 7f 1c 7c
0000000100001538 21 48 8b 48 58 48 39 4a 58 7f 10 7c 15 48 83 c6
0000000100001548 68 48 83 c7 68 c9 e9 77 39 00 00 b8 01 00 00 00
0000000100001558 eb 05 b8 ff ff ff ff c9 c3 55 48 89 e5 53 48 8b
0000000100001568 56 60 48 8b 47 60 b9 01 00 00 00 48 8b 58 60 48
0000000100001578 39 5a 60 7f 18 7d 07 b9 ff ff ff ff eb 0f 48 83
0000000100001588 c6 68 48 83 c7 68 5b c9 e9 35 39 00 00 89 c8 5b
0000000100001598 c9 c3 55 48 89 e5 48 8b 56 60 48 8b 47 60 48 8b
00000001000015a8 48 40 48 39 4a 40 7f 1c 7c 21 48 8b 48 48 48 39
00000001000015b8 4a 48 7f 10 7c 15 48 83 c6 68 48 83 c7 68 c9 e9
00000001000015c8 fe 38 00 00 b8 01 00 00 00 eb 05 b8 ff ff ff ff
```

What does a gadget look like?

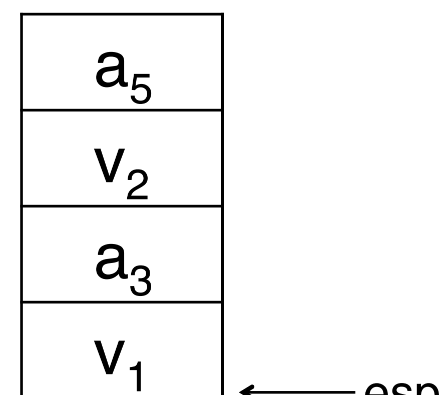
- Gadget for loading a constant
  - Arrange the constant to load to be just past the return address
  - Return to gadget that pops a value and returns.



What does this gadget do?

eip $\longrightarrow$ a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret

Code

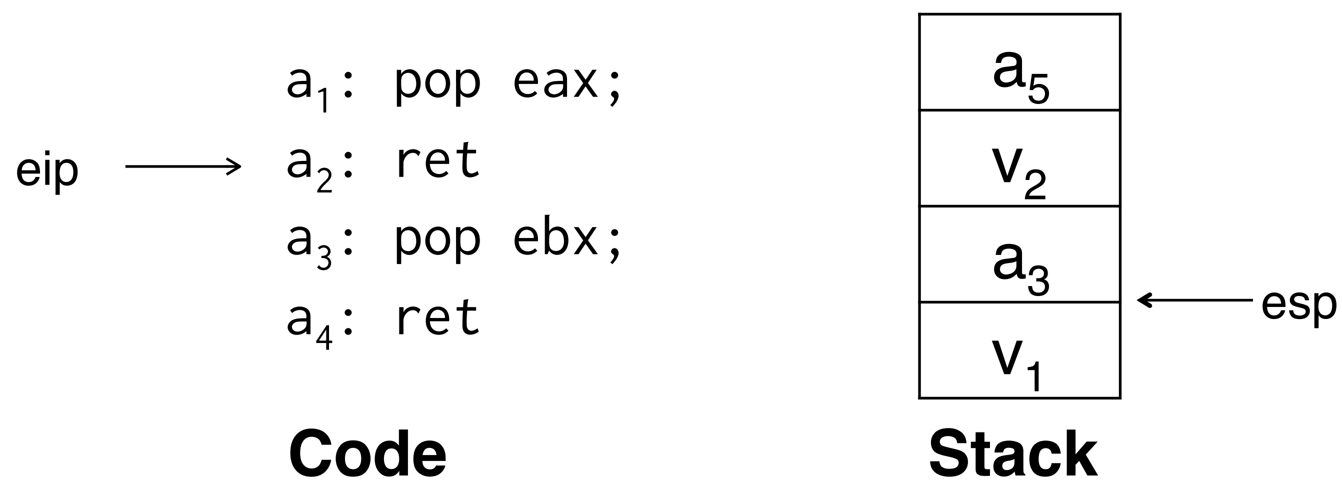


Stack

eax					
ebx					
eip	a_1				

time $\longrightarrow$

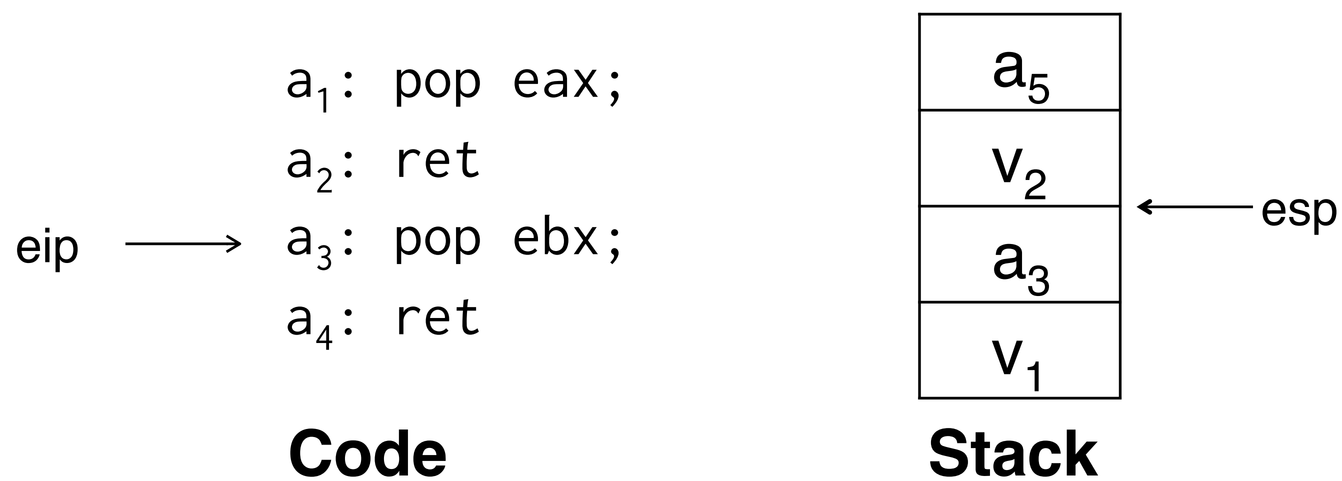
What does this gadget do?



eax		v_1			
ebx					
eip	a_1	a_2			

time →

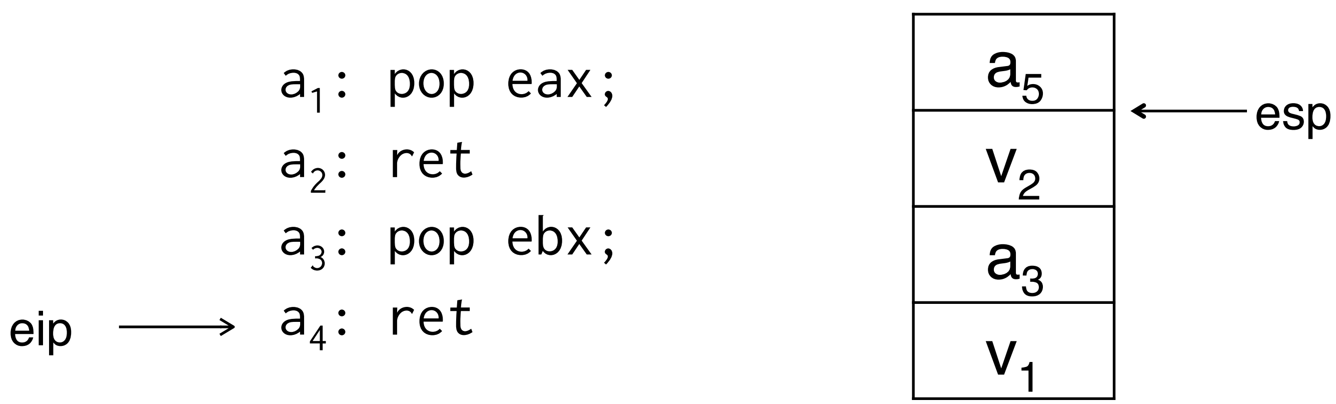
What does this gadget do?



eax		v_1	v_1		
ebx					
eip	a_1	a_2	a_3		

time ⟶

What does this gadget do?



Code **Stack**

eax		v_1	v_1	v_1	
ebx				v_2	
eip	a_1	a_2	a_3	a_4	

time →

What does this gadget do?

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
```

eip →

Code

a_5	← esp
v_2	
a_3	
v_1	

Stack

eax		v_1	v_1	v_1	v_1
ebx				v_2	v_2
eip	a_1	a_2	a_3	a_4	a_5

time →

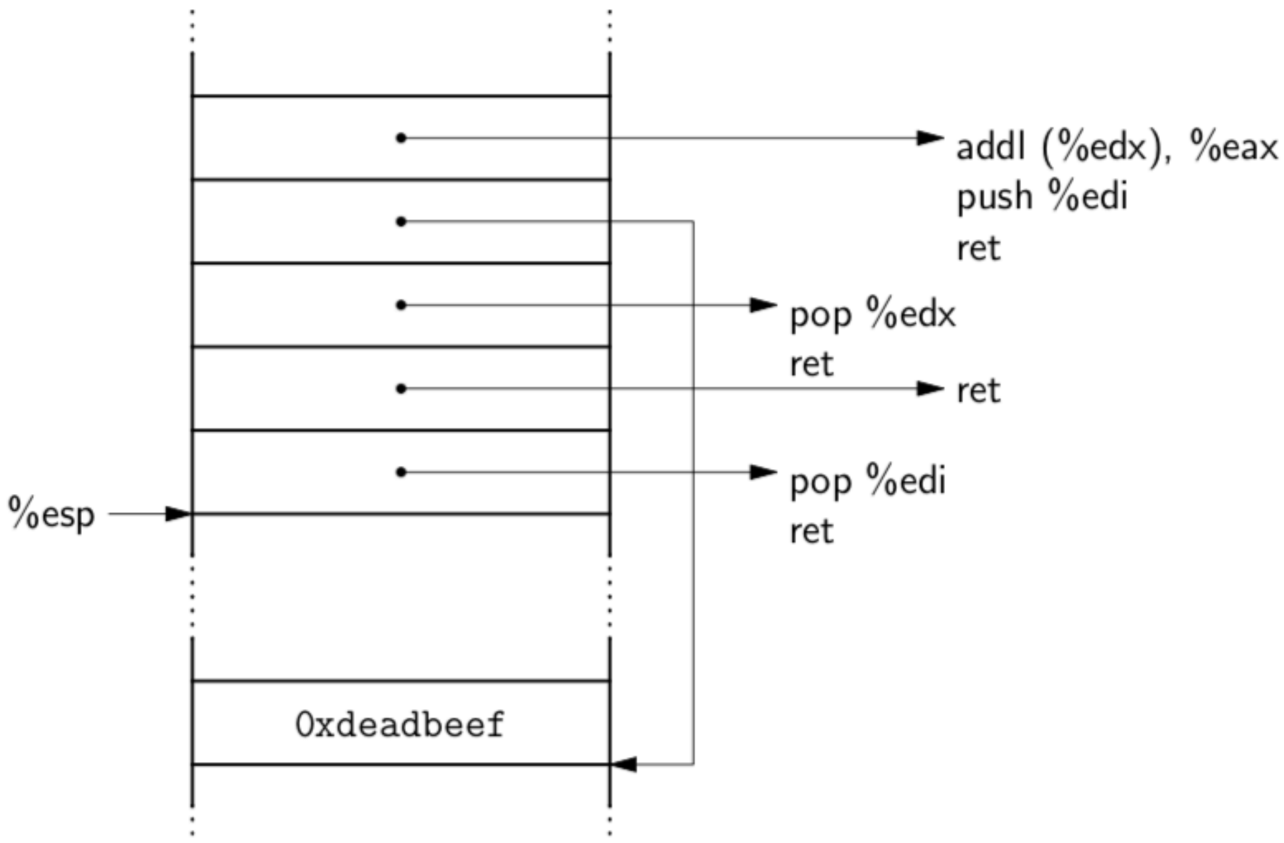


Figure 5: Simple add into %eax.

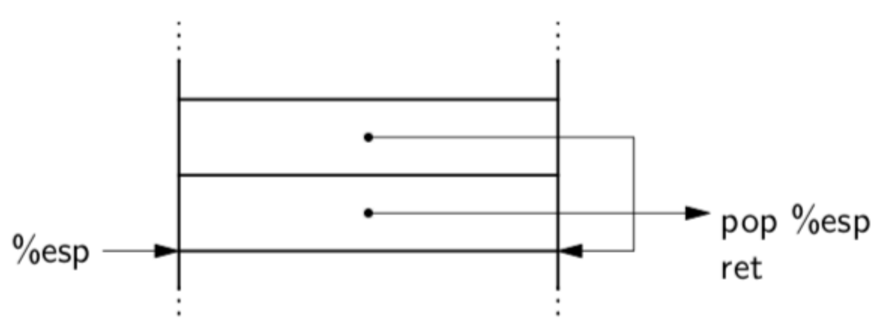


Figure 10: An infinite loop by means of an unconditional jump.

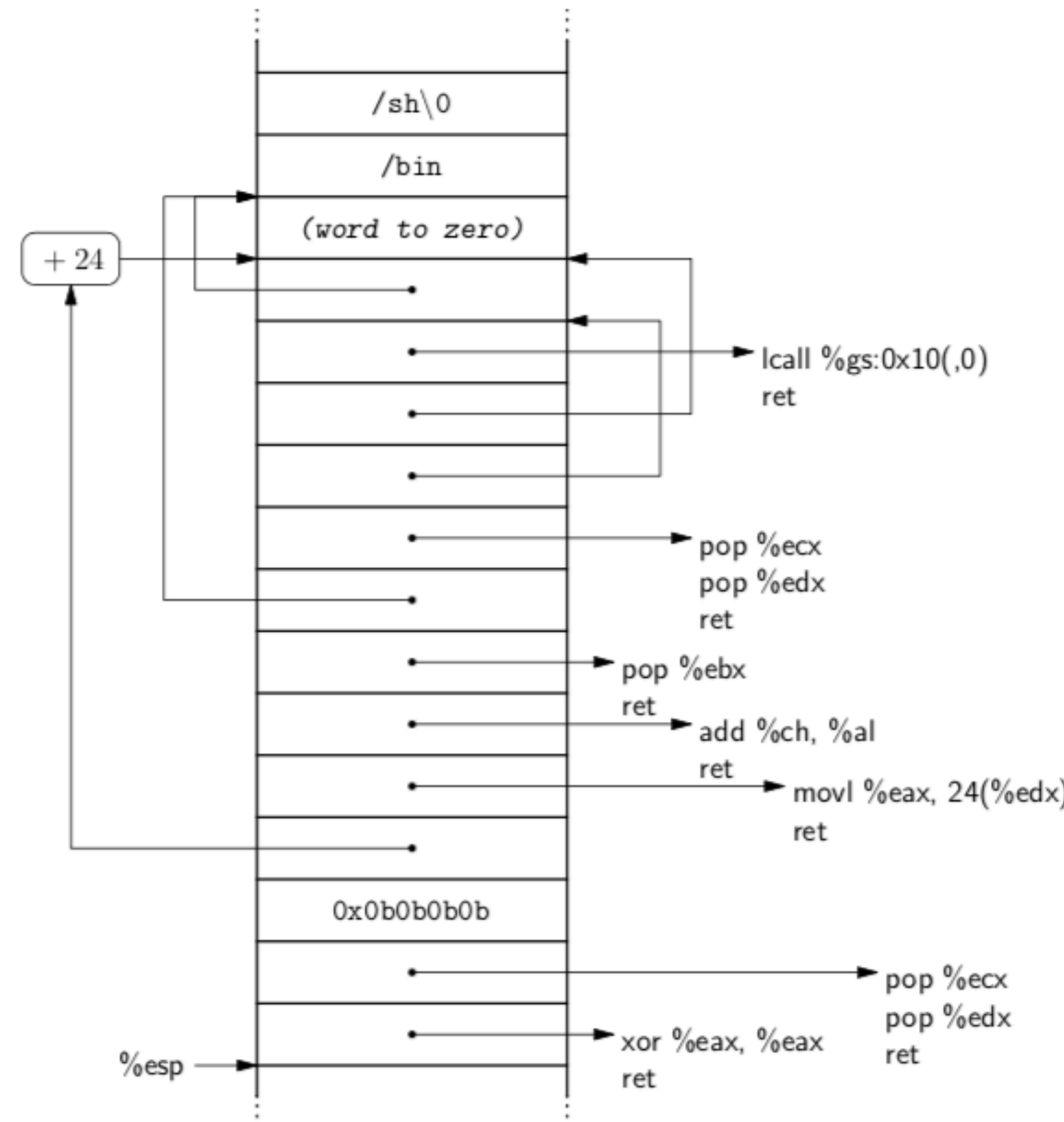


Figure 16: Shellcode.

	"Normal"	Return-oriented
Instruction pointer	eip	
No-op	nop	
Unconditional jump	jmp address	
Conditional jump	jnz address	set esp to address of gadget if some condition is met; ret
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

	"Normal"	Return-oriented
Instruction pointer	eip	esp
No-op	nop	
Unconditional jump	jmp address	
Conditional jump	jnz address	set esp to address of gadget if some condition is met; ret
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

	"Normal"	Return-oriented
Instruction pointer	eip	esp
No-op	nop	ret
Unconditional jump	jmp address	
Conditional jump	jnz address	set esp to address of gadget if some condition is met; ret
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

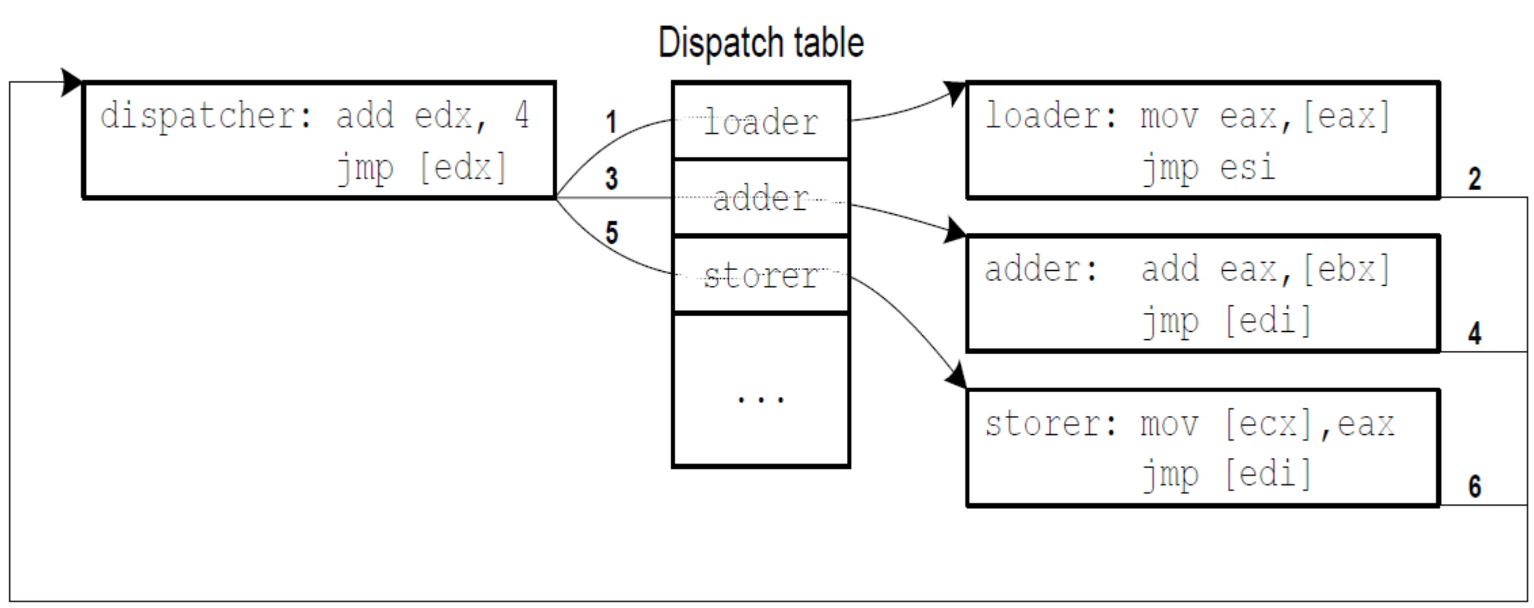
	"Normal"	Return-oriented
Instruction pointer	eip	esp
No-op	nop	ret
Unconditional jump	jmp address	set esp to addr of gadget; ret
Conditional jump	jnz address	set esp to address of gadget if some condition is met; ret
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

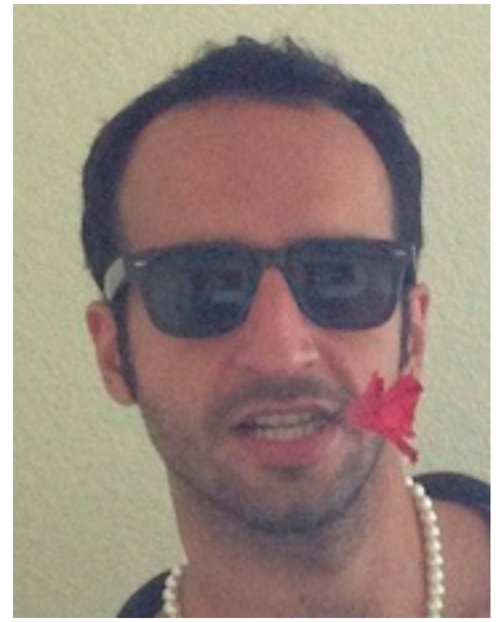
Return-Oriented Programming

not even really about “returns”…

Jump-Oriented Programming

- Identify gadgets ending in indirect jumps.
- Use a “dispatcher gadget” to combine them.
- Dispatch table used in place of stack





Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University