Memory Protection

CMSC 23200/33250, Winter 2023, Lecture 5

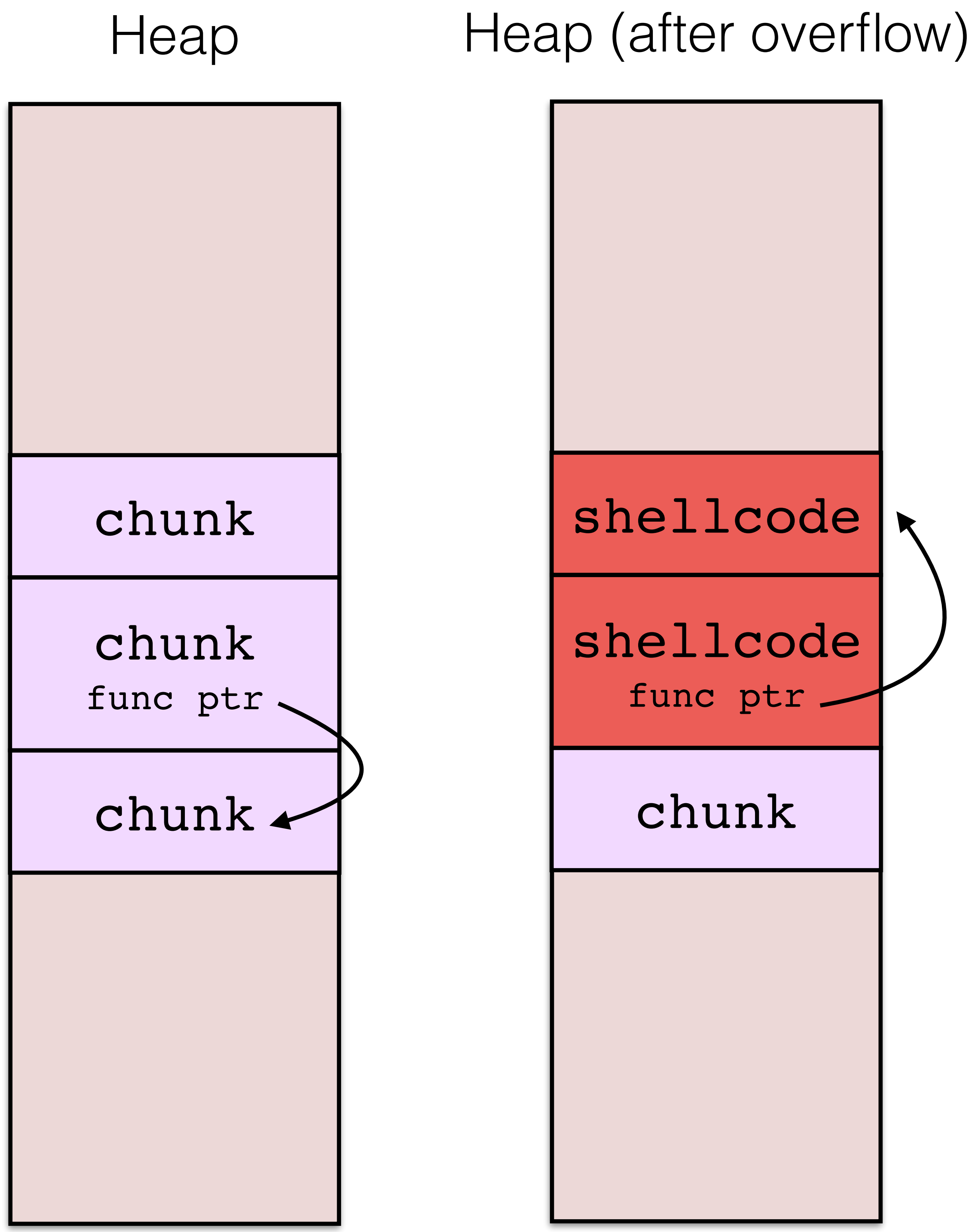
David Cash and Blase Ur

University of Chicago

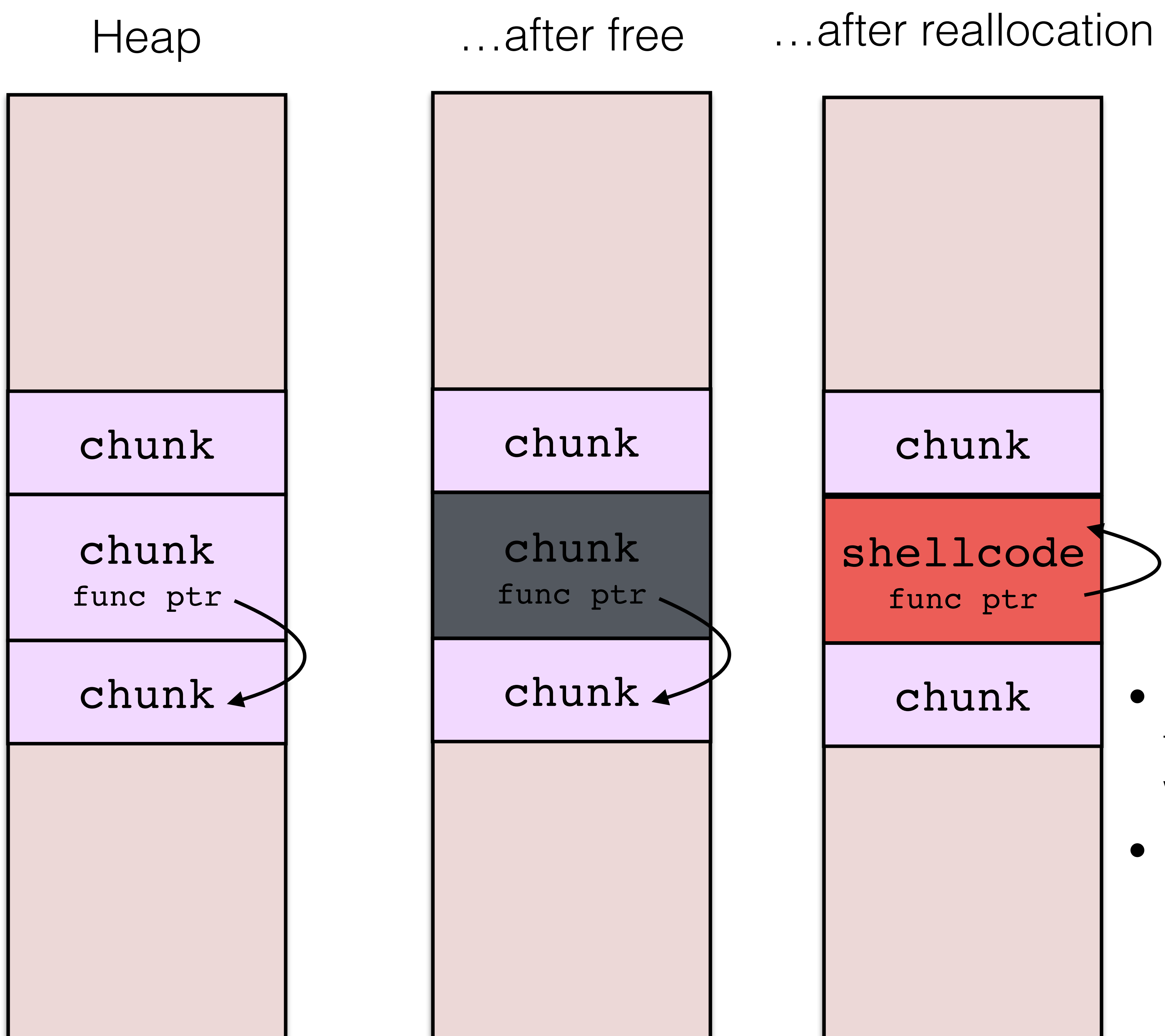
Outline of Lecture 5: Buffer Overflow Countermeasures

1. Heap vulnerabilities (briefly)
2. Stack Protectors
3. Address-Space Layout Randomization
4. W ^ X and ROP

Heap Memory Vulnerabilities: Overflows



Heap Memory Vulnerabilities: Use-after-free



- Dangling reference to free-ed chunk will use wrong pointer!
- Many other heap bugs: Double Free, corrupting metadata…

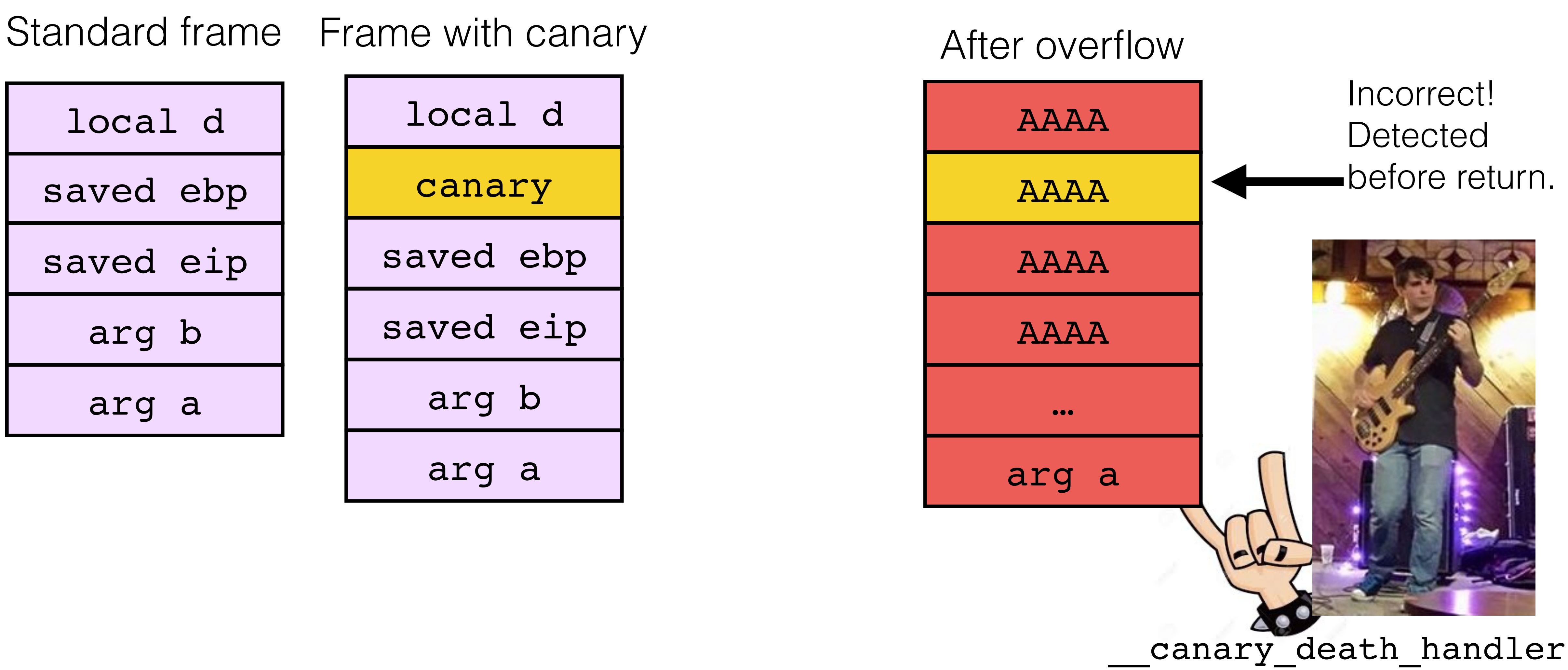
Outline of Lecture 5: Buffer Overflow Countermeasures

1. Heap vulnerabilities (briefly)
2. **Stack Protectors**
3. Address-Space Layout Randomization
4. W ^ X and ROP

Countermeasure #1: Stack Canaries

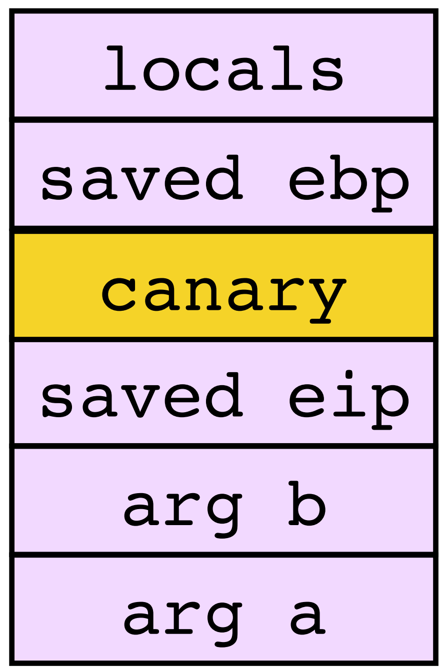
Stack Canaries (a.k.a. Stack Protectors)

- Compiler inserts instructions to each function:
  - At start of function, push a “canary” value onto stack between local variables and saved ebp/eip
  - Before returning, check if canary value is still correct; If not, ABORT.

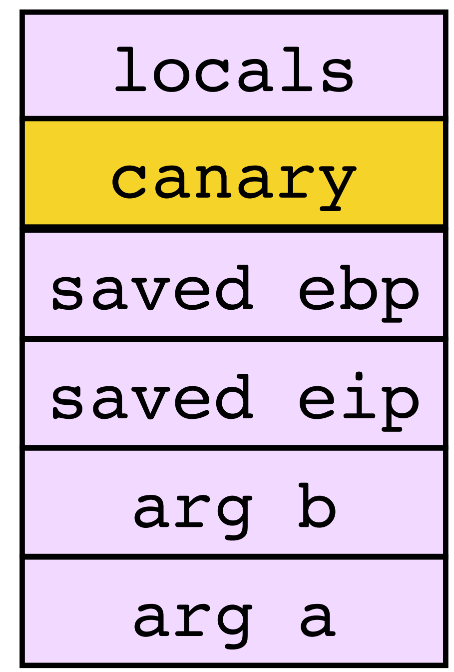


Where to put canaries?

StackGuard (1998)

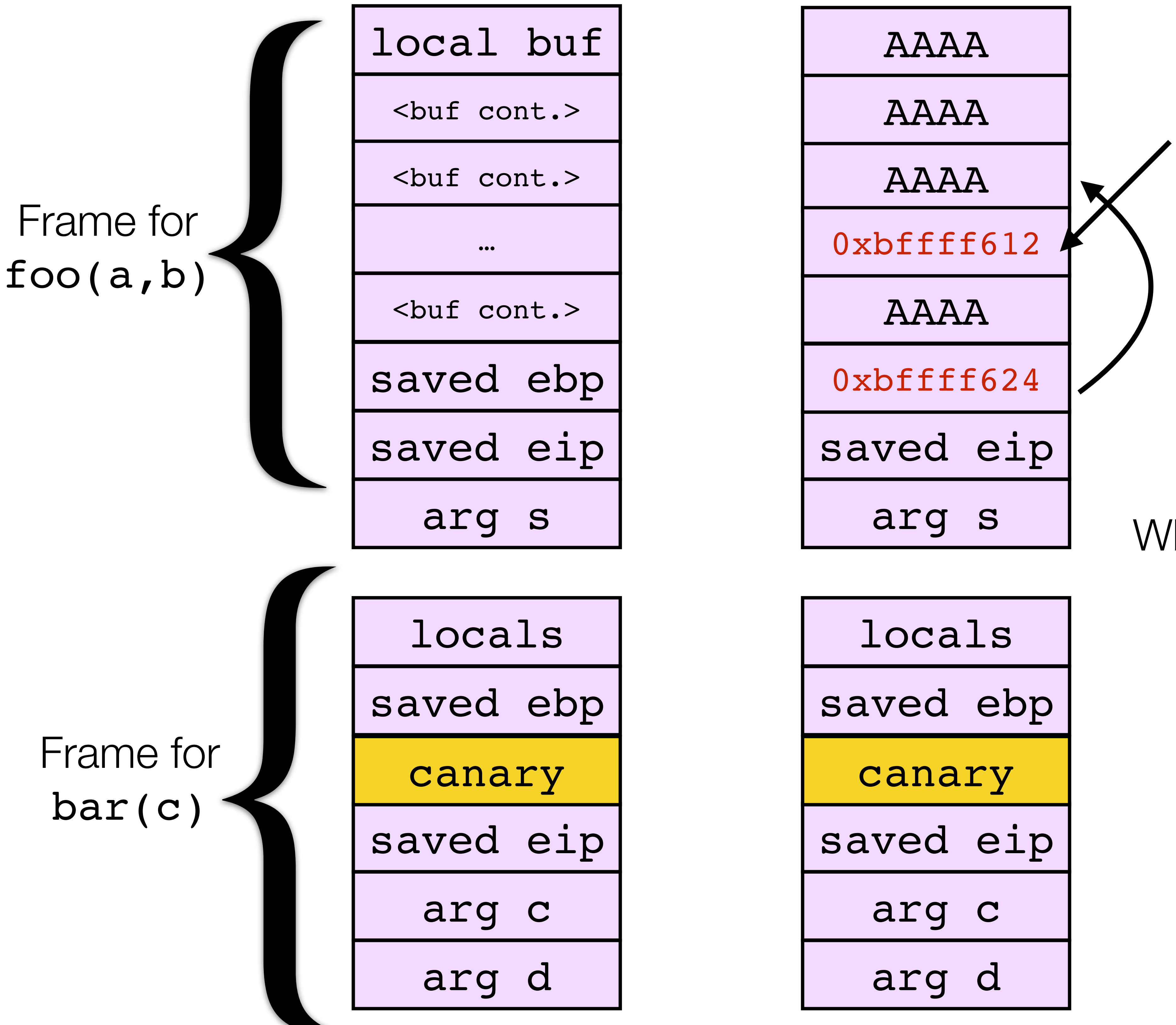


ProPolice (IBM, 2001-2005)



- Manipulating **`ebp`** (frame pointer) is almost as bad as **`eip`** (return address)!

Overwriting the ebp?



When returning from `bar()`, program thinks saved `eip` is the word after `ebp`!

After returning from `foo()`, `ebp` points to adversary input

What will happen?

- Return from foo to bar will happen cleanly
- Program will be mistaken about location of frame for `bar()`
- Program will find *next* return address in this mistaken frame

How should we pick the canary value?

Null: Set to `0x00000000`. Hard for attacker to copy NULLs onto stack.

Terminator: `0x000d0aff` (for example.) `0x0d=CR`, `0x0a=LF`, `0xff=EOF`. Some buggy code will stop at these characters.

Random: Process chooses random value at start, uses same value in every call.

Random XOR: Choose random value as above, but set canary to XOR of value and return address (or other info).

Stack Canaries in gcc

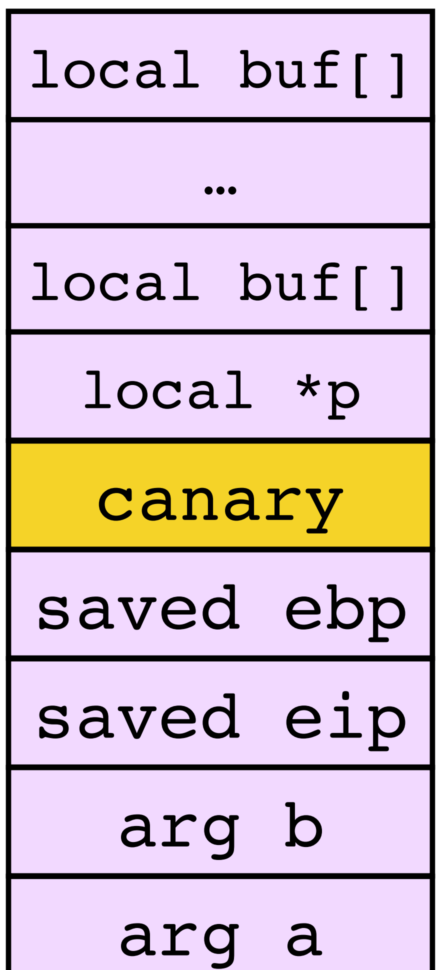
Flag	Default?	Notes
-fno-stack-protector	No	Turns off protections
-fstack-protector	Yes	Adds to funcs that call alloca() & w/ arrays larger than 8 chars (--param=ssp-buffer-size changes 8)
-fstack-protector-strong	No	Also funcs w/ any arrays & refs to local frame addresses. Introduced by ChromeOS team.
-fstack-protector-all	No	All funcs

- With -fstack-protector, 2.5% of functions in kernel covered, 0.33% larger binary
- With -fstack-protector-strong, 20.5% of functions in kernel covered, 2.4% larger binary

Related ProPolice Feature: Rearranging Locals

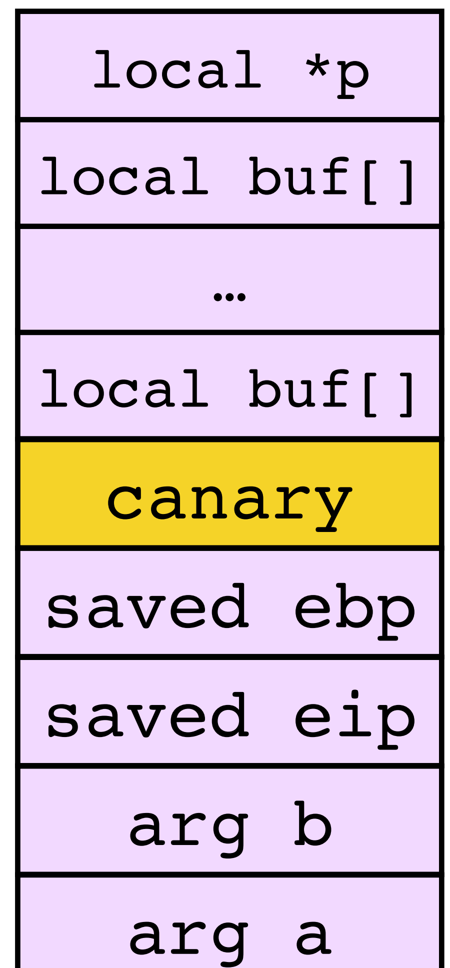
- gcc puts local arrays below other locals, even if declared in other order

```
int foo(…) {
  char *p;
  char buf[64];
  …
}
```

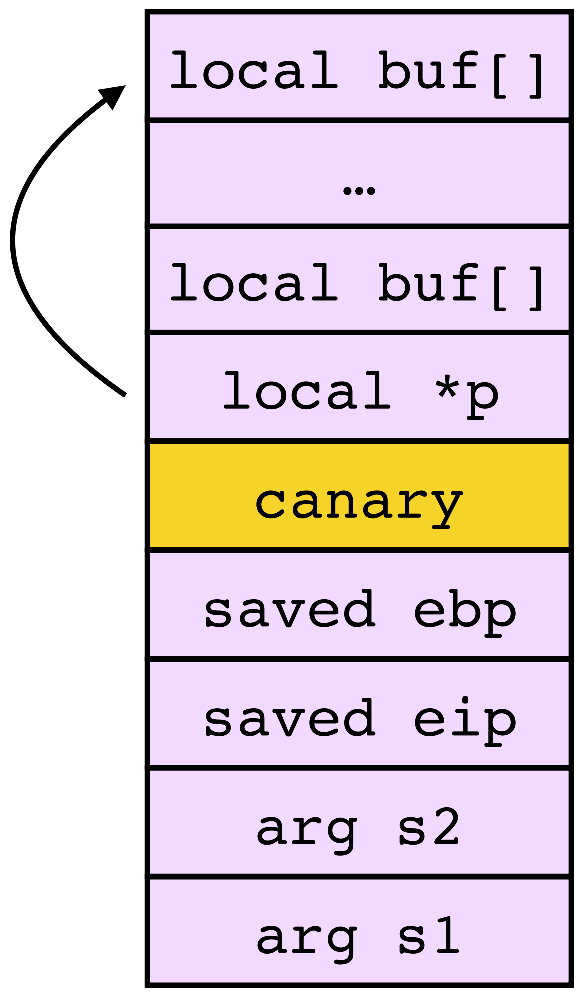


vs

```
int foo(…) {
  char buf[64];
  char *p;
  …
}
```

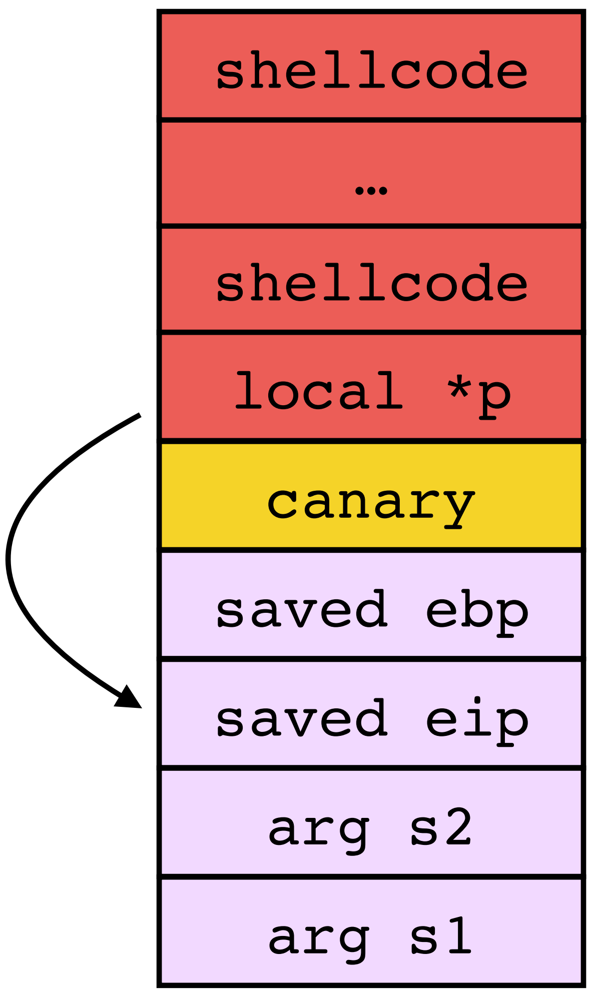


Bypassing Canaries via Complex Bugs



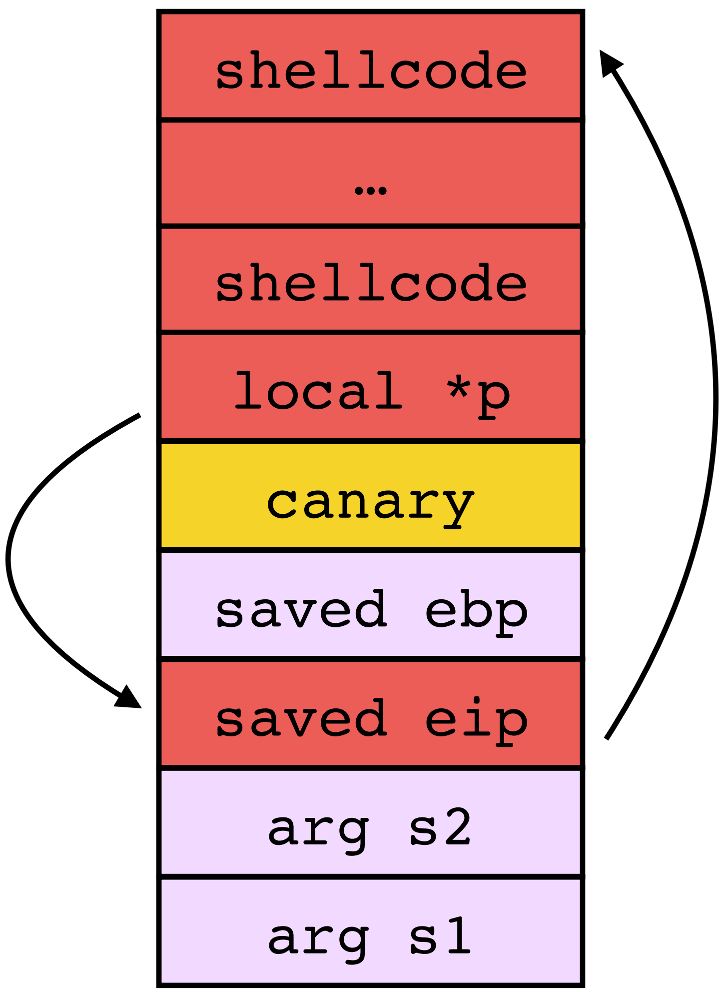

```
int foo(char *s1, char *s2) {
  char *p;
  char buf[64];
  p = buf;
  strcpy(p, s1); // oh no :(
  …
  strncpy(p, s2, 16);
  …
}
```

Bypassing Canaries via Complex Bugs

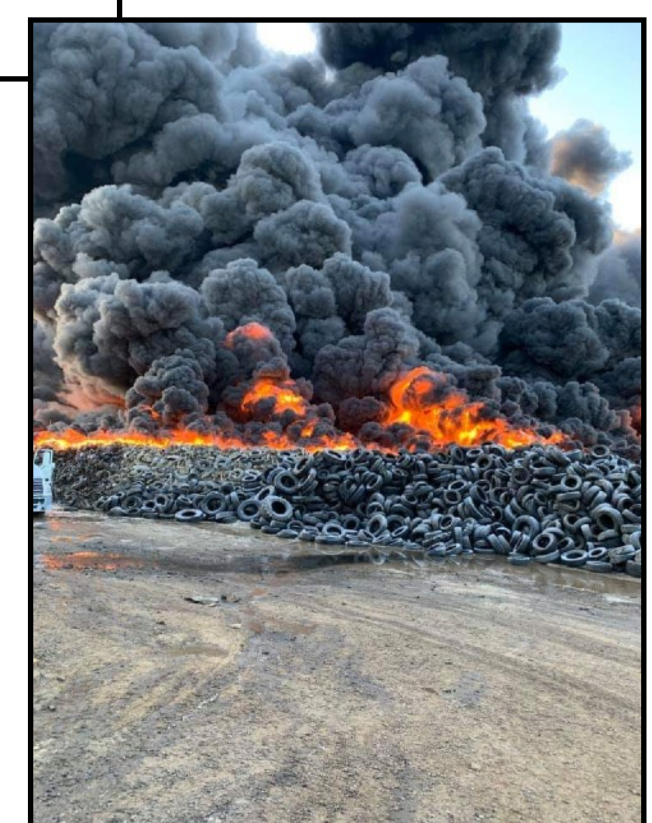



```
int foo(char *s1, char *s2) {
  char *p;
  char buf[64];
  p = buf;
  strcpy(p, s1); // oh no :(
  …
  strncpy(p, s2, 16);
  …
}
```

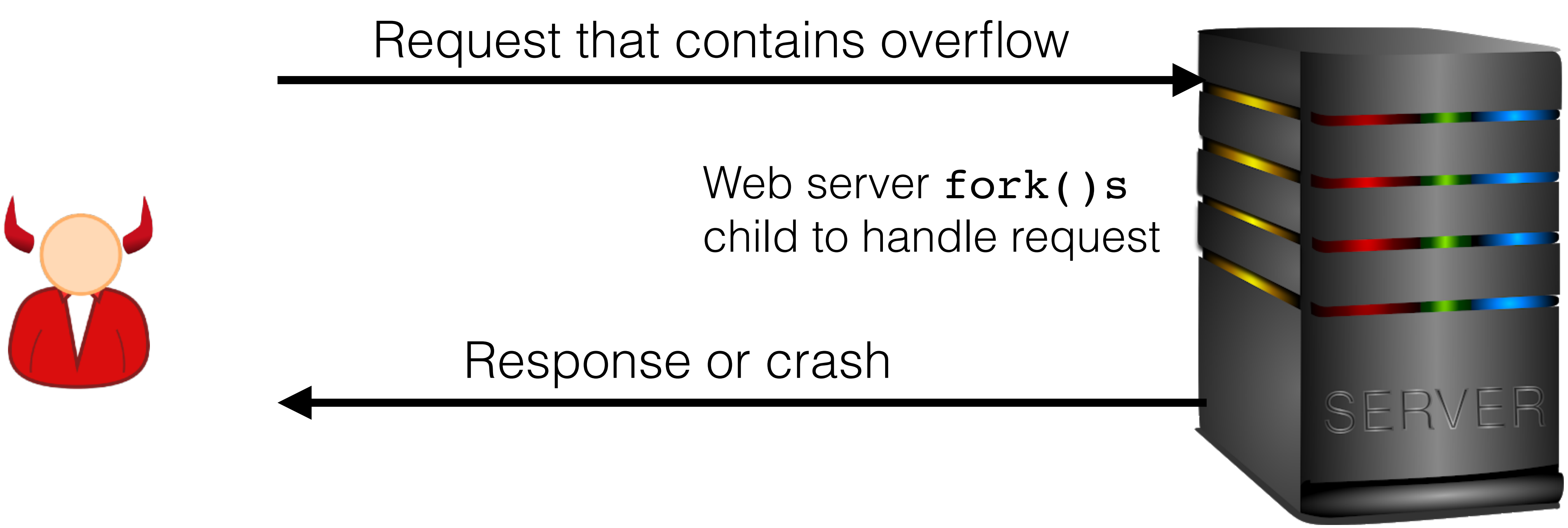
Bypassing Canaries via Complex Bugs




```
int foo(char *s1, char *s2) {
  char *p;
  char buf[64];
  p = buf;
  strcpy(p, s1); // oh no :(
  …
  strncpy(p, s2, 16);
  …
}
```



Bypassing Canaries via “Reading the Stack”



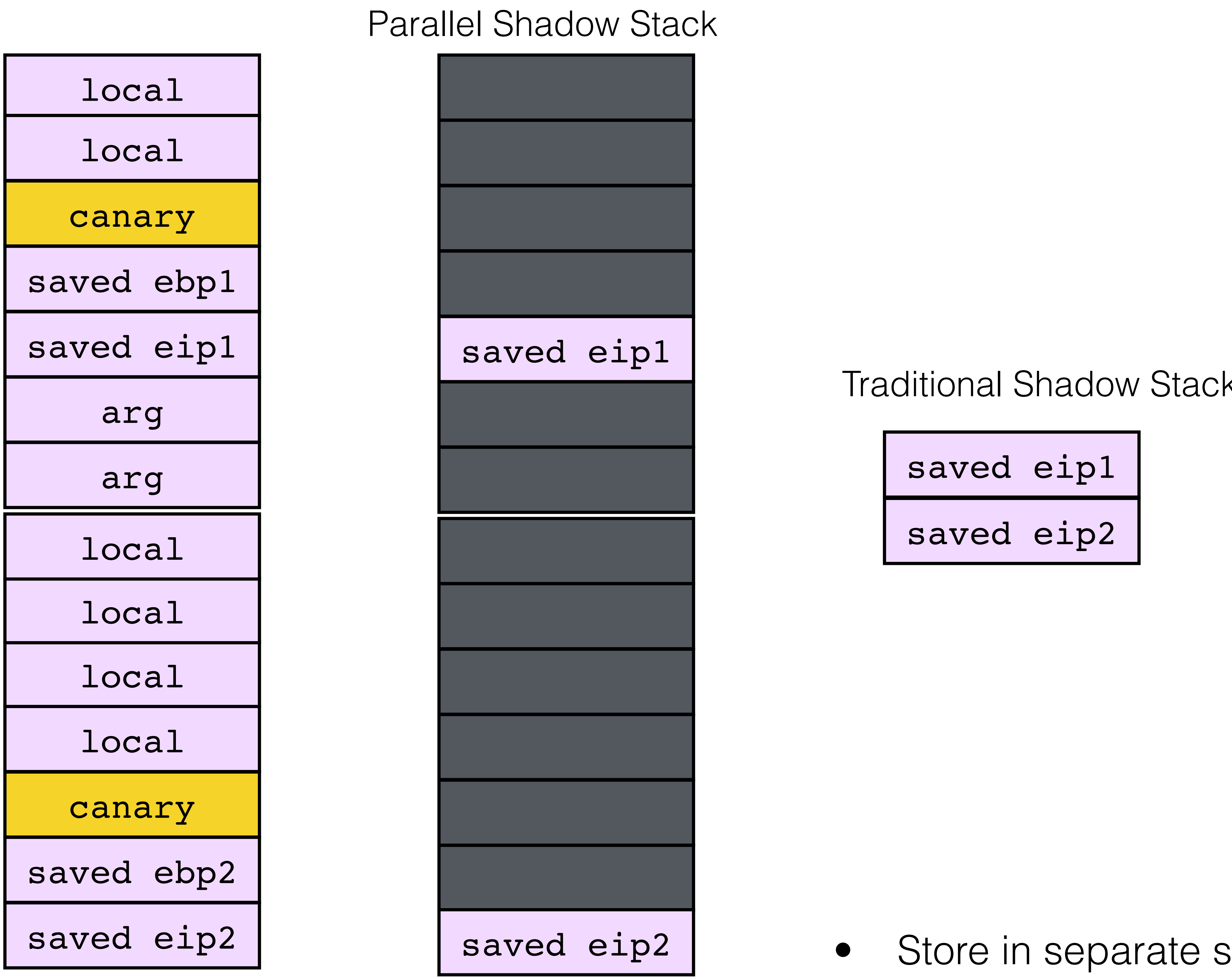
Child inherits same random canary value `0xXXYYZZWW`.

local buf[]
…
local buf[]
XX YY ZZ WW
saved ebp
saved eip

local buf[]	
…	
local buf[]	
XX	YY ZZ WW
saved ebp	
saved eip	

Overflow 1 byte and observe if process crashes.
Learn **`XX`** byte after 256 tries! Repeat for rest.

Other Countermeasures: Shadow Stacks



- Store in separate segment to protect from overflow.

Outline of Lecture 5: Buffer Overflow Countermeasures

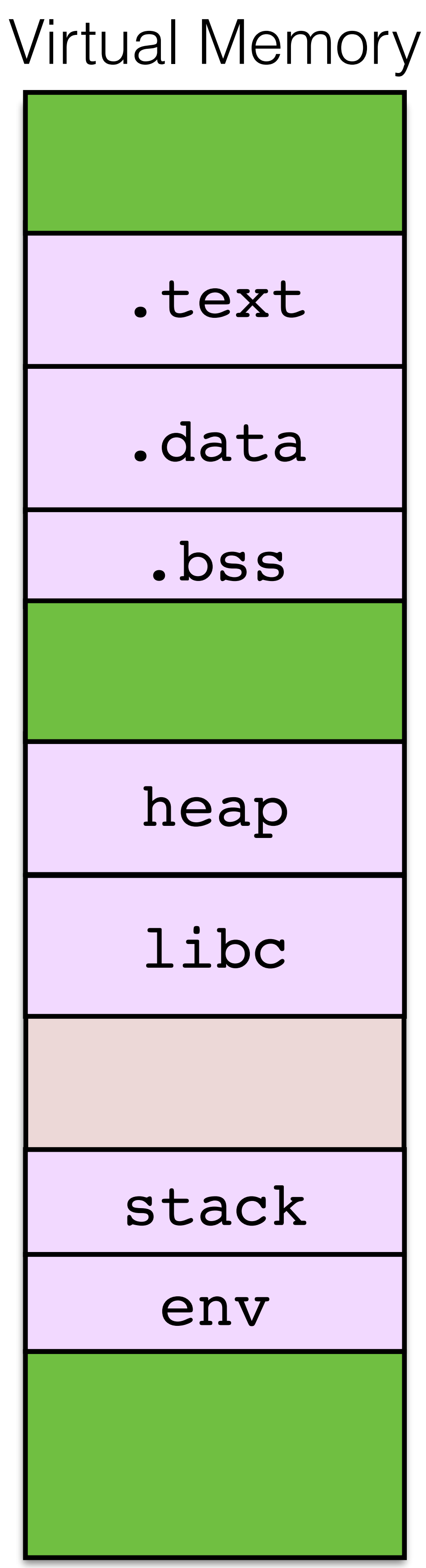
1. Heap vulnerabilities (briefly)

2. Stack Protectors

3. **Address-Space Layout Randomization**

4. W ^ X and ROP

Address-Space Layout Randomization (ASLR)



Linux PaX implementation:

- Add randomize offsets of in green areas
- 16 bits, 16 bits, 24 bits or randomness respectively
- Makes guessing return addresses harder

Possible attacks:

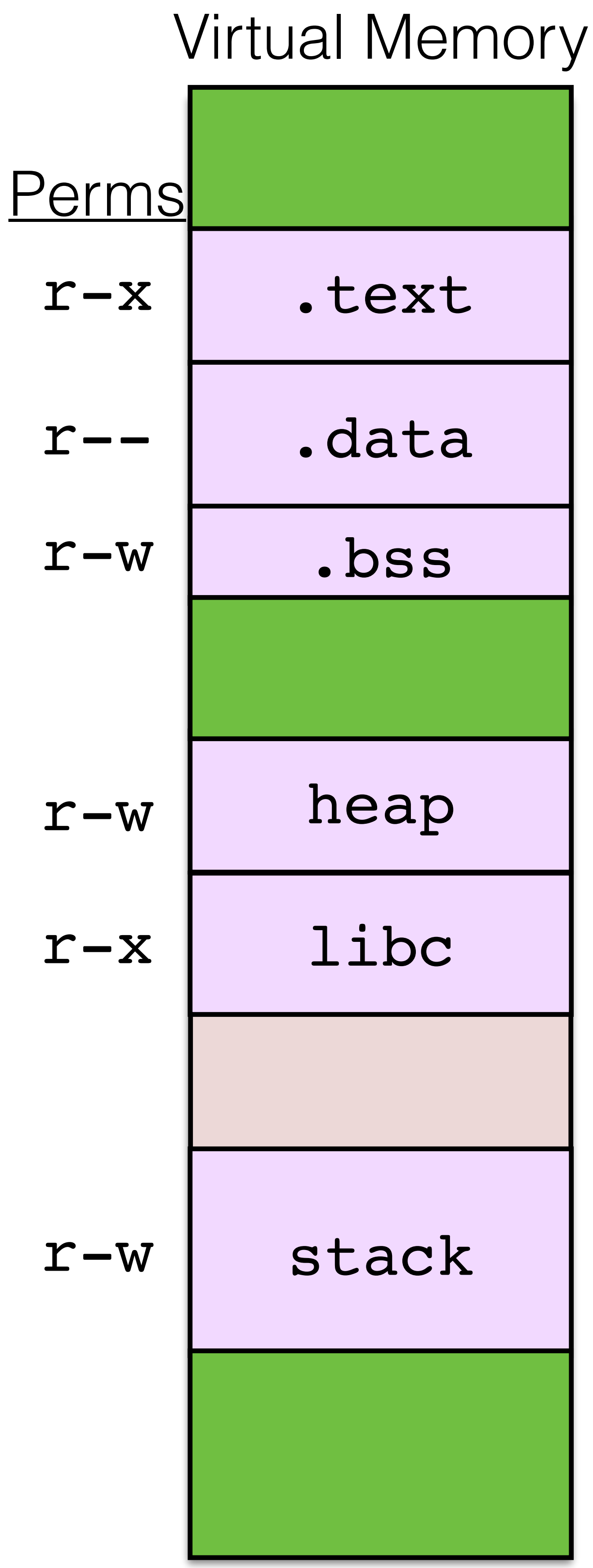
- Huge NOP sleds + Copy shellcode many times in heap.
- Side channels (or printf bugs) can leak random choice
- Brute force with large number of forks

Modern machines have 64-bit addresses, making ASLR stronger.

Outline of Lecture 5: Buffer Overflow Countermeasures

1. Heap vulnerabilities (briefly)
2. Stack Protectors
3. Address-Space Layout Randomization
4. **W ^ X and ROP**

W ^ X (“Write XOR Execute”)



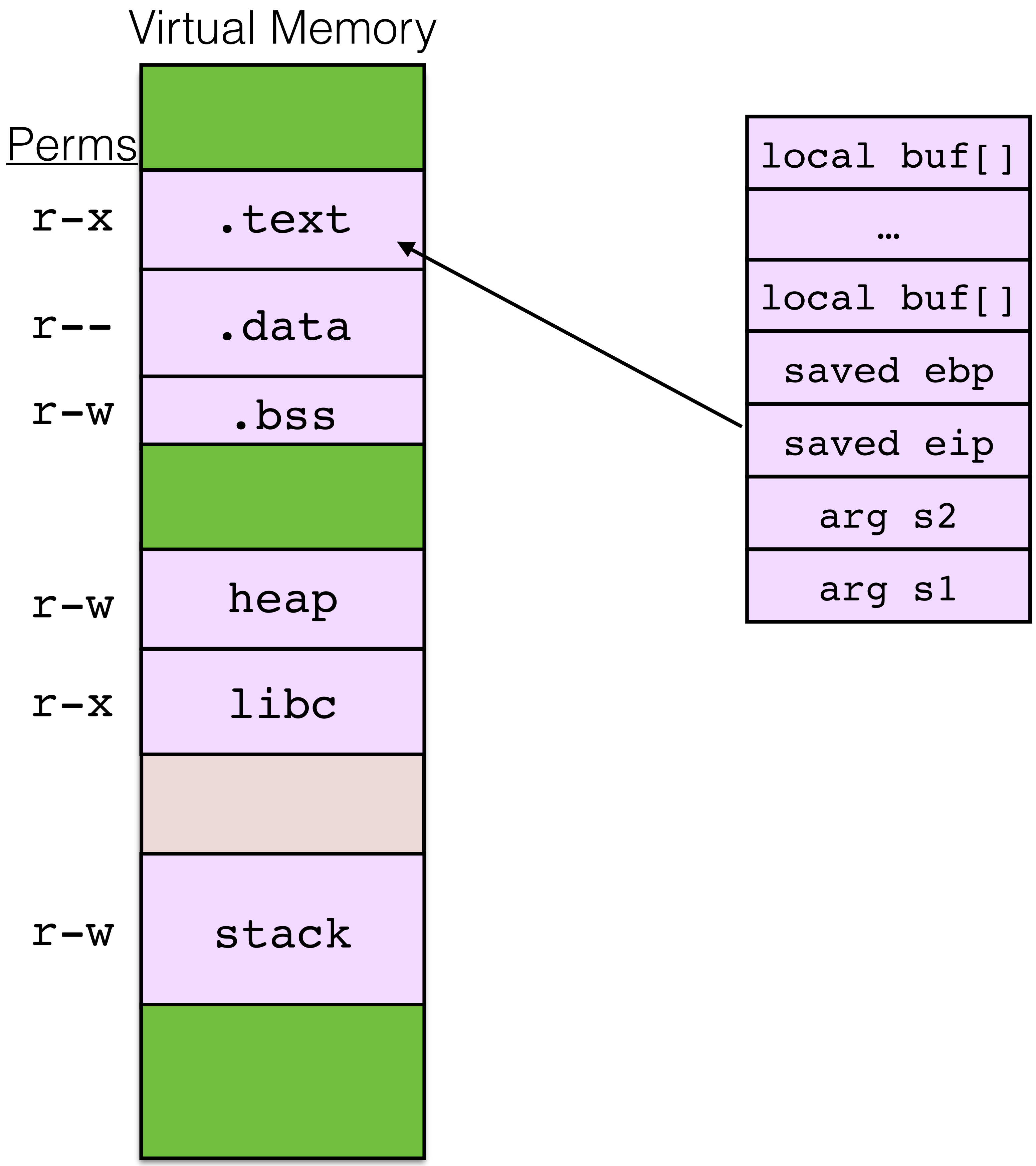
Cannot execute code on stack (will segfault).

May mark each segment as either writeable or executable, but never both.

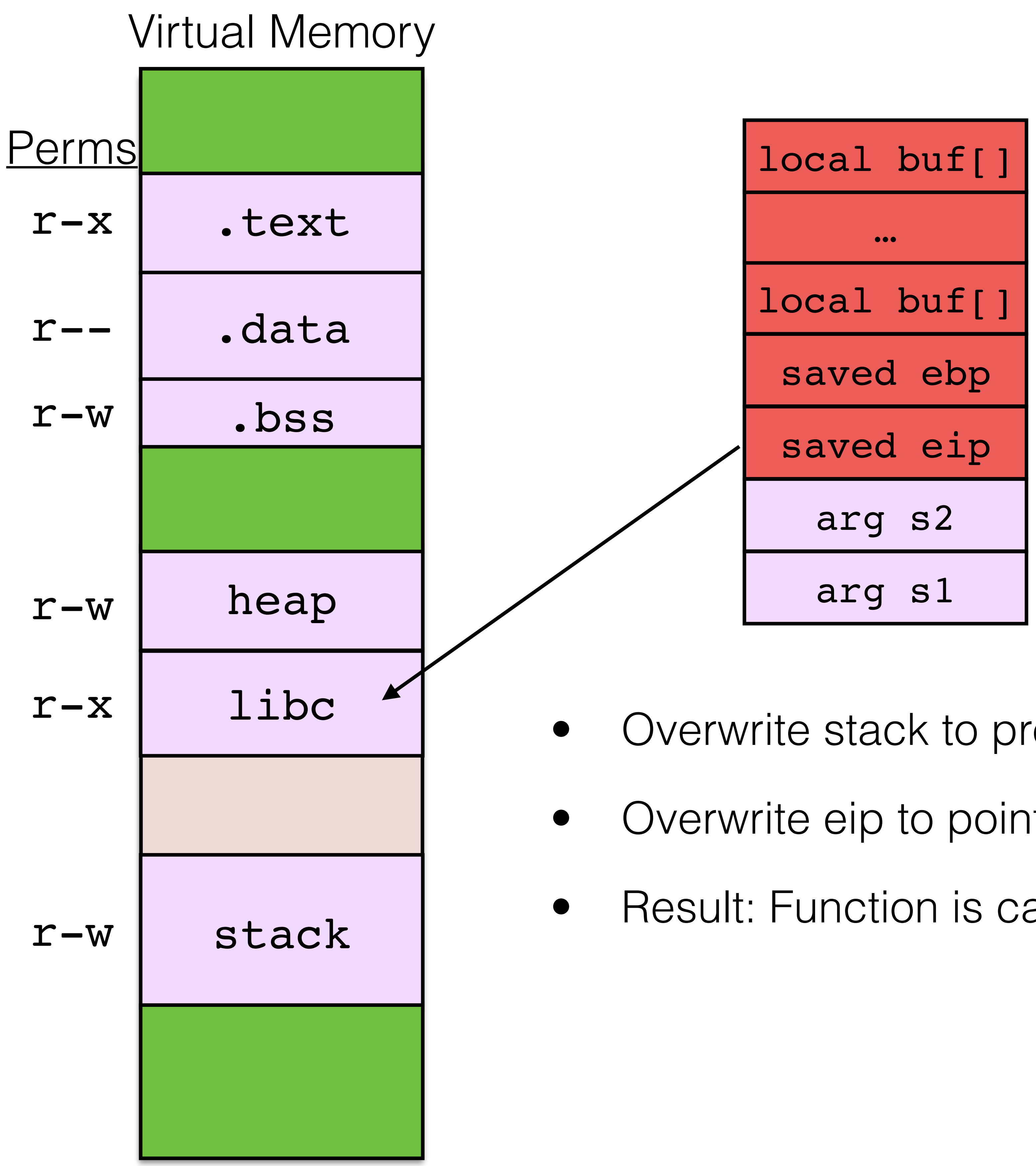
- Modern hardware support: x64 (the x86 successor)
- Software implementations (PaX/ExecShield in Linux, DEP in Windows, …)
- Slowly adopted in software since early 2000s
- Also used in virtual machine / sandboxes

Which of Paul van O.’s principles is this?

Bypassing W ^ X: Return-to-libc

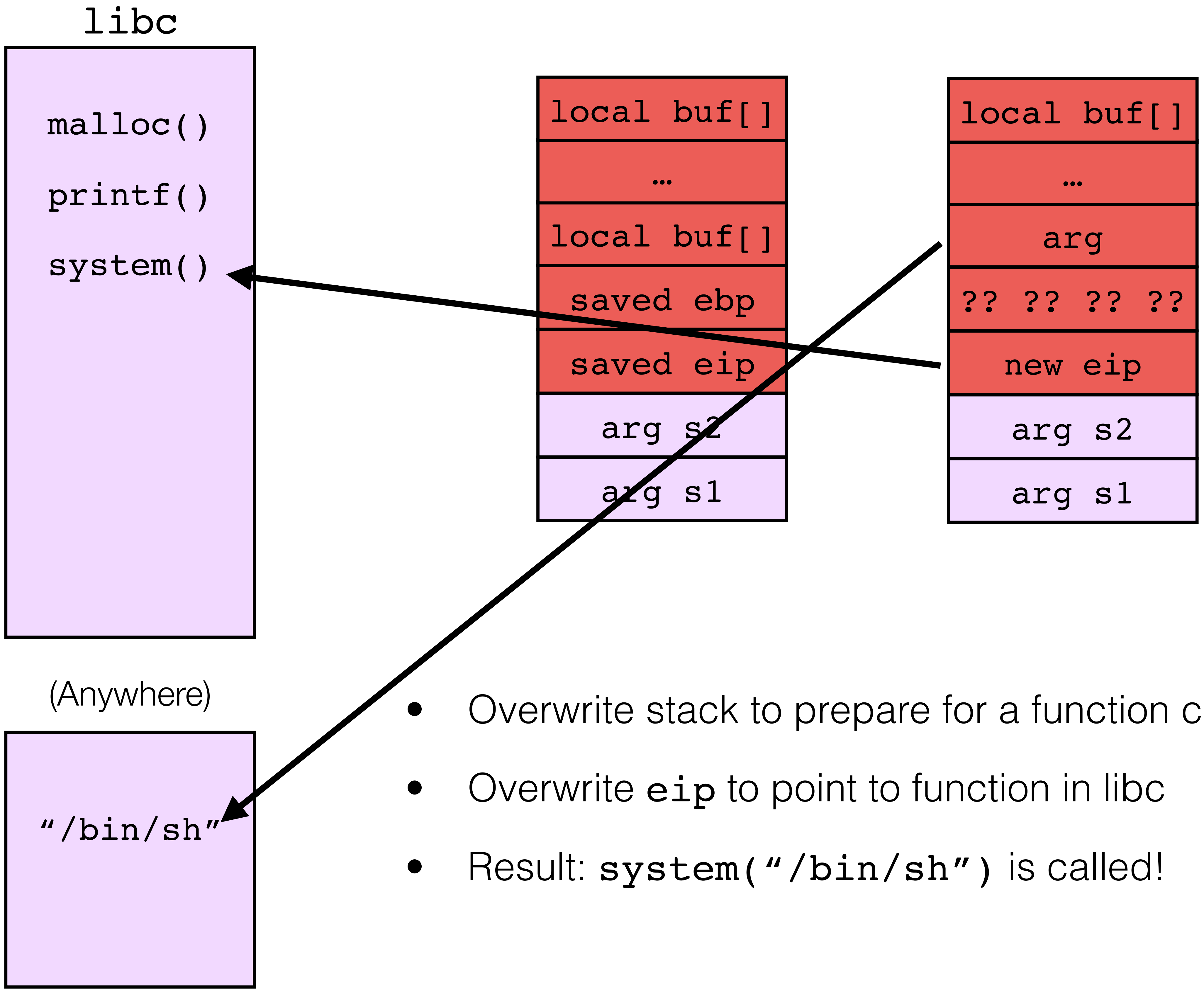


Bypassing W ^ X: Return-to-libc



- Overwrite stack to prepare for a function call
- Overwrite eip to point to function in libc
- Result: Function is called!

Return-to-libc Details



- Overwrite stack to prepare for a function call
- Overwrite `eip` to point to function in libc
- Result: `system("/bin/sh")` is called!

Going Further: Return-Oriented Programming (ROP)

- Return-to-libc enables calling functions in libc
- Going further: Why not “return” into the middle of functions, and only execute the end?

return-to-libc jumps here…

… but we could jump here instead to execute two instructions, then regain control

```
Dump of assembler code for function malloc:
   0xb7ff2110 <+0>:push   %ebx
   0xb7ff2111 <+1>:call   0xb7ff48e9 <__x86.get_pc_thunk.bx>
   0xb7ff2116 <+6>:add    $0xceea,%ebx
   0xb7ff211c <+12>:  sub    $0x10,%esp
   0xb7ff211f <+15>:  pushl  0x18(%esp)
   0xb7ff2123 <+19>:  push   $0x8
   0xb7ff2125 <+21>:  call   0xb7fdb810 <__libc_memalign@plt>
   0xb7ff212a <+26>:  add    $0x18,%esp
   0xb7ff212d <+29>:  pop    %ebx
   0xb7ff212e <+30>:  ret
```

- General ROP attack: Comb through libc for functions that end in useful instructions. Build shellcode as a long string of returns that execute the useful instructions.
- Shown to be “Turing Complete” (Shacham 2008)

Even Crazier ROP

- Can return *into the middle of an instruction(!)*

Example in libc (Shacham 2008): `f7 c7 07 00 00 00 0f 95 45 c3`

Jump to front:

```
f7 c7 07 00 00 00      test $0x00000007, %edi
0f 95 45 c3            setnzb -61(%ebp)
```

Jump one byte later:

```
c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```

The End