# Memory Protection

## CMSC 23200, Spring 2026, Lecture 4

David Cash and Grant Ho

University of Chicago

# Outline of Lecture 4

1. Heap vulnerabilities (briefly)
2. Memory-Safe Languages
3. Stack Protectors
4. Address-Space Layout Randomization
5. W ^ X and ROP
6. Fuzzing
7. Next big topic: Cryptography (begin if time left)

# Outline of Lecture 4

1. **Heap vulnerabilities (briefly)**
2. Memory-Safe Languages
3. Stack Protectors
4. Address-Space Layout Randomization
5. W ^ X and ROP
6. Fuzzing
7. Next big topic: Cryptography (begin if time left)

# Heap Memory Vulnerabilities

Assume that:

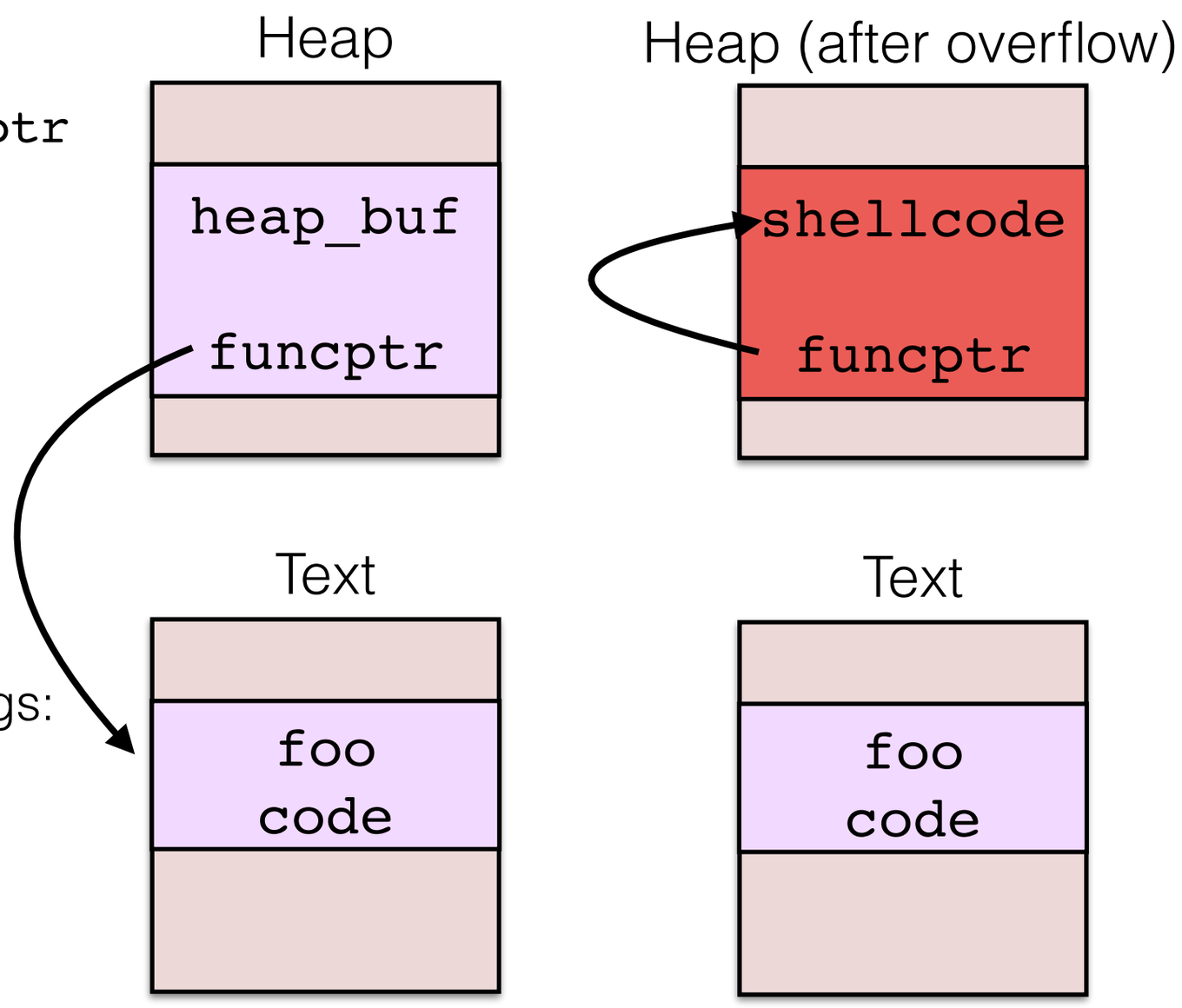
1. Program has heap variable (**`heap_buf`**)
2. A function pointer, stored in **`heap_buf`**, points to **`foo`**

Attack:

1. Overflow **`heap_buf`** and corrupt **`funcptr`**
2. Later calls using **`funcptr`** will call **`shellcode`**

Other notes:

- Attacker can overwrite program vars
  - e.g. change to "**`admin=true`**"
- Vulnerabilities possible from many bugs:
  - use-after-free
  - double-free
  - ...



# Outline of Lecture 4

1. Heap vulnerabilities (briefly)
2. **Memory-Safe Languages**
3. Fuzzing
4. Stack Protectors
5. Address-Space Layout Randomization
6. W ^ X and ROP
7. Next big topic: Cryptography

# Memory Safe Languages

- **<u>Definition</u>**: A language is **memory safe** if it is *impossible* to improperly access memory *within its abstract model of computation*.
- <u>Examples</u>: Java, Python, (safe) Rust, Go
- <u>Non-Examples</u>: C, C++, assembly
- Memory vulnerabilities still happen, but attackers must attack the implementation of the language (i.e. **`python3`** emulator or **`rustc`** compiler)
    - These tools are ultimately written in a non-safe language (something must be)
    - Much harder to attack a mature, community-made object than some random company's implementation

# Memory Safe Languages

- **<u>Definition</u>**: A language is **memory safe** if it is *impossible* to improperly access memory *within its abstract model of computation*.
- <u>Examples</u>: Java, Python, (safe) Rust, Go
- <u>Non-Examples</u>: C, C++, assembly
- Memory vulnerabilities still happen, but attackers must attack the implementation of the language (i.e. `python3` emulator or `rustc` compiler)
  - These tools are must be)
  - Much harder to random compar

**CVE-2021-3177 Detail**

MODIFIED

This CVE record has been updated after NVD enrichment efforts were completed. Enrichment data supplied by the NVD may require amendment due to these changes.

## Description

Python 3.x through 3.9.1 has a buffer overflow in PyCArg_repr in _ctypes/callproc.c, which may lead to remote code execution in certain Python applications that accept floating-point numbers as untrusted input, as demonstrated by a 1e300 argument to c_double.from_param. This occurs because sprintf is used unsafely.

# Outline of Lecture 4

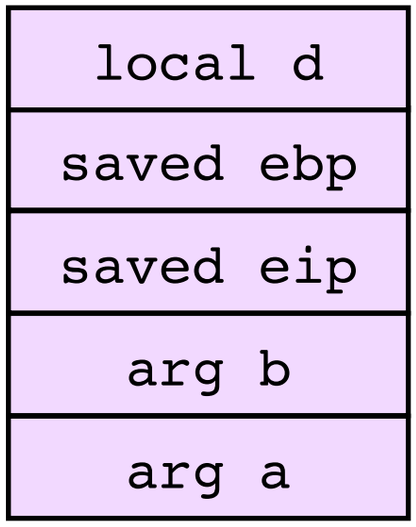
1. Heap vulnerabilities (briefly)
2. Memory-Safe Languages
3. **Stack Protectors**
4. Address-Space Layout Randomization
5. W ^ X and ROP
6. Fuzzing
7. Next big topic: Cryptography

# Countermeasure #1: Stack Canaries

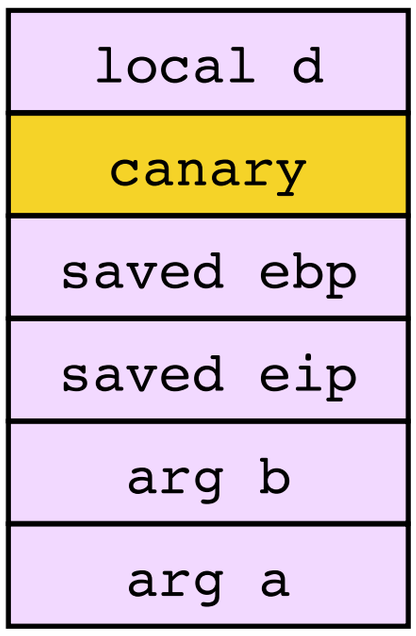
# Stack Canaries (a.k.a. Stack Protectors)

- 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.

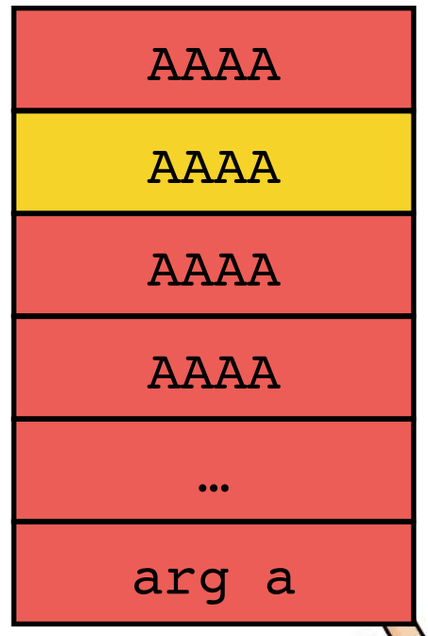
Standard frame




Frame with canary




After overflow

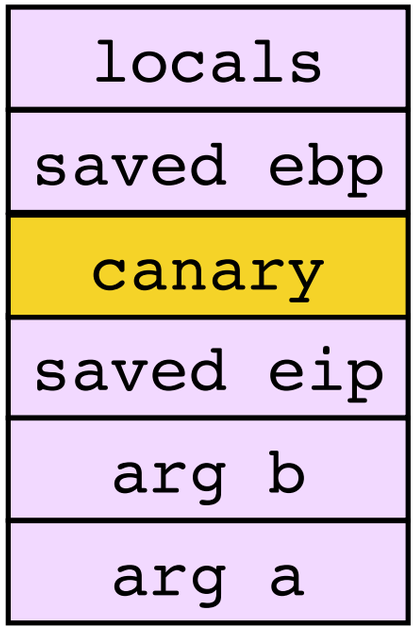



Incorrect!
Detected
before return.

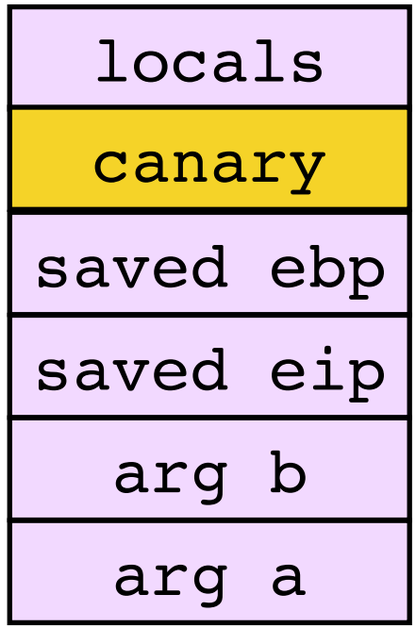
__canary_death_handler

# Where to put canaries?

StackGuard (1998)

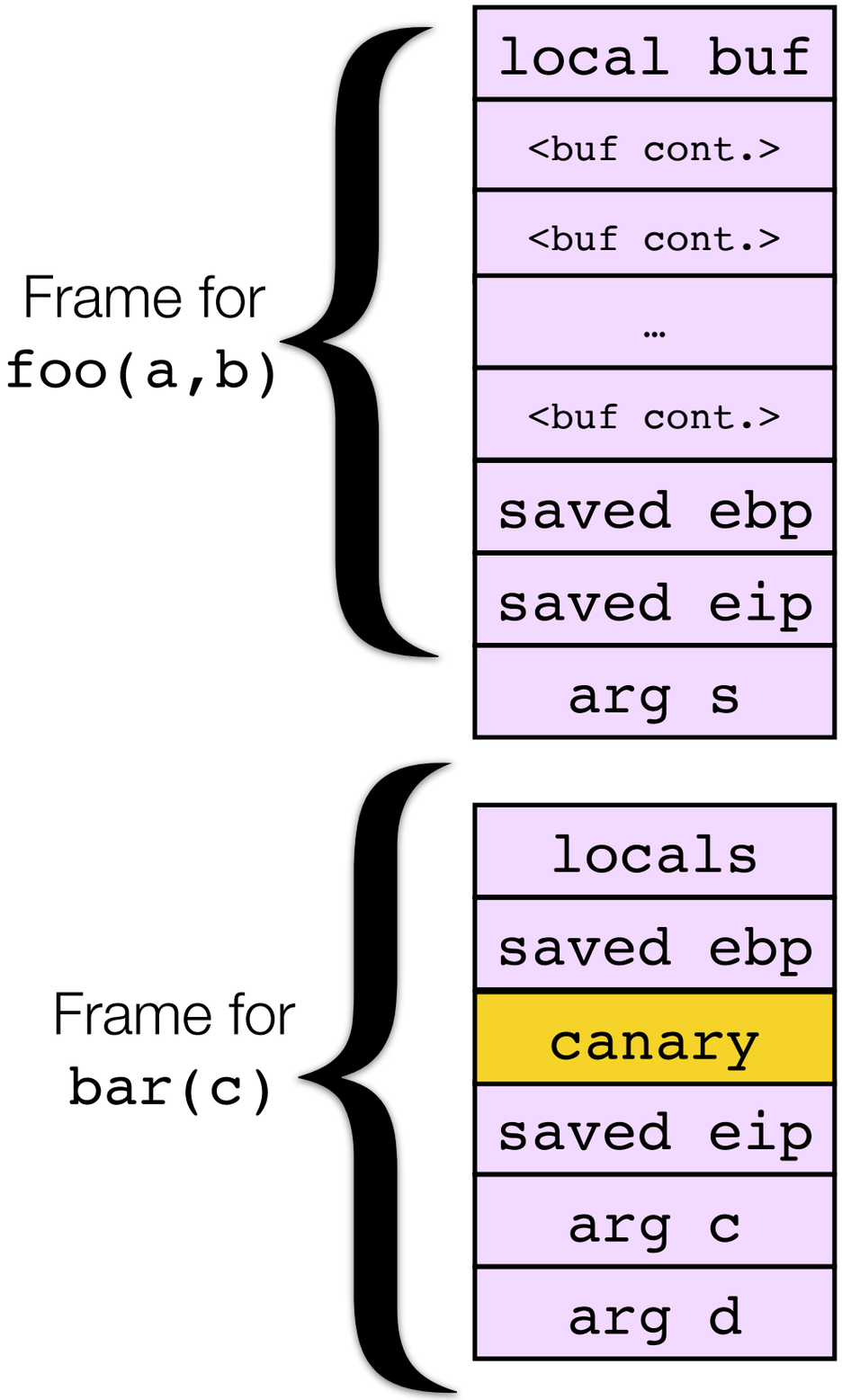


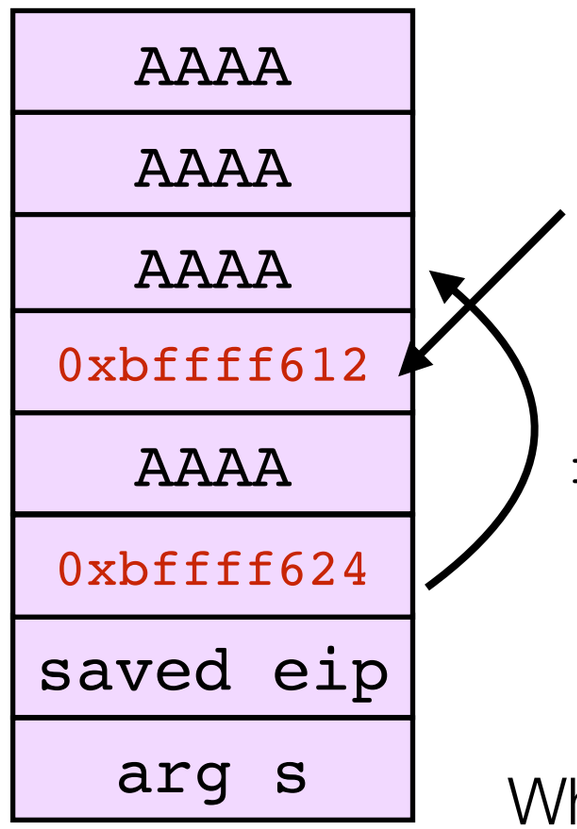

ProPolice (IBM, 2001-2005)

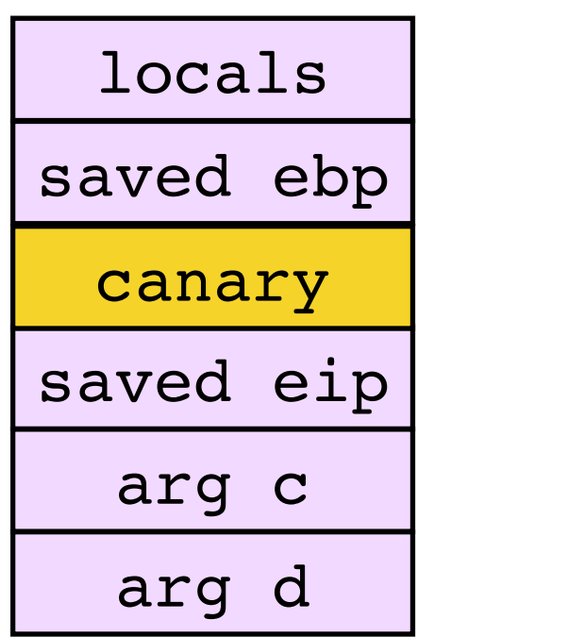



- Manipulating **`ebp`** (frame pointer) is almost as bad as **`eip`** (return address)!

# 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

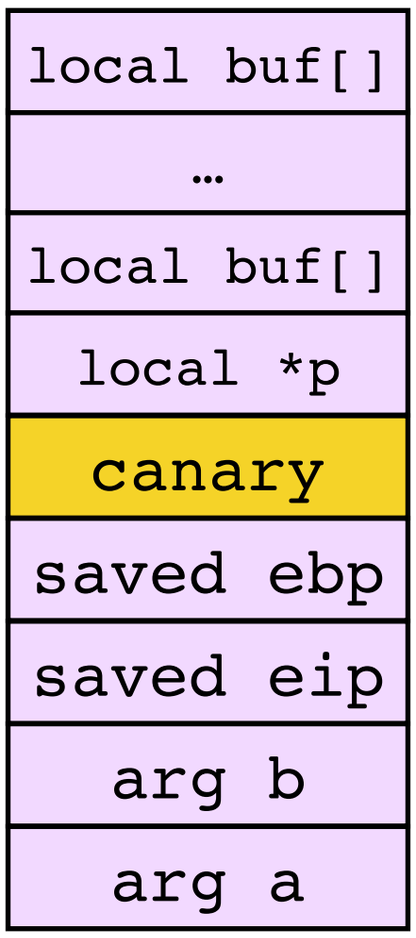
| Flag | Default? | Notes |
|---|---|---|
| -fno-stack-protector | No | Turns off protections |
| -fstack-protector | Yes | Adds to funcs that call alloca() & w/ arrays larger than 8 chars (--param=ssp-buffer-size changes 8) |
| -fstack-protector-strong | No | Also funcs w/ any arrays & refs to local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

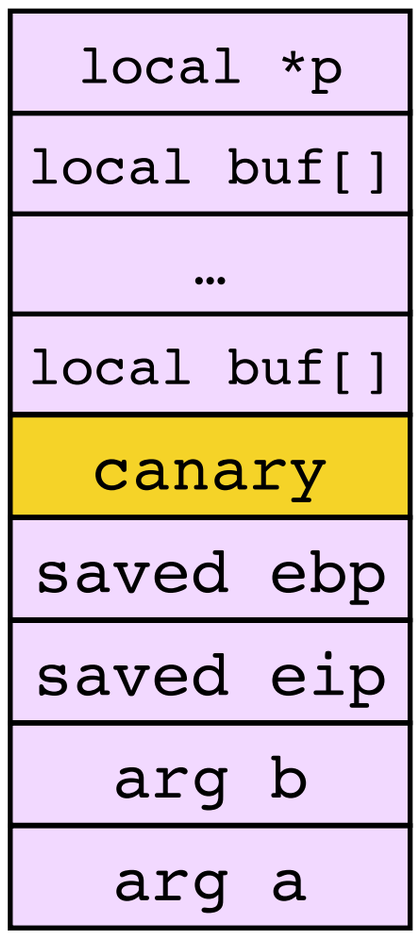
- gcc puts local arrays below other locals, even if declared in other order

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

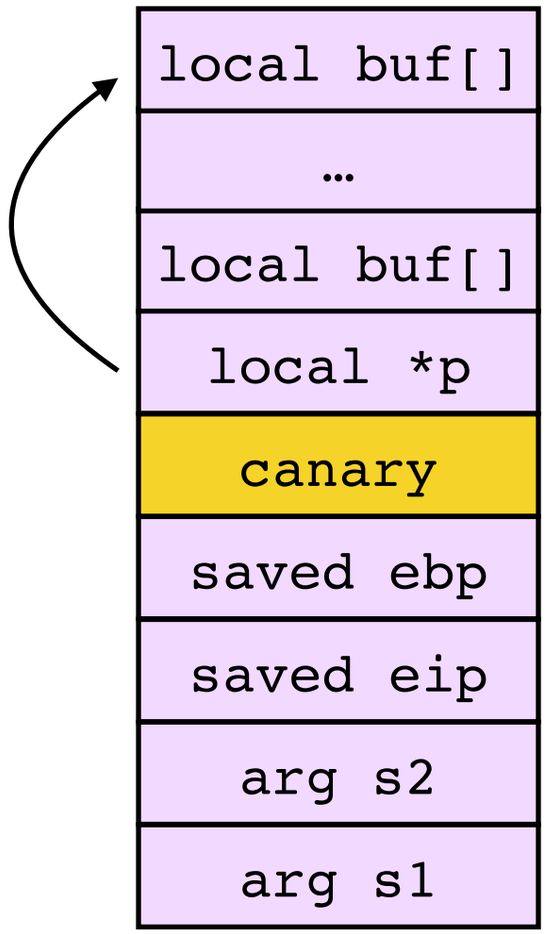



VS

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

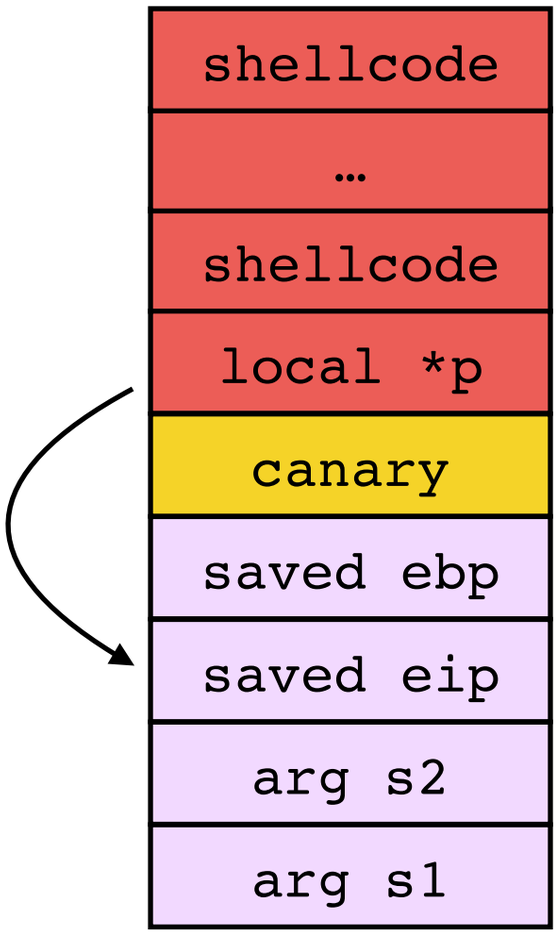


# Bypassing Canaries via Complex Bugs



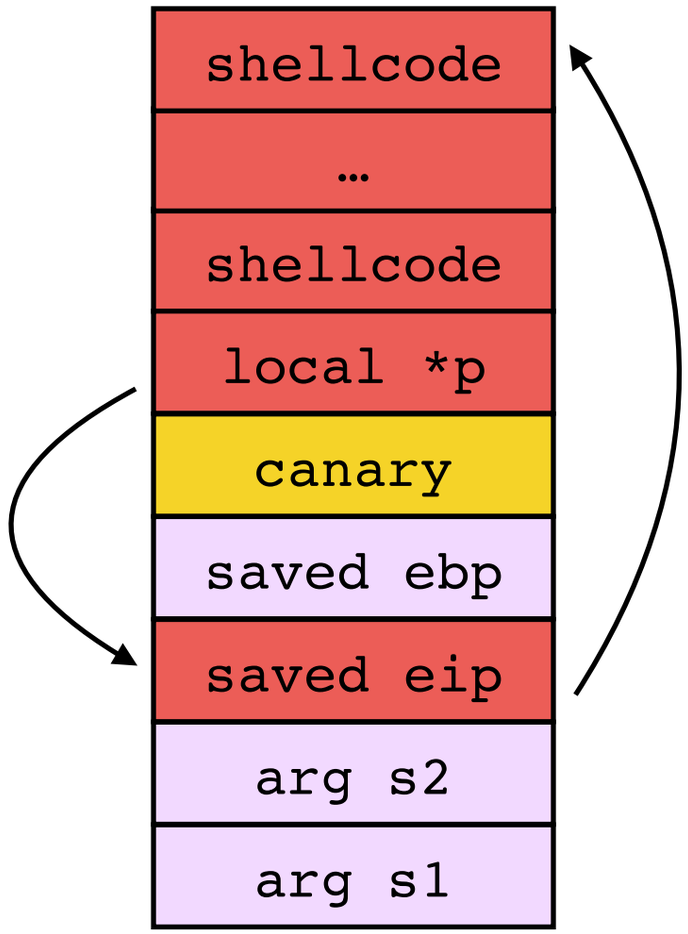

```
int foo(char *s1, char *s2) {
  char *p;
  char buf[64];
  p = buf;
  strcpy(p, s1); // oh no :(
  …
  strncpy(p, s2, 16);
  …
}
```

# Bypassing Canaries via Complex Bugs



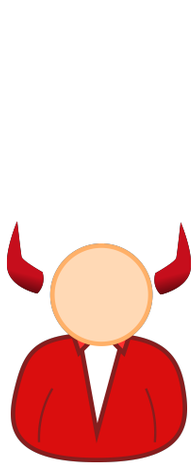

```
int foo(char *s1, char *s2) {
  char *p;
  char buf[64];
  p = buf;
  strcpy(p, s1); // oh no :(
  …
  strncpy(p, s2, 16);
  …
}
```

# 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);
  …
}
```

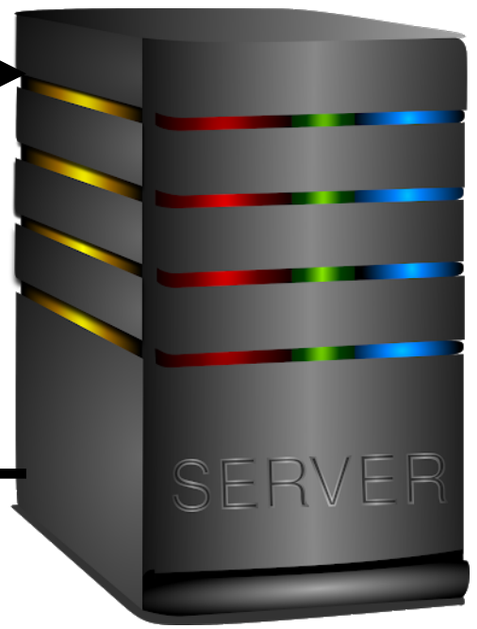
# Bypassing Canaries via “Reading the Stack”



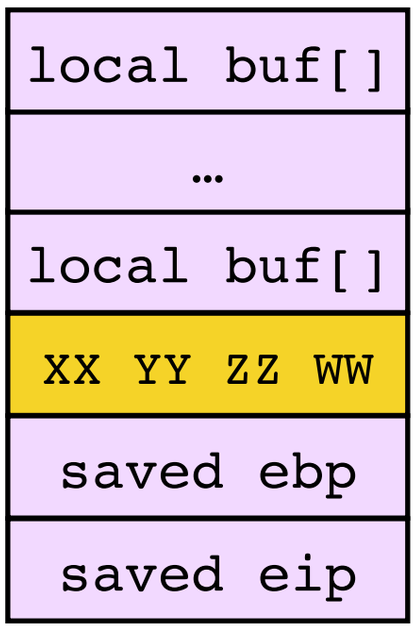
Request that contains overflow

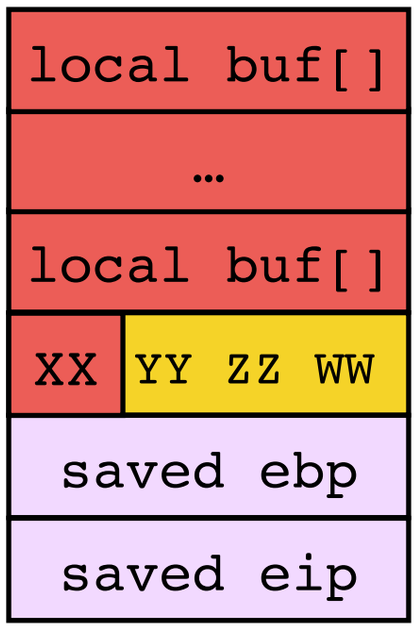
Web server **`fork()`s** child to handle request

Response or crash



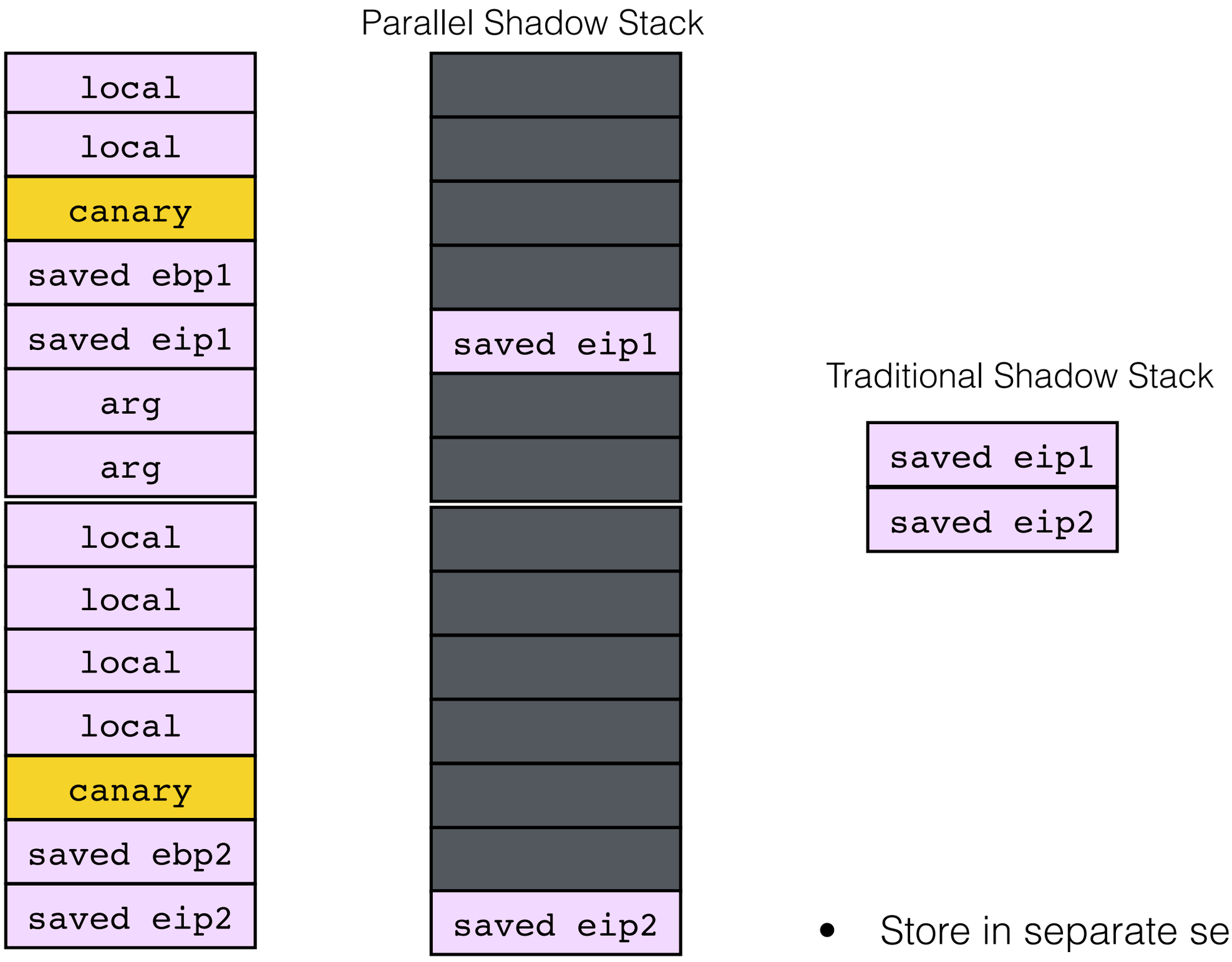

Child inherits same random canary value **`0xXXYYZZWW`**.




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

# Other Countermeasures: Shadow Stacks

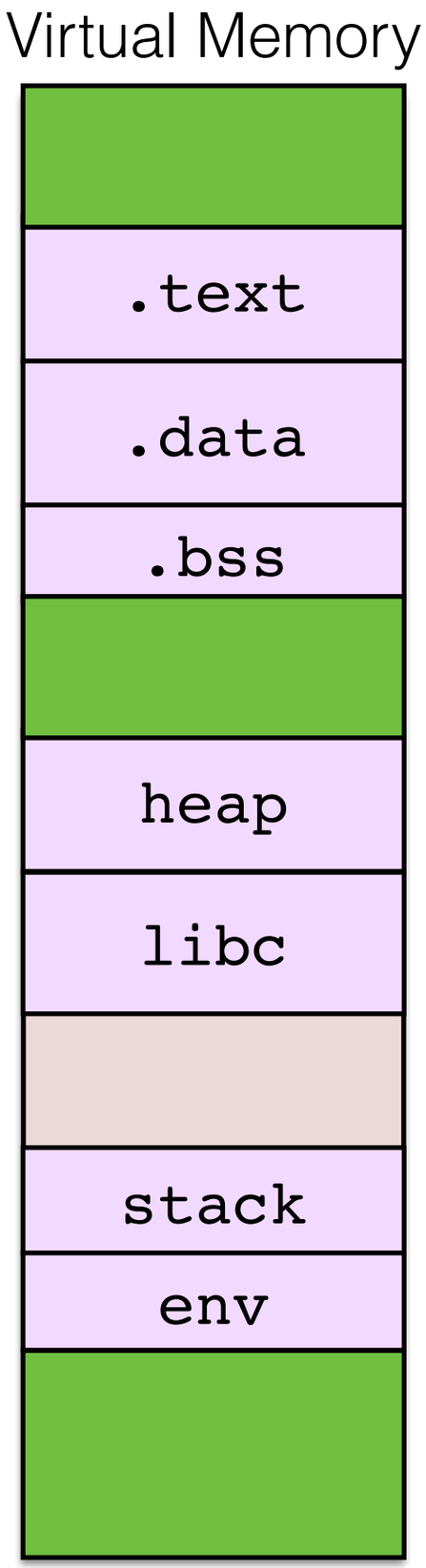



- Store in separate segment to protect from overflow.

# Outline of Lecture 4

1. Heap vulnerabilities (briefly)
2. Memory-Safe Languages
3. Stack Protectors
4. **Address-Space Layout Randomization**
5. W ^ X and ROP
6. Fuzzing
7. Next big topic: Cryptography

# 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:

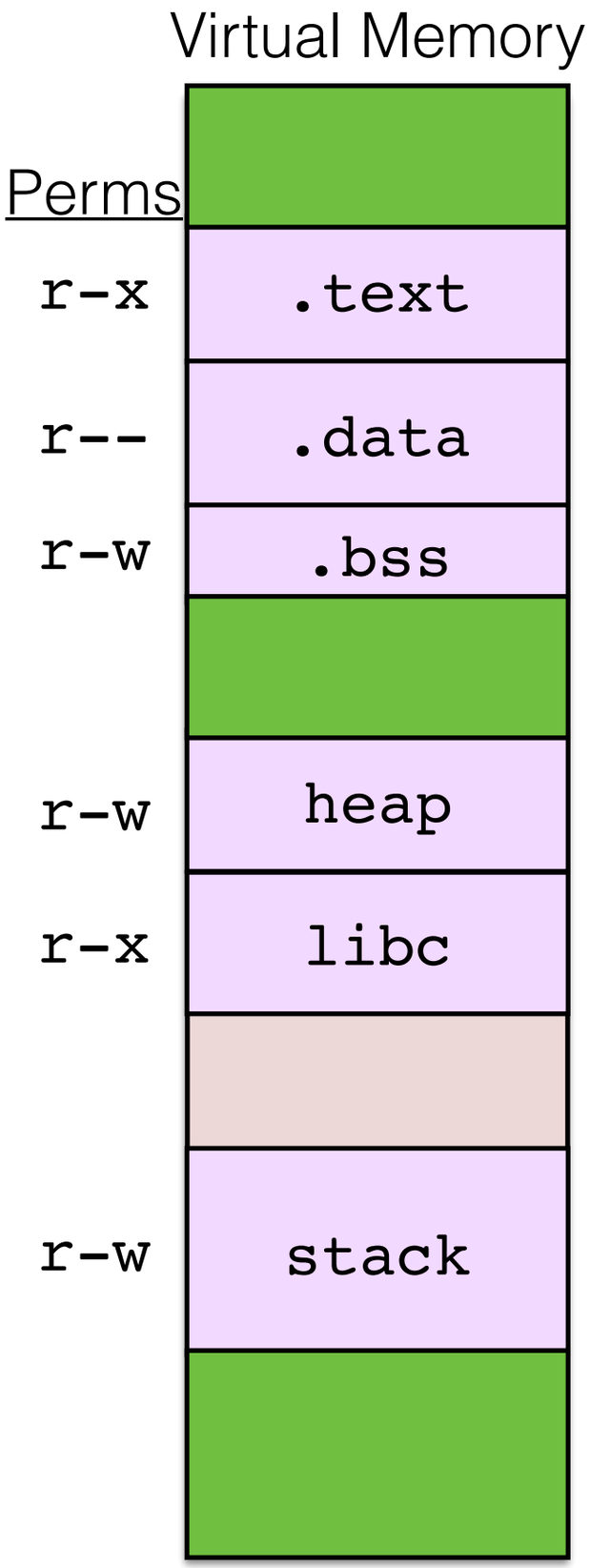
- Huge NOP sleds + Copy shellcode many times in heap.
- Side channels (or printf bugs) can leak random choice
- Brute force with large number of forks

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

# Outline of Lecture 4

1. Heap vulnerabilities (briefly)
2. Memory-Safe Languages
3. Stack Protectors
4. Address-Space Layout Randomization
5. **W ^ X and ROP**
6. Fuzzing
7. Next big topic: Cryptography

# W ^ X (“Write XOR Execute”)



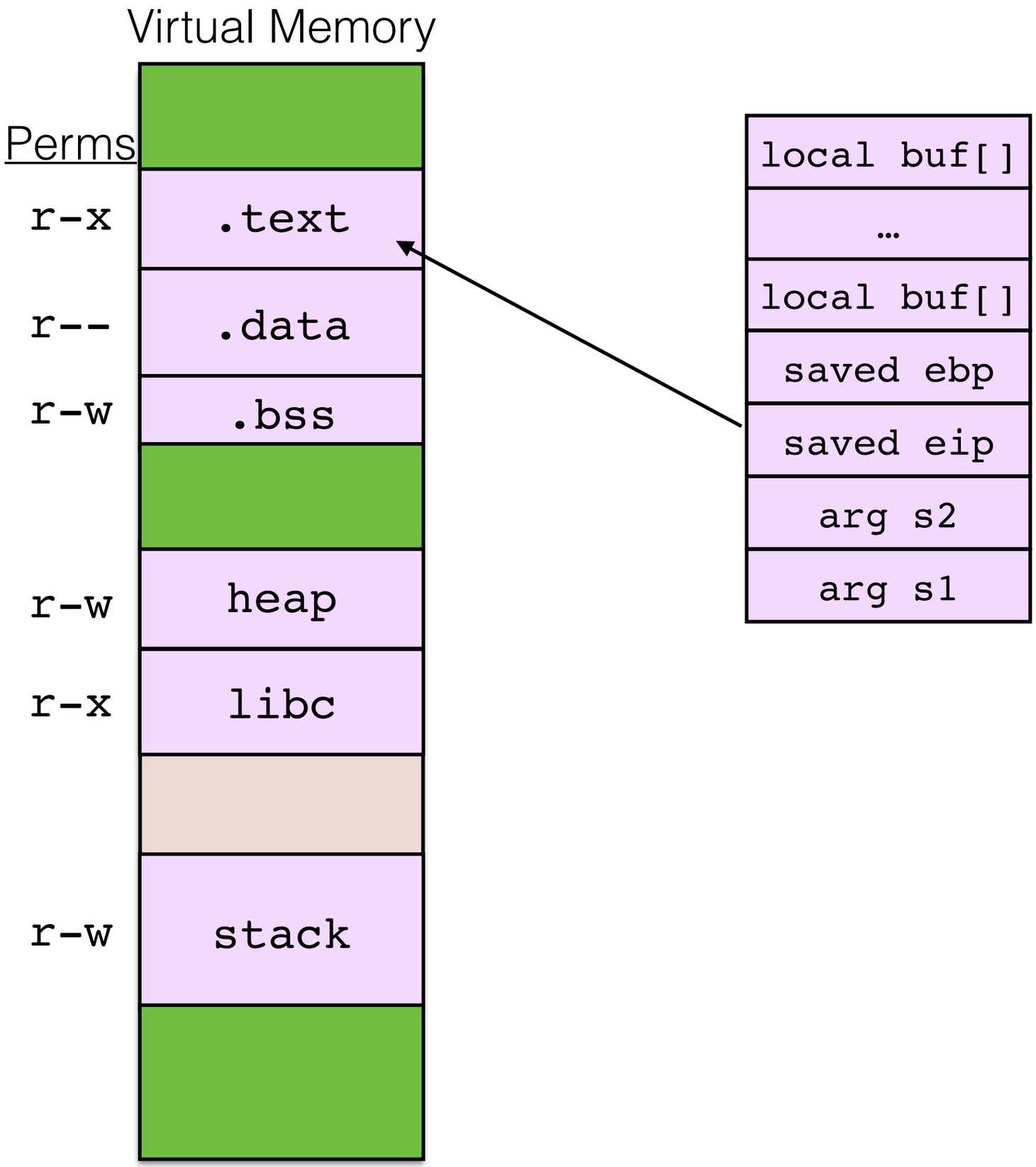

Cannot execute code on stack (will segfault).

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

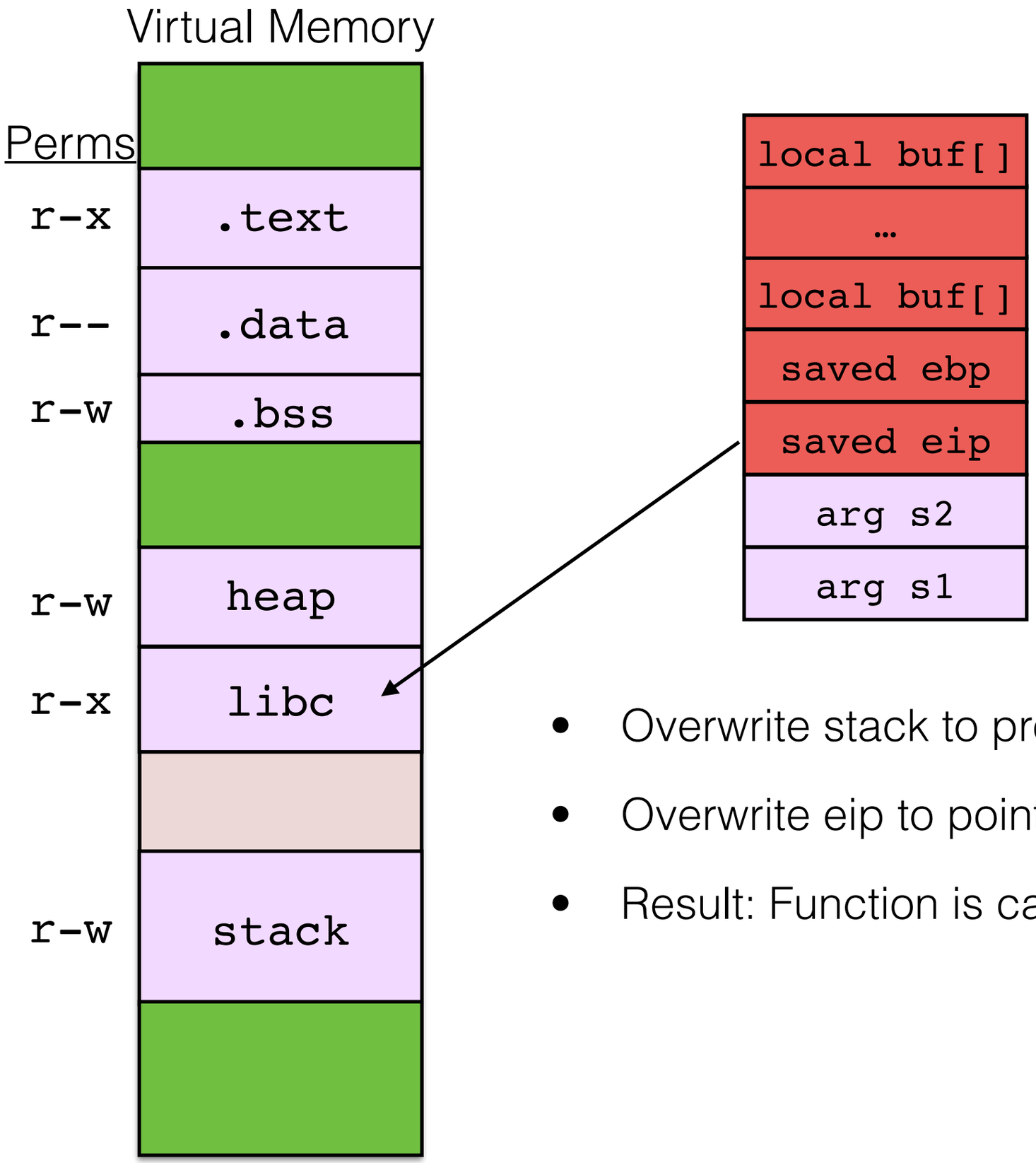
- Modern hardware support: x64 (the x86 successor)
  - (x86 did not support executable restriction)
- Software implementations (PaX/ExecShield in Linux, DEP in Windows, …) used tricks to similar effect
- Slowly adopted in software since early 2000s
- Also used in virtual machine / sandboxes

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

# Bypassing W ^ X: Return-to-libc

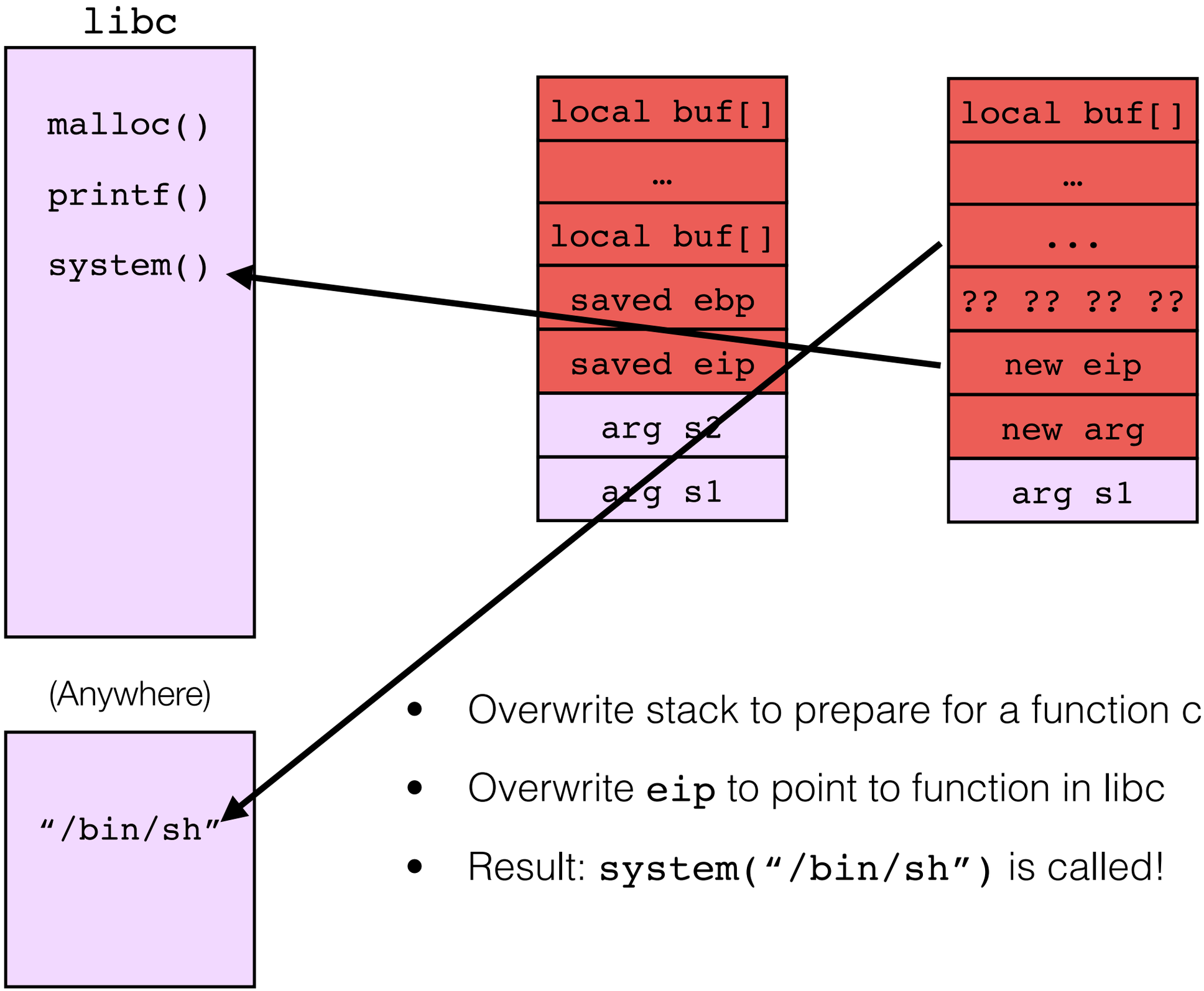


# Bypassing W ^ X: Return-to-libc




- 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
```

# Outline of Lecture 4

1. Heap vulnerabilities (briefly)
2. Memory-Safe Languages
3. Stack Protectors
4. Address-Space Layout Randomization
5. W ^ X and ROP
6. **Fuzzing**
7. Next big topic: Cryptography

# Program Fuzzing

Run program on huge number of automatically-generated inputs, searching for crashes.

## Linux Mint fixes screensaver bypass discovered by two kids

Two children playing on their dad's computer accidentally found a way to bypass the screensaver and access locked systems.

By Catalin Cimpanu for Zero Day | January 15, 2021 -- 18:28 GMT (10:28 PST) | Topic: Security

MORE FROM CATALIN CIMPANU

Security
**Hacker leaks data of**

"A few weeks ago, my kids wanted to hack my Linux desktop, so they typed and clicked everywhere while I was standing behind them looking at them play," wrote a user identifying themselves as robo2bobo.

According to the bug report, the two kids pressed random keys on both the physical and on-screen keyboards, which eventually led to a crash of the Linux Mint screensaver, allowing the two access to the desktop.

"I thought it was a unique incident, but they managed to do it a second time," the user added.

# Types of Fuzzing

**Mutation-based (dumb)**: Take an initial set of examples and make random changes to them.

- Millions of inputs (can just run forever)

- Possibly lower quality, unable to find certain types of inputs

**Generative (smart)**: Describe inputs to fit format/protocol, then generate inputs from that grammar with changes.

- Run with fewer inputs, which can be directed to certain types

**Q:** Which is better for `func()`?

**Q:** Which is better for heart bleed?

```
int func(char *s) {
  if(check_sum_is_valid(s)) {
    complicated_func(s);
  }
  else {
    simple_func(s);
  }
}
```

# Problems with Fuzzing

**Mutation-based (dumb)**: How long to run? And we need a strong server.

**Generative (smart)**: Run out of test cases. A lot more work.

**General problems**:

— Need to identify when bug/crash occurs automatically.

— Don’t want to report same bug 1000s of times.

# Fuzzing and Code Coverage

**Testing heuristic**: The more of the code that is executed by tests, the more likely we are to find bugs.

Can try to cover:

- Lines/instructions of source/binary
- Branches in binary/source
- Paths in binary/source

**Example:**

```
int func(int a, int b) {
  if(a > 2)
    a = 2;
  if(b > 2)
    b = 2;
  return a+b;
}
```

# A Notable Example: Dumb Mutation Fuzzing of PDFs

Charlie Miller, 2010:

1. Download 1000s of PDFs from internet

2. For each one, change some bytes literally at random.

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
 rbyte = random.randrange(256)
 rn = random.randrange(len(buf))
 buf[rn] = “%c"%(rbyte)
```

Results:

Apple Preview: 250 unique crashes, 60 exploits

Acrobat: 100 unique crashes, 4 exploits

*Slide credit: https://cs155.stanford.edu/lectures/06-testing.pdf*

# American Fuzzy Loop (AFL)

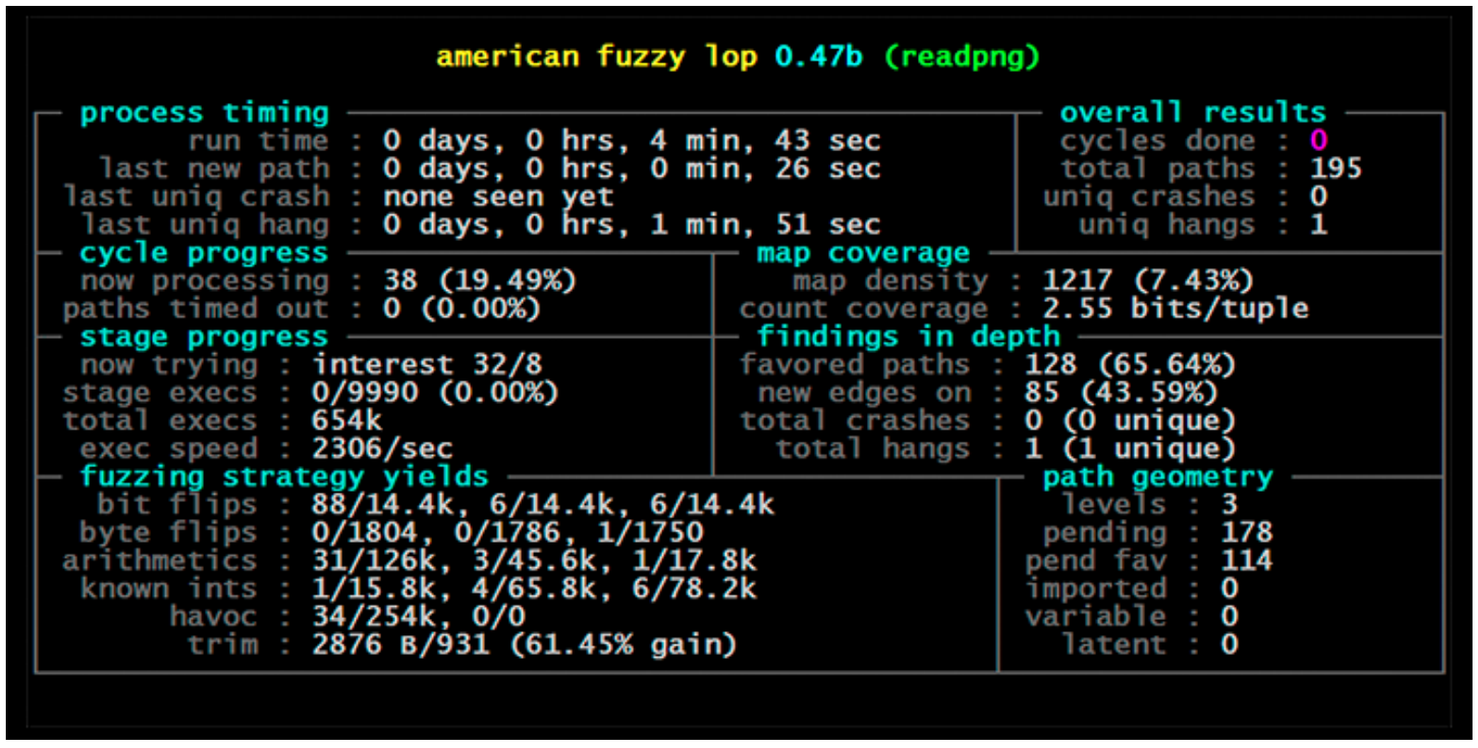
Popular, impactful project by Google.

Easy to set up with seed examples for mutation-based fuzzing.

Can instrument code for fast execution.

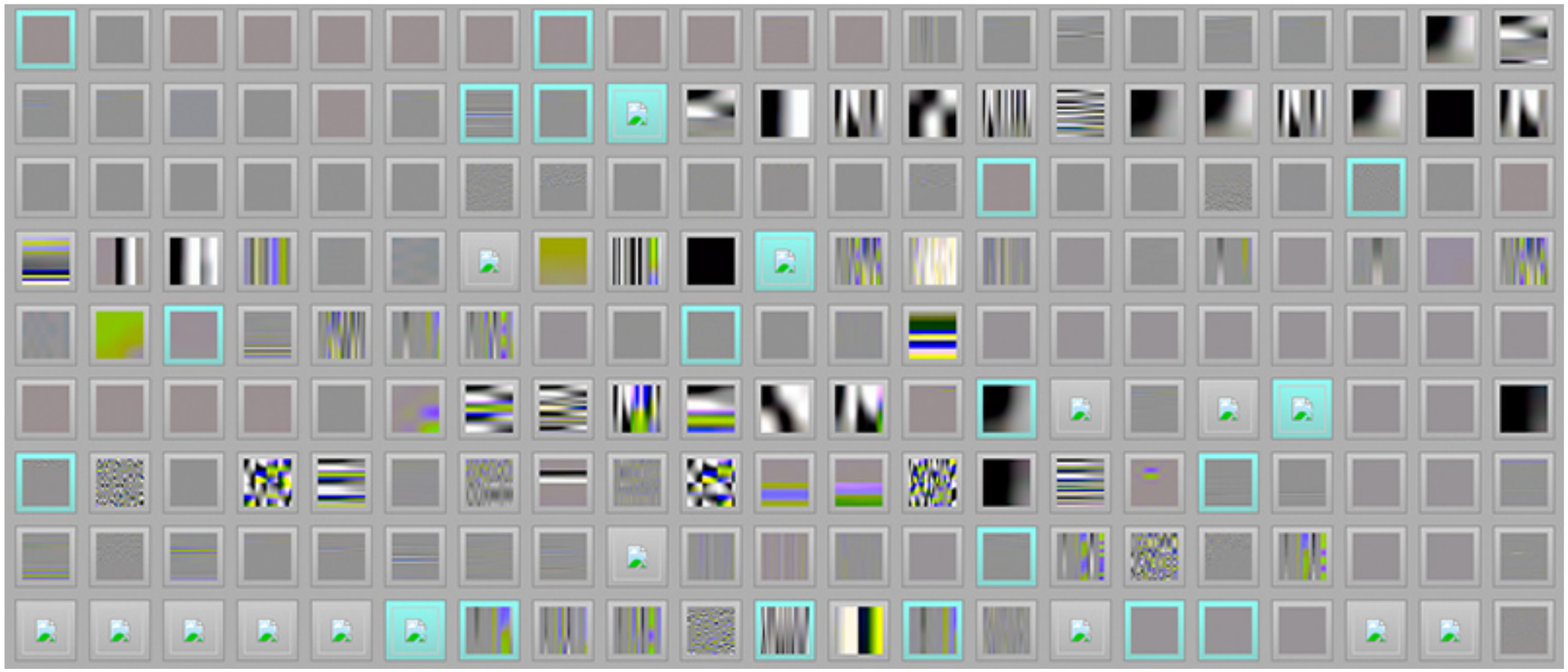
Deterministic bit-flipping, randomized stacked transforms.

Measures path coverage and favors increasing coverage.



# AFL Fuzz and File Formats

```
$ mkdir in_dir
$ echo 'hello' >in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```



Automatically discovered well-formed jpeg format by exploring code!

https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

# Fuzzing in Production

Large companies constantly fuzz products with dedicated servers.

Anecdote: Found 95 vulnerabilities in Chrome during 2011.

**OneFuzz**

## A self-hosted Fuzzing-As-A-Service platform

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a single command, which can be baked into CICD, developers can launch fuzz jobs from a few virtual machines to thousands of cores.

# Recap: Software Defenses

**Pre-deployment, before the program runs:** find or prevent bugs

- Fuzzing: proactively finding & fixing bugs by testing many program inputs
- Memory safe languages: automatically avoid exploitable memory bugs
- Not covered: Static and dynamic analysis

**Program runtime**: stopping exploits / violations of program's memory

- Stack Canaries, ASLR, W^X, etc
- Implemented by compiler (stack canaries) or OS (ASLR,W^X)
- Attacks adapt & evolve (reading the stack, ROP)

**Program runtime**: stopping exploits / violations of program's memory

- Stack Canaries, ASLR, W^X, etc
- Implemented by compiler (stack canaries) or OS (ASLR,W^X)
- Attacks adapt & evolve (reading the stack, ROP)

# The End