Final Review

CSE 565: Fall 2024
Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Exam details

- All relevant info will also be announced on Piazza
- **Final Exam**
    - Time: **Tue, Dec 17, 8-11:00 AM**.
    - Location:
        - **Knox 20** (for *both* section B & C)
    - Sit on your assigned seat (will be sent out later)

Exam details

- All relevant info will also be announced on Piazza
- **Final Exam Policy**
  - Things to bring:
    - **Allowed**: Pen, 1 page of A4-sized cheatsheet.
      - No scratch paper: you will use the backside of exam paper
    - **Required**: your UB card. You will not receive an exam paper until we verified your identity.
    - **Forbidden**: anything other than above, especially electronic devices.
      - If we found an electronic device with you after exam begins, you automatically fail the exam.

Exam details

- All relevant info will also be announced on Piazza
- **Final Exam Policy**
  - Academic Integrity
    - This is a closed-book exam. Usage of any electronic devices is forbidden.
    - You cannot speak to or communicate with any of the other students during the exam.
    - We will take video recording of the exam (especially when we are collecting exam papers at the end).
  - All violations are considered A.I. violation and the subject will receive 0 score for the exam. The case will also be reported to the A.I. office.

Exam details

- All relevant info will also be announced on Piazza
- **Final Exam Syllabus**
  - The exam will cover *all* topics we learned in this semester.
  - Weight:
    - 20%~30% for topics taught *before* the midterm
    - 70%~80% for topics *after* the midterm.
  - Amount of questions: ~2x compared with midterm.

Exam details

- All relevant info will also be announced on Piazza
- **Final Exam Syllabus**
  - Question types: similar to midterm
    - True or False (10~20%)
    - Multiple choice (20~30%)
    - Short answer questions (50%~60%)
      - All questions can be answered in a few sentences.
      - There won't be anything requiring a calculator.
      - The questions will be based on HW and Labs you've done, but they are ***not*** going to be exactly the same.

Review for Network Security

What to prepare

Review for Network Security

What to prepare

- Network basics
  - Protocol Layering Model:
    - What are the major layers?
    - What are the canonical protocols of each layer?
    - How are data being wrapped (unwrapped) when passed down (up) through the layers?

What to prepare

- Major Layers
  - Data Link Layer (Layer 2):
    - What are major protocols in this layer? (Ethernet, Wifi)
    - How is addressing done in this layer? (MAC address in local network)
    - What is a primary device facilitating routing in this layer? (Network switch; Routing achieved by simple port forwarding)
    - How do you know the MAC address of certain device? (via ARP, but needs the target IP)
    - What are the vulnerabilities of ARP? (no authentication, easy to spoof)

What to prepare

- Major Layers
  - Network Layer (Layer 3)
    - What are major protocols in this layer? (IP)
    - How is addressing done in this layer? (IP address; private IP vs public IP)
    - What is a primary device facilitating routing in this layer? (Router; Routing achieved by routing tables generated from BGP)
    - How do you know the IP address of certain host? (via DNS)
    - What are the vulnerabilities of IP / BGP / DNS? (no authentication, easy to spoof)

What to prepare

- Major Layers
  - Transport Layer
    - What are major protocols in this layer? (UDP / TCP)
    - How is addressing done in this layer? (IP address + port; from service / application to service / application)
    - How does TCP achieve reliable (ordered) byte transmission? (Sequence num. + Ack num.)
    - How are TCP connection established? (3-way handshake)
    - Vulnerabilities of
      - UDP?: basically IP + ports, inherits all vulnerabilities of IP, esp. easy to spoof.
      - TCP?:
        - Spoofing is harder: need to guess the correct combination of seq no. + ack no. + port no.
        - DDoS: SYN flooding.

What to prepare

- DNS
  - What is host name / domain name?
  - How is domain name resolving done? (Hierarchical, recursive resolution from root -> TLD -> authoritative server)
  - Vulnerabilities:
    - Spoofing / cache poisoning: fake DNS reply. Brute force is easier since the query ID is only 16 bits.
    - DNS rebinding attack: how does this relate to CSRF/XSS?

What to prepare

- DDoS Attack & Defense
  - Two types of DDoS attack? (amplification & flooding)
  - What is the basic idea of amplification attack? (asymmetry in UDP-type protocols: small request causes large response)
    - What are common targets exploited for amplification attack? (DNS; NTP)
  - Examples of flooding? (TCP SYN flooding)
  - Why is flooding becoming easier nowadays? (large no. of networking IoT devices with weak security measure)

What to prepare

- Secure Network Protocols:
  - IPSec: Layer? (IP). Purpose? (secure IP). How? (Authentication of payload and header; Encryption of payload).
    - Limits? (no end-to-end security: encryption ends at IP layer. AH makes NAT tricky since the header needs to be replaced)
  - TLS: Layer? (b/w Transport & Application). Motivation? (application-layer security; E2E security; easy network boundary traversal).
    - Canonical usage? (HTTPS)
  - VPN: Secure tunneling protocol, often builds on TLS or IPSec.

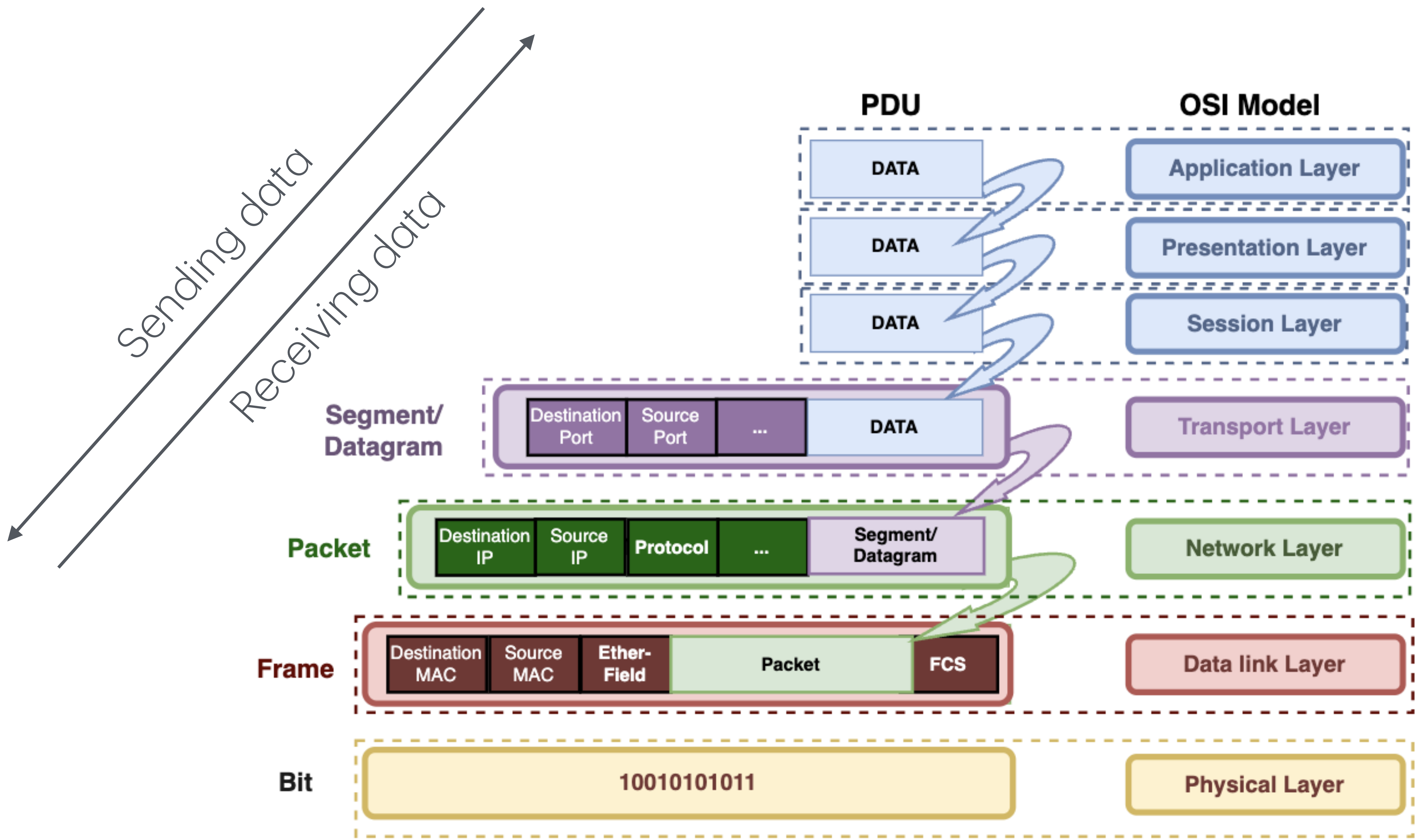
What to prepare

- Web privacy
  - What is 3p cookies? (cookie set by codes not from the current domain). How does 3p cookies enable tracking?
  - What is browser fingerprinting? Why is it hard to prevent?
- Tor: How does Tor prevents interceptors from learning the source & destination info of a packet?

Reference

Review for Network Security

The Protocol Layering



Layer 2: Link Layer

- Canonical protocols: Ethernet, WiFi
- Provides connectivity between hosts on a single **Local Area Network** (physically connected devices)
- Data is split into ~1500 byte Frames, which are addressed to a device's 6-byte physical (MAC) address — assigned by manufacturer
- **Switches** forward frames based on learning where different *MACs* are located. No guarantees that frames are not sent to other hosts!
- No security (confidentiality, authentication, or integrity)

ARP: Address Resolution Protocol

- ARP lets hosts to find each others' MAC addresses *on a local network*. For example, when you need to send packets to the upstream router to reach Internet hosts
    - Works only within a LAN.
- Client: Broadcast (all MACs): Which MAC address has IP `192.168.1.1`?
  Response: I have this IP address (sent from correct MAC)
- No built-in security. Attacker can impersonate a host by faking its identity and responding to ARP requests or sending gratuitous ARP announcements

Layer 3: Network Layer

- Canonical Protocol: Internet Protocol (IP)
- Provides routing between hosts on the Internet. Unreliable. Best Effort.
  - Packets can be dropped, corrupted, repeated, reordered
- Routers simply route IP packets based on their destination address.
  - Must be simple in order to be fast — insane number packets FWD'ed
- No inherent security. Packets have a checksum, but it’s non-cryptographic. Attackers can change any packet.
- Source address is set by sender—can be faked by an attacker

BGP (Border Gateway Protocol)

- Internet Service Providers (ISPs) announce their presence on the Internet via BGP. Each router maintains list of routes to get to different announced prefixes

- No authentication—possible to announce someone else's network
  - Commonly occurs (often due to operator error but also due to attacks)

Communicating at the Link & Network Layer

- Summary
  - **Within a same LAN**
    - Only need Link Layer support: MAC address (from e.g. ARP)
    - Peer-to-peer or centralized (forwarded by a switch)
  - **Between different LANs**
    - Need Network Layer support: IP address (from e.g. DNS)
    - Go through routers: Network Address Translation

Communicate between LANs

- Devices in a same LAN shares a *same* public IP via the router
  - Usually they are bound to different ports of the router
  - E.g., a web server may bind port 80 of the router
- Anyone outside the LAN can *only* send msg to the router's public IP addr
- The router will forward the msg based on the receiving port
  - Usually involves *translating* the destination from **`public_ip:port_A`** to some **`private_ip:port B`**
  - Known as Network Address Translation (NAT)

Transport Layer

- Canonical Protocols: UDP & **TCP** (focus)
- Bytestream Model: A stream of bytes delivered reliably and in-order between applications on different hosts
- **Transmission Control Protocol (TCP)** provides
  - Connection-oriented protocol with explicit setup/teardown
  - Reliable in-order byte stream
  - Congestion control
- Despite IP packets being dropped, re-ordered, and duplicated

TCP Seq. No. and Ack. No.

- Two data streams in a TCP session, one in each direction
- **Sequence Number**: Bytes in each data stream are numbered with a 32-bit number.
  - The numbering starts with an random offset.
- **Acknowledge Number**: Receiver sends acknowledgement number that indicates data received
  - The value of the acknowledgment field in a segment defines the number of the *next* byte a party expects to receive.

TCP Attacks

- Packet ordering are based on header information sent in **cleartext**, no authentication / integrity
  - Leads to spoofing attacks: Session Hijacking
  - Needs to get IP, Port, & Seq Number correct
- TCP is **stateful**:
  - To implement reliable & ordered transmission & traffic control, the two sides need to maintain state info $\Longrightarrow$ consume resources
  - Leads to Denial of Service attack: SYN flooding; TCP Reset Attack

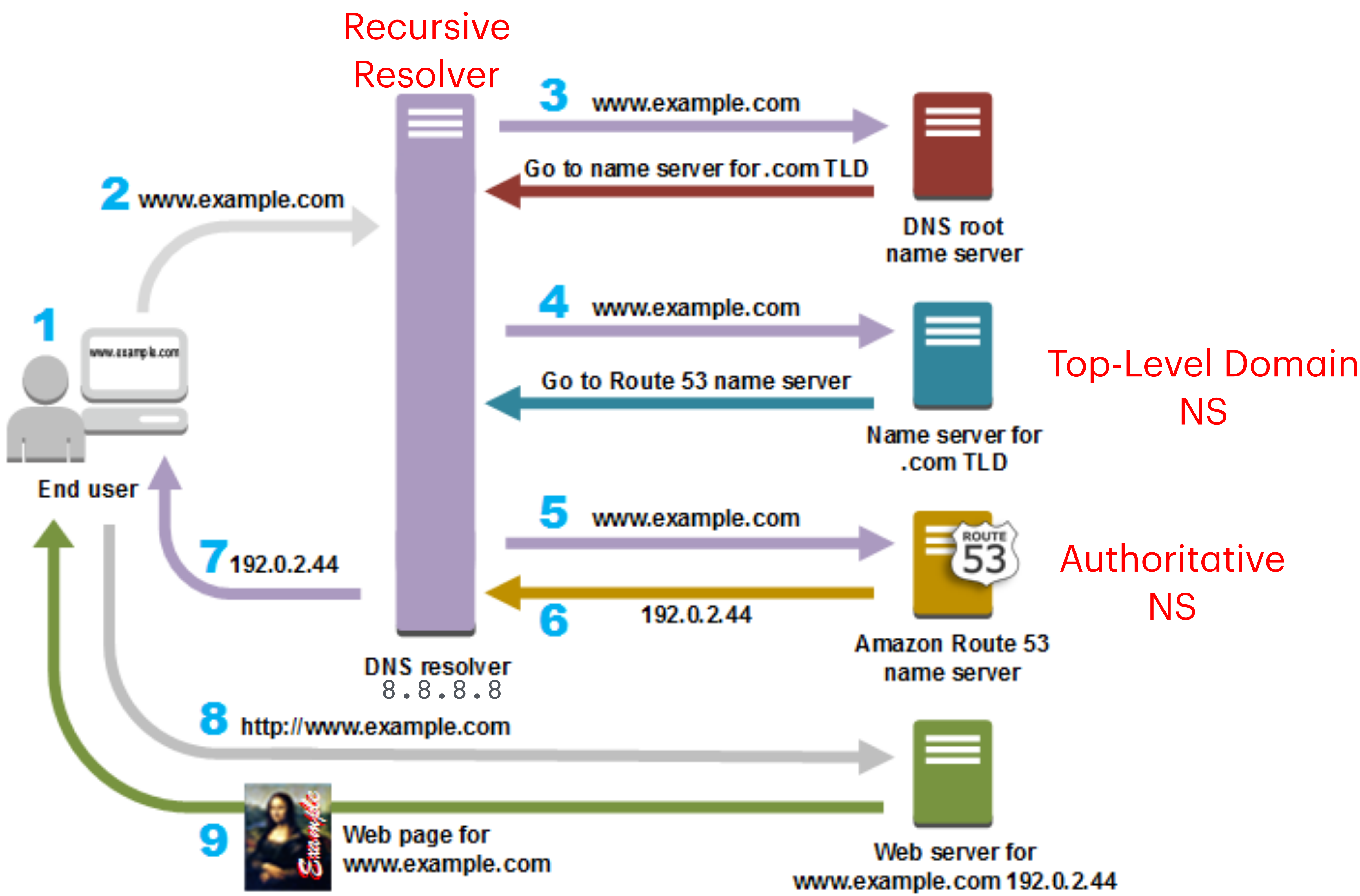
DNS (Domain Name Service)

- Map host name to IP; Similar role as ARP in Link Layer (map IP to MAC).
- Implemented as a distributed, delegatable, and hierarchical name space (database)

`www.buffalo.edu`

3rd Level Domain	2nd Level Domain	Top Level Domain
www	buffalo	edu

(Recursive) DNS resolution



DNS Security

- Again, like ARP, no built-in authentication / integrity.
  - Leads to DNS Poisoning attacks: Kaminsky Attack (Birthday attack)
- Working at the boundary of Application layer & Network layer
  - Able to circumvent application-layer security measures: DNS Rebinding attack
- Natural message amplifier: DNS response might be much larger than DNS request
  - Leads to Denial-of-Service attack (Today's topic)

DDoS Attack

- Denial-of-Service Attack: overwhelming the victim with huge network traffic
  - DoS via Amplification
    - Exploits asymmetries in network protocols: DNS, NTP
  - DoS via Flooding ("DDoS")
    - Generating traffic from many network devices ("botnet")
    - Getting more and more powerful due to the huge number of IoT devices today
    - Example: Mirai
  - DoS Defense: Anycast Network, Reverse Proxy, etc.

Secure Network Protocols

- **IPSec**: securing the Network Layer. Default in IPv6
  - AH (Integrity of the whole packet) , ESP (Confidentiality & Integrity of the payload), & IKE (secure session establishment).
- **VPN**: Secure *tunneling* protocols.
- **TLS**: Securing the Transport Layer.
  - The go-to solution for implementing secure channel: HTTPS, QUIC, VPNs
- **Firewall / IDS**
  - Packet filtering

Secure Network Protocols

- Expectation:
  - Know how crypto tools help defend against the vulnerabilities in vanilla network protocols
  - Example
    - How does IPSec prevent spoofing / man-in-the-middle?
    - How DNSSec prevents spoofing / DDoS?

Privacy & Anonymity

- **Web tracking**
  - 3rd-Party Cookies: how does that track users across site
  - Browser Fingerprinting: uses various unique browser and device characteristics, not just IP addresses, to identify users.
    - why is it difficult to prevent? Because many fingerprints have legitimate usage and are required by the website to correctly rendering contents.
- **Anonymity**
  - Tor network: enhances online anonymity by routing traffic through multiple volunteer-operated servers and encrypting data multiple times.
- Secure messaging app
  - E2E Encryption
  - Deniability

Review for Software Security

What to prepare

Review of Software Security

Guidelines

- Linux Process Basics: ELF; process loading with statical/dynamic link.
- Process Memory space: code segment, library, heap, stack
- Memory corruption:
  - Buffer overflow
  - Control hijacking: return-oriented programming
- Microarchitecture attacks

What to prepare

- **Linux Process Basics**
  - ELF: What is it? What's the purpose? (Tell OS how to correctly load the executable into memory).
  - Process loading
    - What is a loader/interpreter? (Resolving dynamic-linked libraries of an executable)
  - Process memory layout
    - What are the major parts? (Process binary code; (shared) library code; heap; stack; kernel reserved)
    - How is stack used? (for function call; grows from higher addr to lower addr)

What to prepare

- Memory corruption
  - What is it?
  - How does memory corruption leads to control flow hijacking? (Code and data exists together in the same memory address space)
  - What is stack smashing (overflow)? (Overflow the stack to overwrite control info stored on the stack, specifically, saved rbp and saved rip (ret address))
  - What does a function frame look like? *Where* is rbp and rip saved when you start a function call?
    - You should know how to calculate the number of input bytes needed to overflow a certain buffer.

What to prepare

- Return-Oriented Programing
  - What's the difference between ROP and shellcode? (ROP uses existing code gadgets living in the memory; Shellcode is injected by attackers)
  - Why is ROP more powerful than shellcode injection? (Not affected by NX; Most executables have abundant gadgets to utilize due to linked to libc)
  - Simple examples of ROP attack: given a list of gadgets, you should know how to initialize the stack to hijack the control flow, in order to chaining up the multiple gadgets.

What to prepare

- Mitigations for memory corruption
  - Stack canary: What is it? (a random value put just before the saved control info, i.e., saved rbp and rip). How does it stop buffer overflow?
    - How do you break it, especially in forking services?
  - ASLR: What is it? (A random offset added to the processes address).
    - How would you break it?
  - Control flow integrity: What is it?

What to prepare

- Microarchitecture security
  - You should understand that things can be different inside the CPU.
  - Data can persist for a while in the cache: What are primary features of CPU cache? (small and fast) How can you leak it (the side channel due to the access time difference between cache & RAM)
  - Instructions are not executed in order: What is out-of-order execution? What is a transient instruction? Why does transient execution can still leak info? What’s the idea of Meltdown?
  - Branching can be guessed (and fooled): What is speculative execution? How does CPU guess the result of branching? How do you bias CPU’s guess? What’s the idea of Spectre?

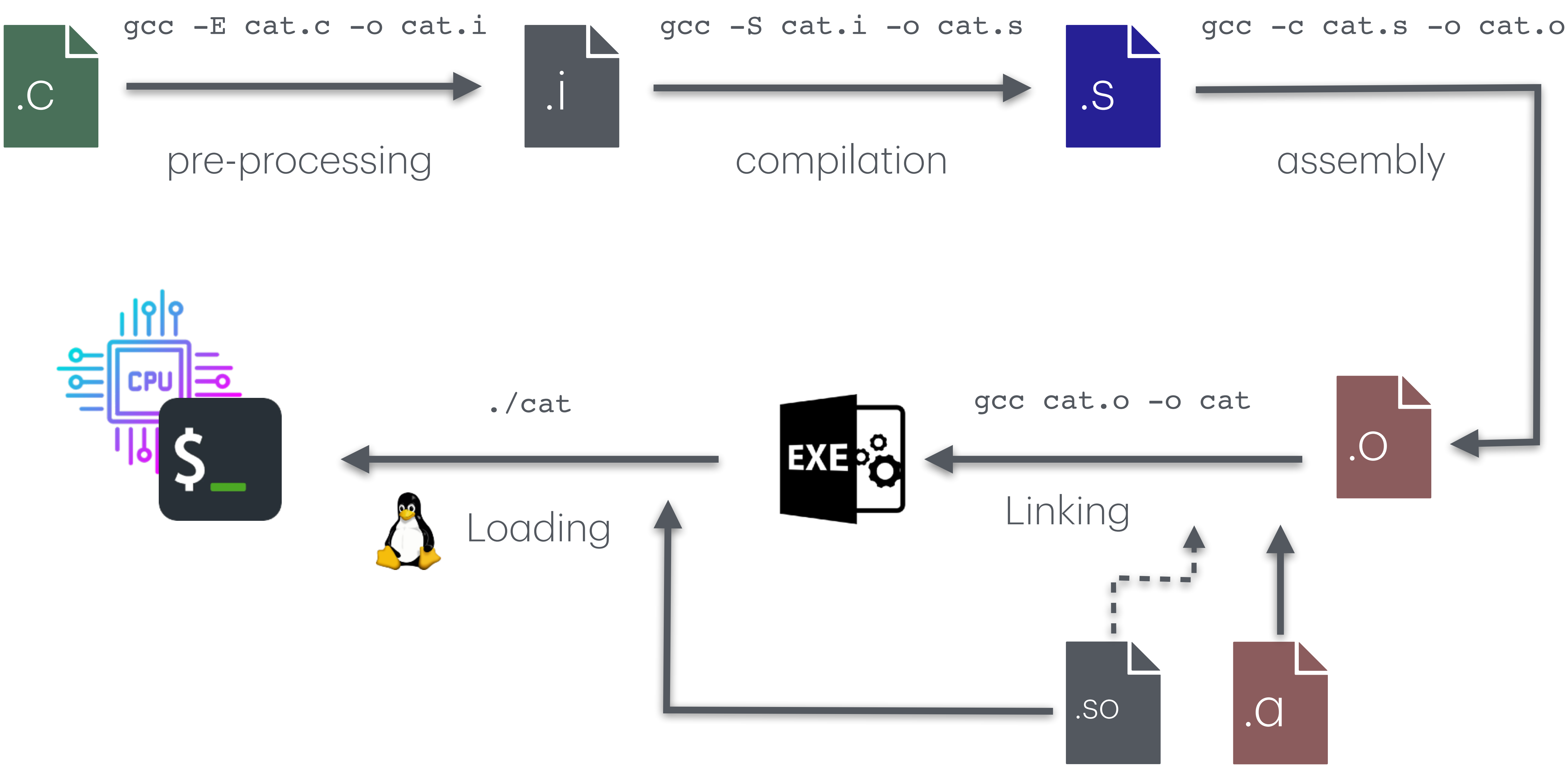
Reference

Review of Software Security

Linux process basics

- ELF: Executable and Linkable Format
  - Tells the OS how to execute a program
  - Program Headers: necessary for execution. Specifies the interpreter and how to load the executable into memory
  - Section Headers: Good for debugging
- Linux process Loading & Execution
  - Dynamic-Linked ELF: Kernel load (interpreter & executable) -> Interpreter load shared libraries -> run
  - Syscalls
  - Memory layouts

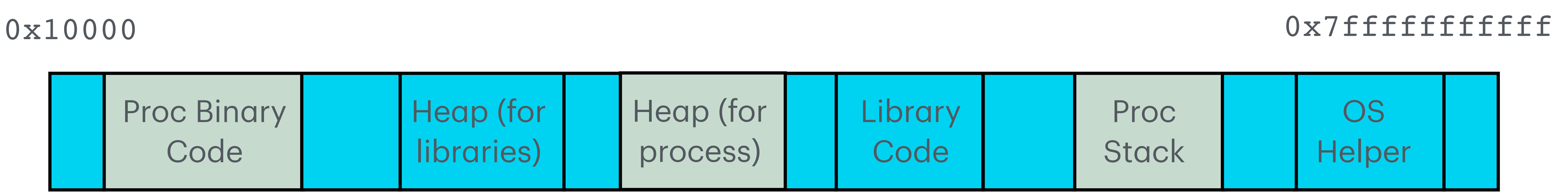
From a C program to a process



Memory: Process Perspective

- Memory is addressed linearly from **0x10000** (for security reasons) To **0x7fffffffff** (47 bits, for arch / OS purpose)
- A process' memory starts out partially filled-in by the OS

- The process can ask for more memory from the OS



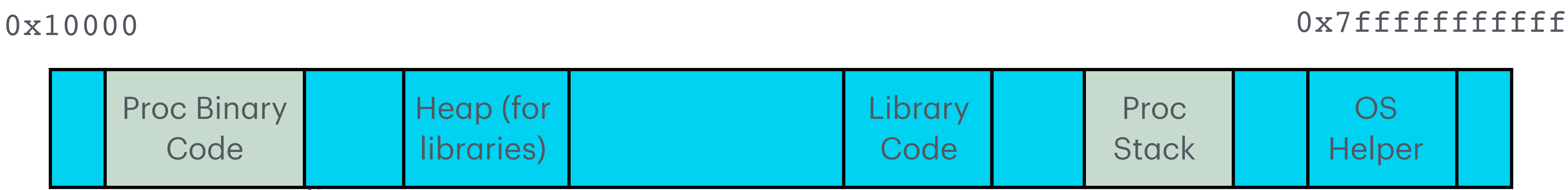
Addressing the Stack

- The memory address of the stack's top is stored in **`rsp`**

- Stack grows backwards: **`push`** decreases **`rsp`**, **`pop`** increases it.

Code also lives in memory

Assembly instr. are direct translations of binary code, which lives in *memory*.



- Example:

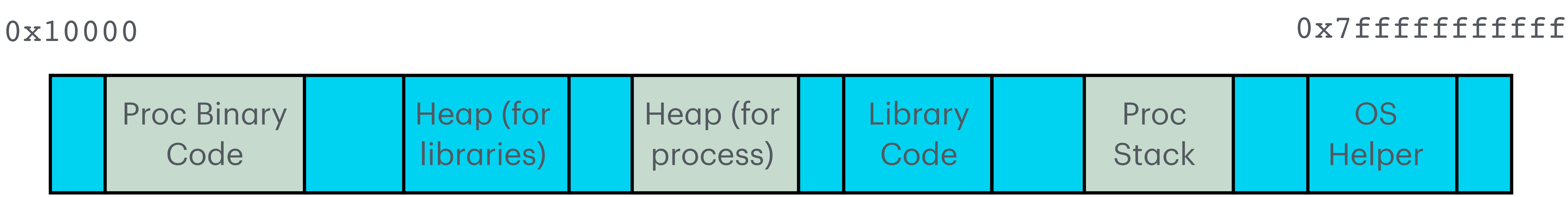
Proc Binary Code	`pop rax`	`pop rbx`	`add rax, rbx`	`push rax`	
	0x400800				

- In hex:

Proc Binary Code	0x400800	0x400801	0x400802	0x400805	
	58	5b	48 01 d8	50	

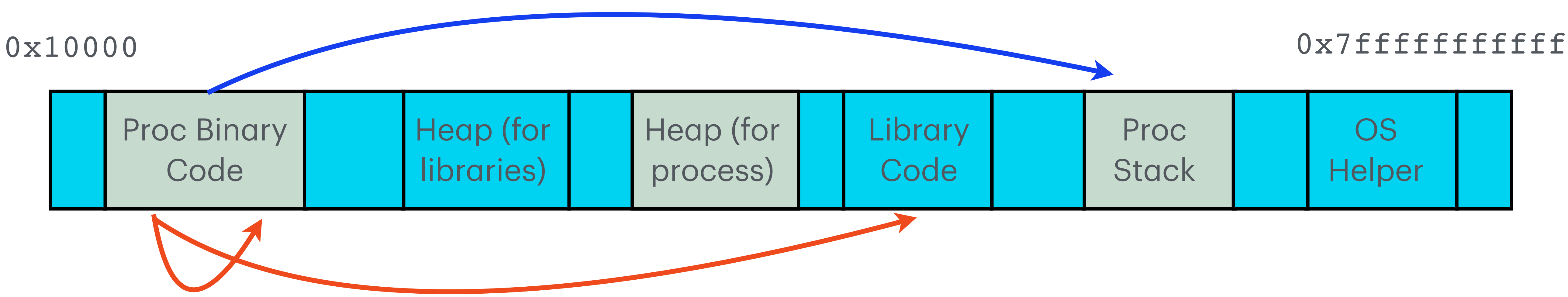
Memory corruption

- Software vulnerability that caused by accessing the memory in unintended ways. Prevalent in low-level languages like C or C++.
  - Attacker manipulate a program's internal state by forcing it to *read* or *write* data to memory locations beyond the intended boundaries.
    - Program code is stored in memory: direct attack
    - Control flow can depend on data in memory: var used in `if`, function `ret`urn addr, etc
    - Library codes are also in memory: used as gadget



Control-flow hijack

- Attacker overrides a **`ret`** address or **`jmp`** address to direct execution to a code segment of their choice
  - Return to code injected by attacker ("shellcode")
    - Prevented by *Non-Executable Data* policy
  - Return to existing code in memory: return-to-libc attack; Return Oriented Programming (ROP); Jump Oriented Programing (JOP)



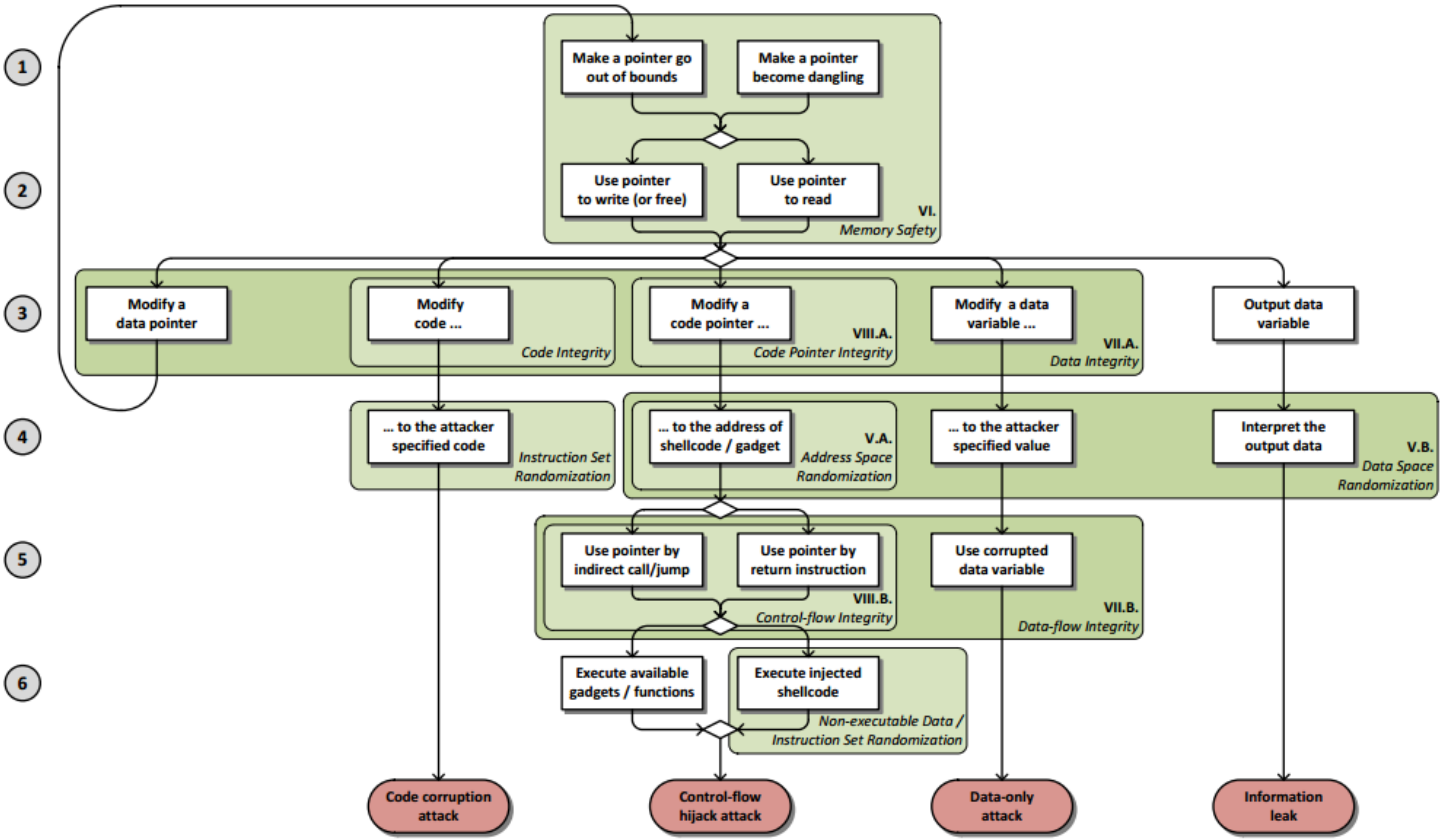


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Recall: the Stack

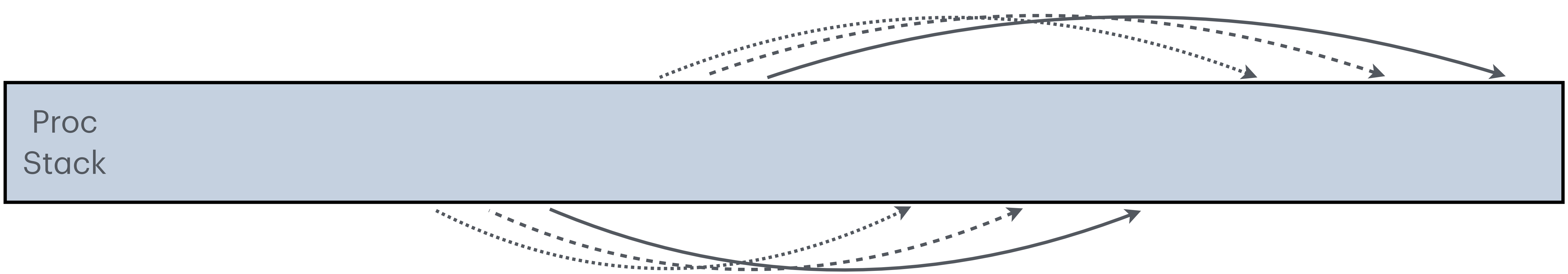
- Everything is jumbled together…
  - local variables of the active function
  - saved pointers to other places on the stack (**`rbp`**) or to data in memory
  - saved pointers to code (**`ret`**urn addresses)
  - local variables of the caller function (and its caller function and so on)
- All of this data is stored together and treated the same…

```
int a[3] = { 1, 2, 3 };
a[10] = 0x41;
```

The Stack: Initial Layout

- The stack starts out storing (among some other things) the environment variables and the program arguments. For example:

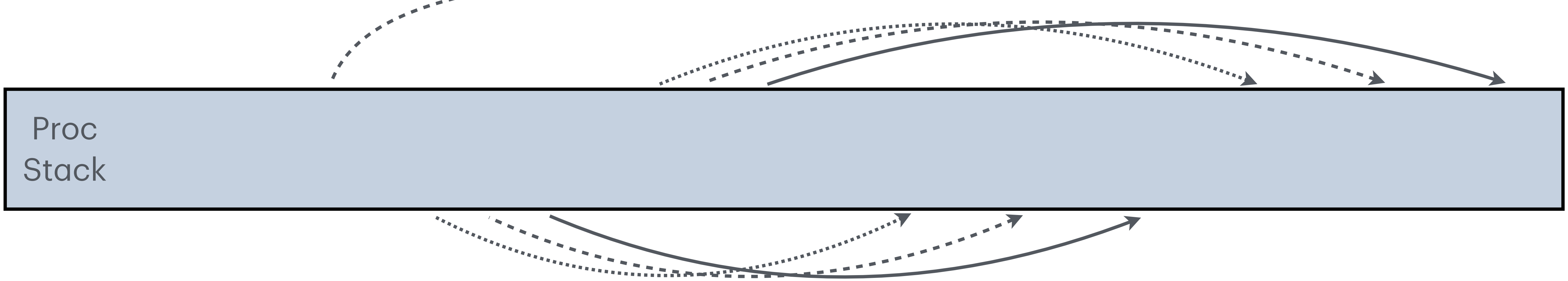
```
$ env
USER=seed
HOME=/home/seed
PWD=/home/seed
$ ./echo hello world
hello world
```



The Stack: Calling a function

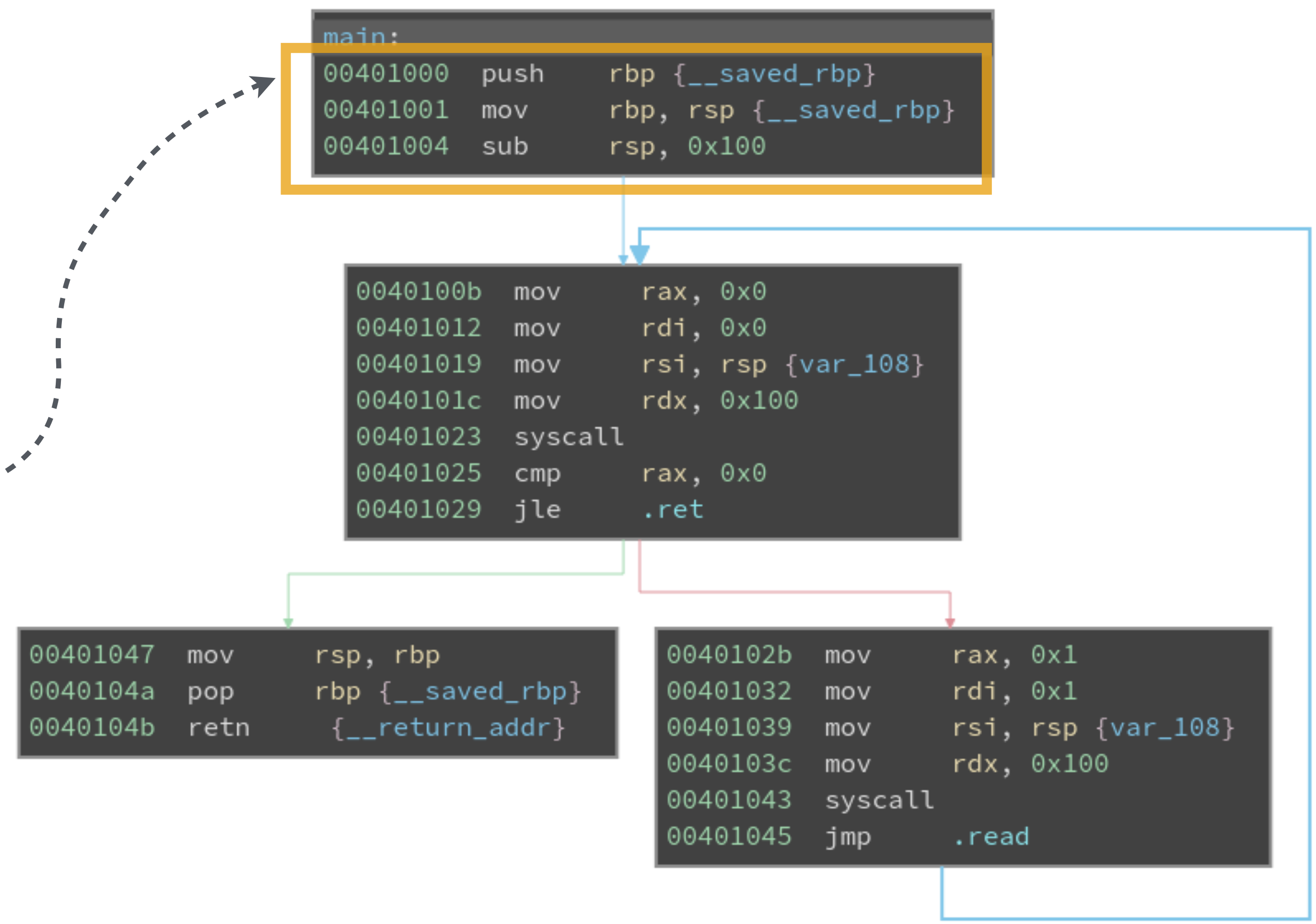
- When a function is `call`ed, the address that the called function should return to is implicitly `push`ed onto the stack.
- This return address is implicitly `pop`ped when the function `ret`urns.

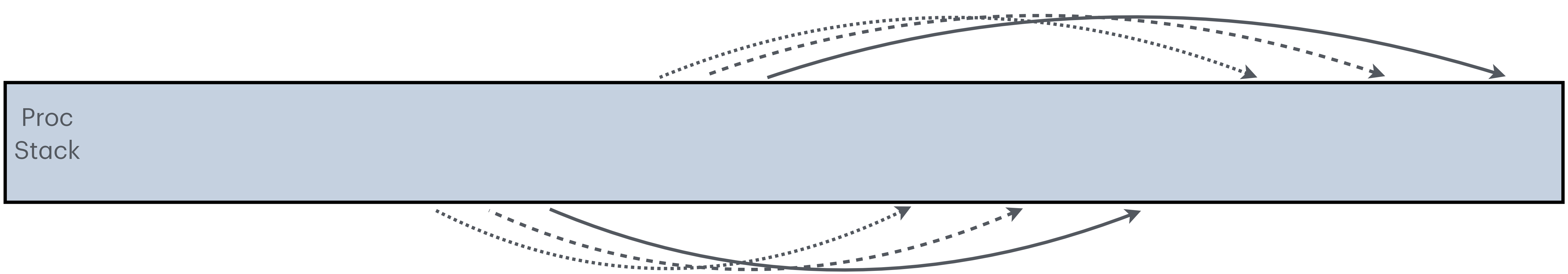
```
_start:
0040104c  mov     rdi, qword [rsp {__return_addr}]
00401050  call    main
00401055  mov     rax, 0x3c
0040105c  mov     rdi, 0x0
00401063  syscall
{ Does not return }
```



The Stack: Function Frame Setup

- Every function sets up its stack frame. It has:
  - **Stack pointer** (`rsp`): points to the leftmost side of the stack frame.
  - **Base pointer** (`rbp`): points to the rightmost side of the stack frame.
- **Prologue**:
  1. save off the caller's base pointer
  2. set the current stack pointer as the base pointer
  3. "allocate" space on the stack (subtract from the stack pointer).

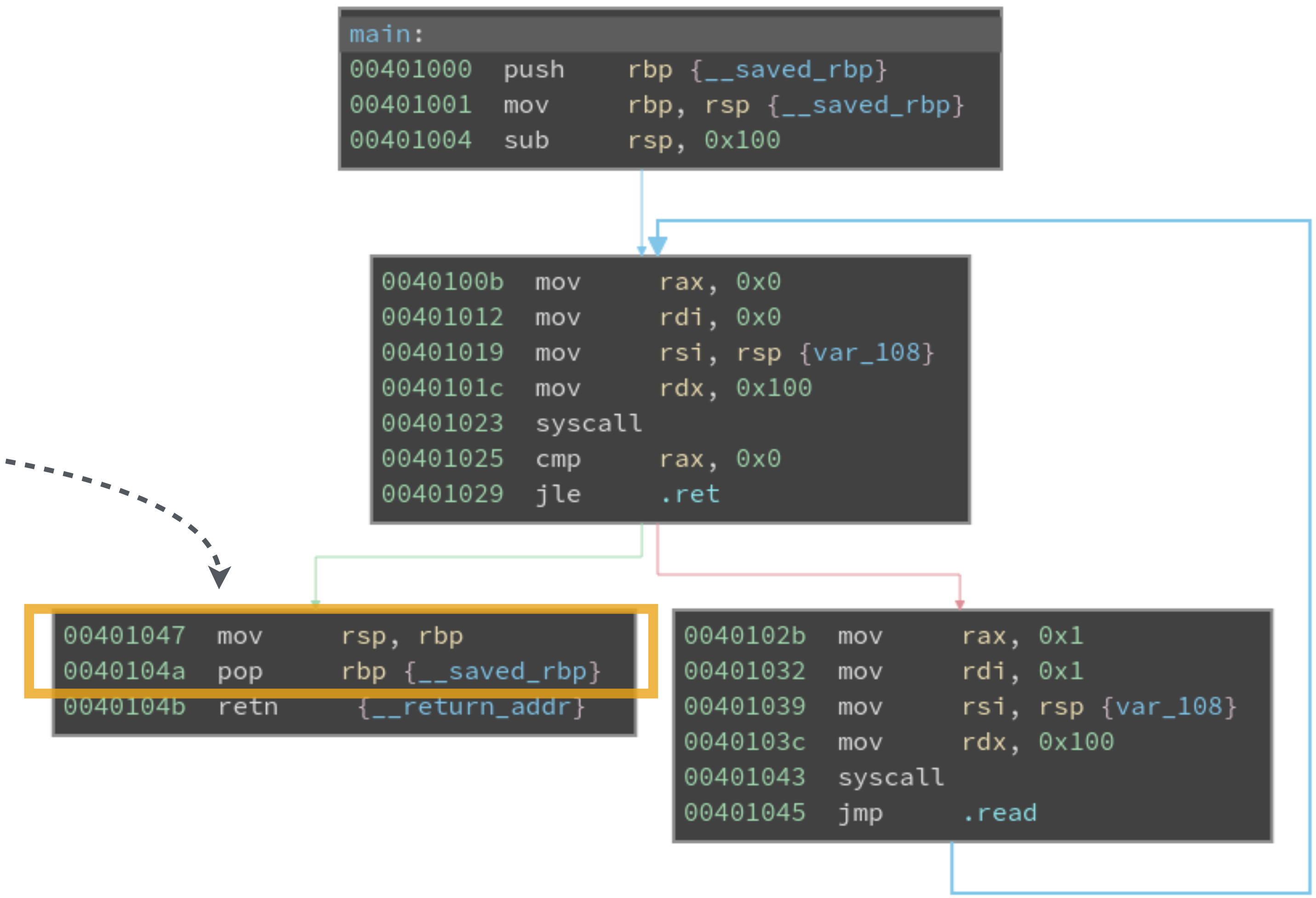


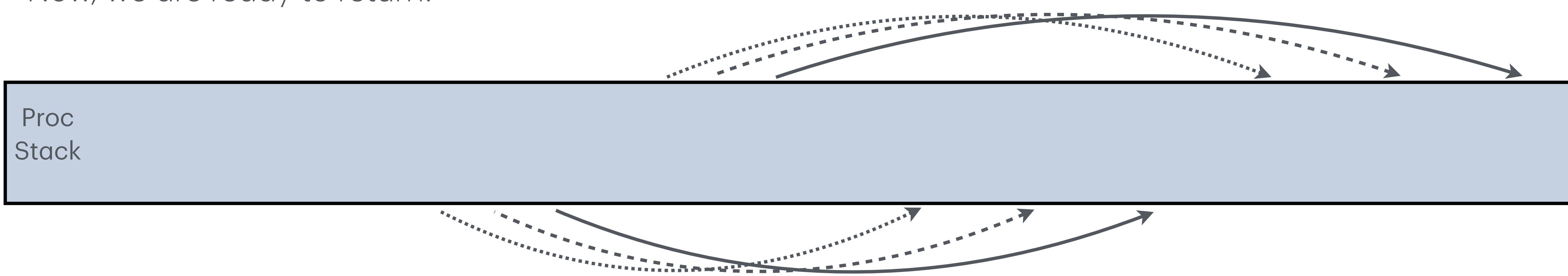


The Stack: Function Frame Teardown

- Every function sets up its stack frame. It has:
  - **Stack pointer** (`rsp`): points to the leftmost side of the stack frame.
  - **Base pointer** (`rbp`): points to the rightmost side of the stack frame.
- **Epilogue**:
  1. "deallocate" the stack (`mov rsp, rbp`).
     - note: the data is NOT destroyed by default!
  2. restore the old base pointer

Now, we are ready to return!





Example

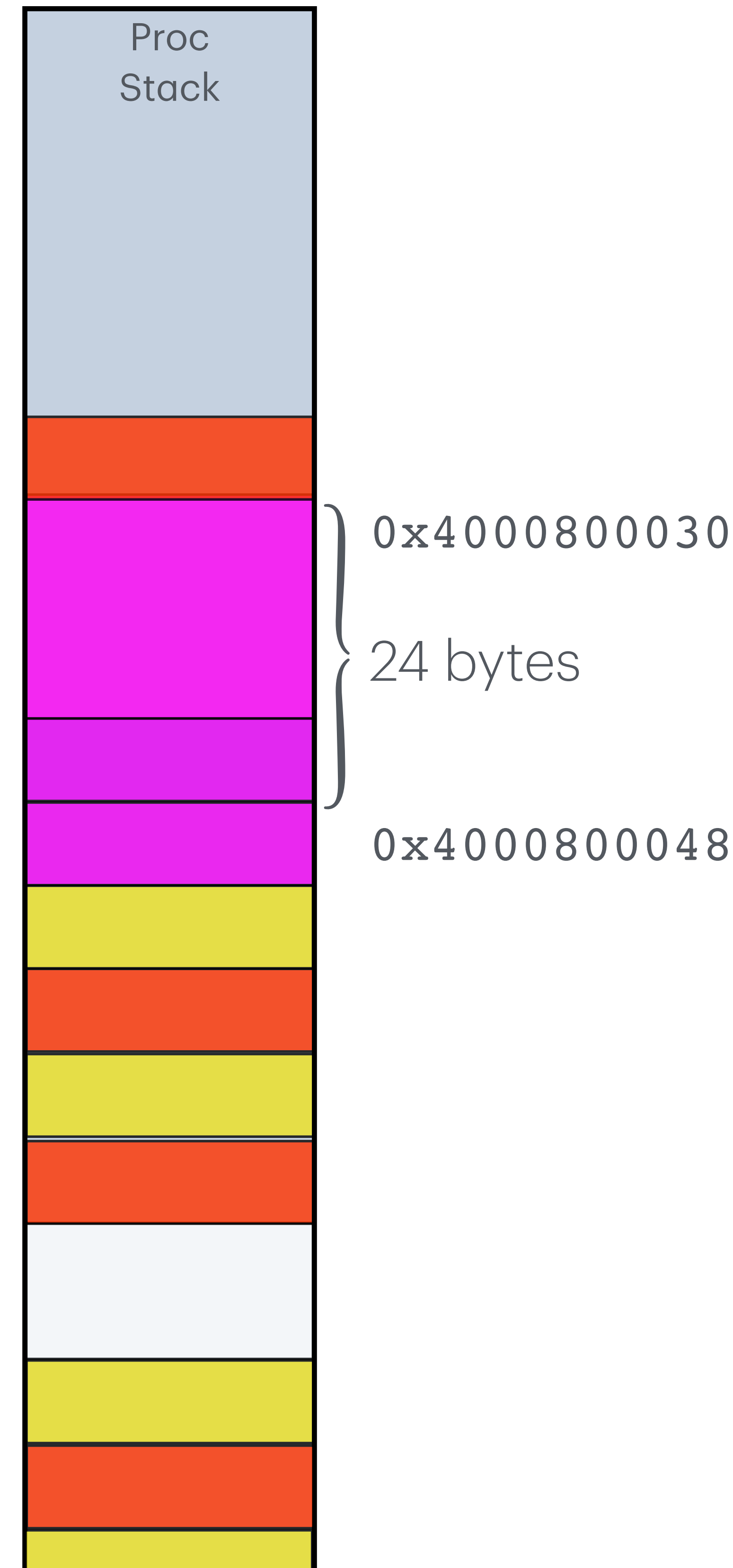
```
int main(int argc, char **argv, char **envp)
{
   print_actually(argv[1]);
   return 0;
}
void print_actually(char *s)
{
   printf(actually(s));
   return;
}
char * actually(char *s) {
   char output[16];
   sprintf(output, “Actually, %s", s);
   return output;
}
```

Proc
Stack

Example

Actual exploit

- Compile the vulnerable code:
  - `gcc -g -fno-stack-protector -no-pie print_actual.c -o print_actual`
    - Disabled **stack canary** and **ASLR**
- Use gdb & disassembler we find the addr of `output` (on stack) , the saved `ret` address (on stack `rbp + 8`), and the target addr of `win()` (see next slide)
- The exploit should start from `&output`, reach `&ret`, and make `*ret = &win`
  - `./print_actual (python3 -c "print('a'*14 + '\x2e\x12\x40')")`



Lab 4

- Stack layout understanding
  - Argument passing using register / stacks
- (**Partial**) ret address overwriting (for the 64-bit attack task)
  - saved ret = 0x0007ffff<u>abcd1234</u>
  - Target ret = 0x0007ffffa7654321

Memory corruption defense

RELocation Read-Only

Stack canary

Non-e**X**ecutable Memory

```
seed@seed-vm ~/Programs> checksec --file=cat
[*] '/home/seed/Programs/cat'
    Arch:       aarch64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    Stripped:   No
    Debuginfo:  Yes
```

Position **I**ndependent **E**xecutable (basically ASLR for the program's *own memory region*)

Stack Canaries

- Goal: To fight buffer overflows into the `ret`urn address.
  - In **function prologue**, write *random* value at the *end* of the stack frame. (immediately below saved `rbp`)
  - In **function epilogue**, make sure this value is still intact.
- Stack canaries are VERY effective in general.

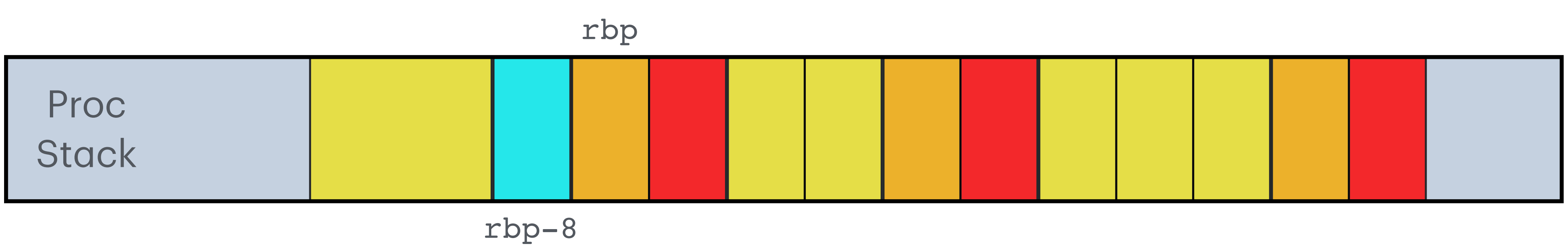
Stack Canaries

```
seed@seed-vm ~/Programs> checksec --file=buffer_overflow_x86
[*] '/home/seed/Programs/buffer_overflow_x86'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
    Debuginfo:  Yes
seed@seed-vm ~/Programs> ./buffer_overflow_x86
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
qemu: uncaught target signal 6 (Aborted) - core dumped
fish: Job 2, './buffer_overflow_x86' terminated by signal SIGABRT (Abort)
```

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 64
        mov     DWORD PTR -36[rbp], edi
        mov     QWORD PTR -48[rbp], rsi
        mov     QWORD PTR -56[rbp], rdx
        mov     rax, QWORD PTR fs:40
        mov     QWORD PTR -8[rbp], rax
        xor     eax, eax
        lea     rax, -32[rbp]
        mov     edx, 128
        mov     rsi, rax
        mov     edi, 0
        mov     eax, 0
        call    read@PLT
        mov     eax, 0
        mov     rdx, QWORD PTR -8[rbp]
        sub     rdx, QWORD PTR fs:40
        je      .L3
        call    __stack_chk_fail@PLT
.L3:
buffer_overflow_x86.s
```

Put canary (read from `fs:40`) at `rbp-8`

Before return to caller, check if canary is changed. If check failed then abort to `__stack_chk_fail`



Bypass canaries

- Situational bypass methods:
  - Leak the canary (using *another* vulnerability).
  - **Brute-force the canary (for *forking* processes)**.

```
int main() {
    char buf[16];
    while (1) {
        if (fork()) { wait(0); }
        else { read(0, buf, 128); return; }
    }
}
```

  - Jumping the canary (if the situation allows).

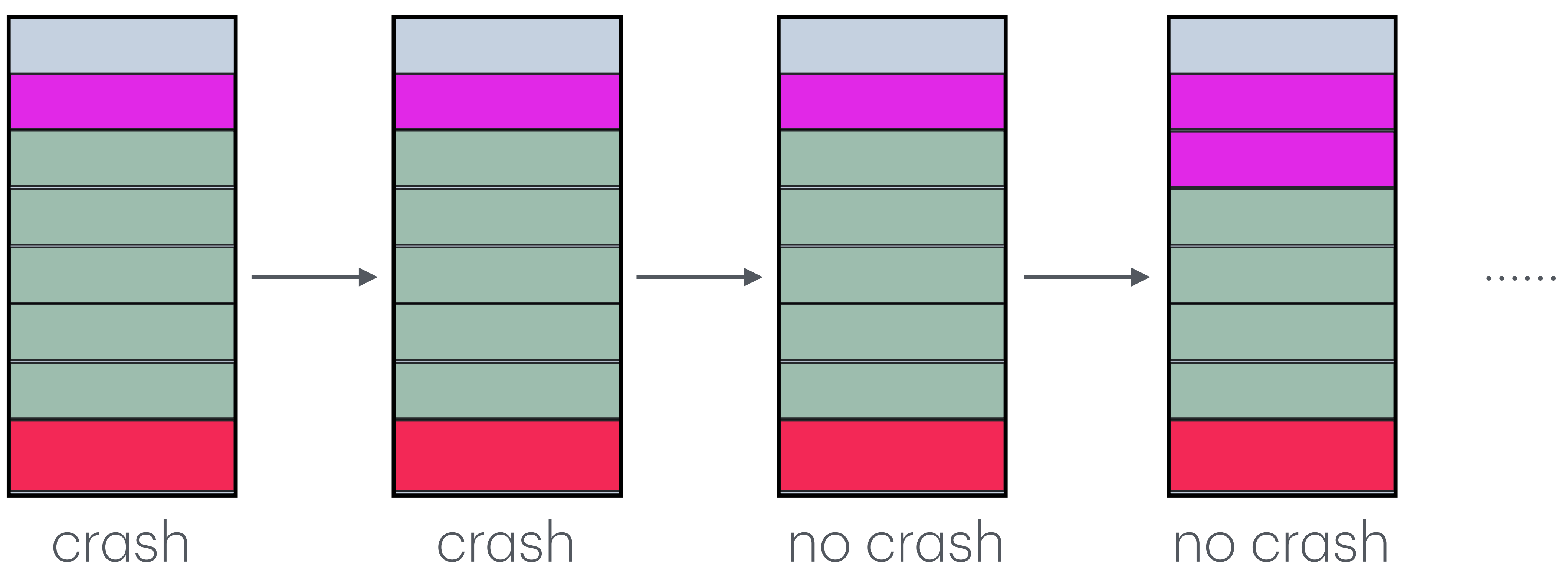
```
int main() {
    char buf[16];
    int i;
    for (i = 0; i < 128; i++) read(0, buf+i, 1);

}
```

  Depending on the stack layout, you can overwrite `i` and redirect the read to point to after the canary!

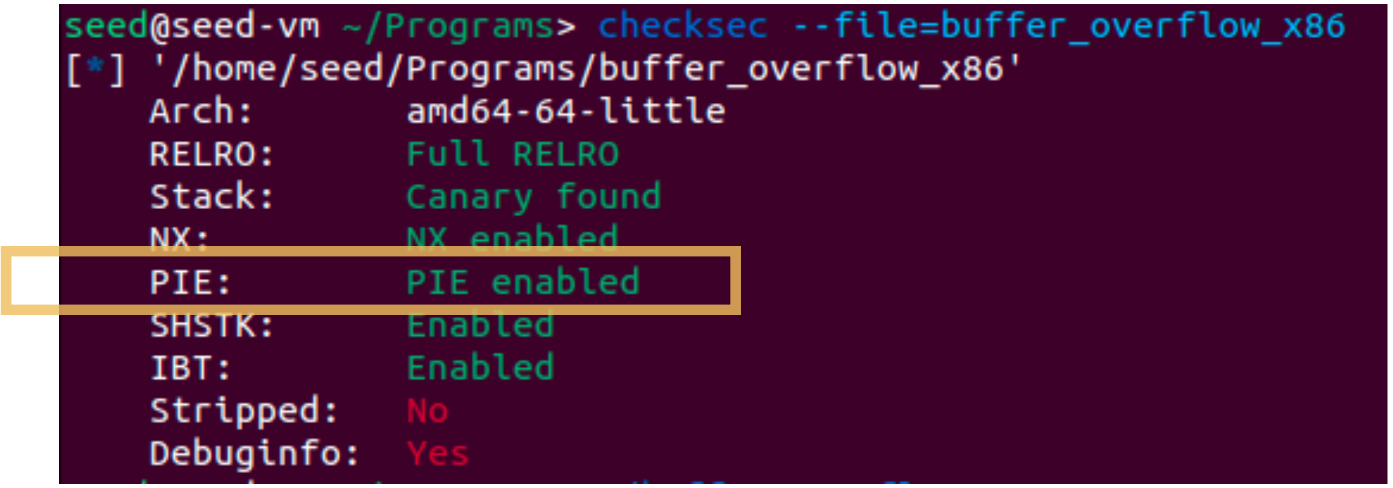
Bypass canaries for forking process

- Forking process Examples:
  - network services: one *same* server proc `fork`s a new child for each client connection)
  - Crash recovery: automatically relaunch a crashed child proc using `fork`
- Issue: Canary is often unchanged (always equal *the one used by the parent proc*)
- Result: Canary extraction **byte-by-byte**.



Address Space Layout Randomization (ASLR)

- Randomizes the base addresses of memory segments
  - Make it harder for an attacker to predict the location of important data, such as the stack, heap, or code segments.
  - Required to make PIE work
- Can be defeated similarly as canaries.



(Shell)Code injection

- Goal: subverts the intended control-flow of a program to previously *injected* malicious code
  - Code <u>supplied by attacker</u> – often saved in buffer *being overflowed*
  - “shellcode”: typical attack goal is to launch a shell: **`execve("/bin/sh", NULL, NULL)`**

```
mov rax, 59          # this is the syscall number of execve
lea rdi, [rip+binsh] # points the first argument of execve at the /bin/sh string below
mov rsi, 0           # this makes the second argument, argv, NULL
mov rdx, 0           # this makes the third argument, envp, NULL
syscall              # this triggers the system call
binsh:               # a label marking where the /bin/sh string is
.string "/bin/sh"
```

- Recall what you did in Lab 4.

ROP

- Chain chunks of code (gadgets; *not* functions; no function prologue and epilogue) in the memory together to accomplish the intended objective.

- The gadgets are not stored in contiguous memory, but they all end with a **`ret`** instruction or a **`jmp`** instruction.

- The way to chain they together is similar to chaining functions with no arguments. So, the attacker needs to control the stack, but does *not* need the stack to be executable.

ROP by induction

- **Step 0**: overflow the stack
- ......
- **Step n**: by controlling the return address, you trigger a gadget:
  - `0x004005f3:  pop rdi ; ret`
- **Step n+1**: when the gadget returns, it returns to an address you control (i.e., the next gadget)
- ....
- By chaining these gadgets, you can perform arbitrary actions in a ropchain!

Microarchitecture

- Things are different inside and outside CPU
  - Cache: side-channel for leaking memory
  - Out-of-order execution: something shouldn't be executed might got a chance to execute (Meltdown attack)
    - No architectural effect: registers won't be affected
    - BUT caches will.
  - Speculative execution: something shouldn't be executed might got a chance to execute (Spectre attack)
    - Biasing branch predictor
    - No architectural effect: registers won't be affected
    - BUT caches will.

Questions?