CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science
George Washington University

Course Review

Definition: Software Security

Allow *intended* use of software and prevent *unintended* use that may cause harm

Goal: Prevent information “mishaps”, but don’t stop good things from happening

- Good things include functionality (e.g. legal information access).
- Tradeoff between functionality and security is the key.

Bugs vs. Vulnerabilities

Wikipedia: *"A software bug is a bug in computer software."*

Wikipedia: *"In engineering, a bug is a design **defect** in an engineered system that causes an **undesired result**."*

Wikipedia: *"Vulnerabilities are **flaws** in a computer system that weaken the overall security of the system."*

Vulnerabilities -> Exploitable Bugs

https://en.wikipedia.org/wiki/Vulnerability_(computer_security)

Fact 1: Software Has Bugs

BLACK HAT

Windows Update Flaws Allow Undetectable Downgrade Attacks

Researcher showcases hack against Microsoft Windows Update architecture, turning fixed vulnerabilities into zero-days.

By Ryan Naraine
August 7, 2024

LAS VEGAS — SafeBreach Labs researcher Alon l
major gaps in Microsoft's Windows Update archi
hackers can launch software downgrade attacks
meaningless on any Windows machine in the wo

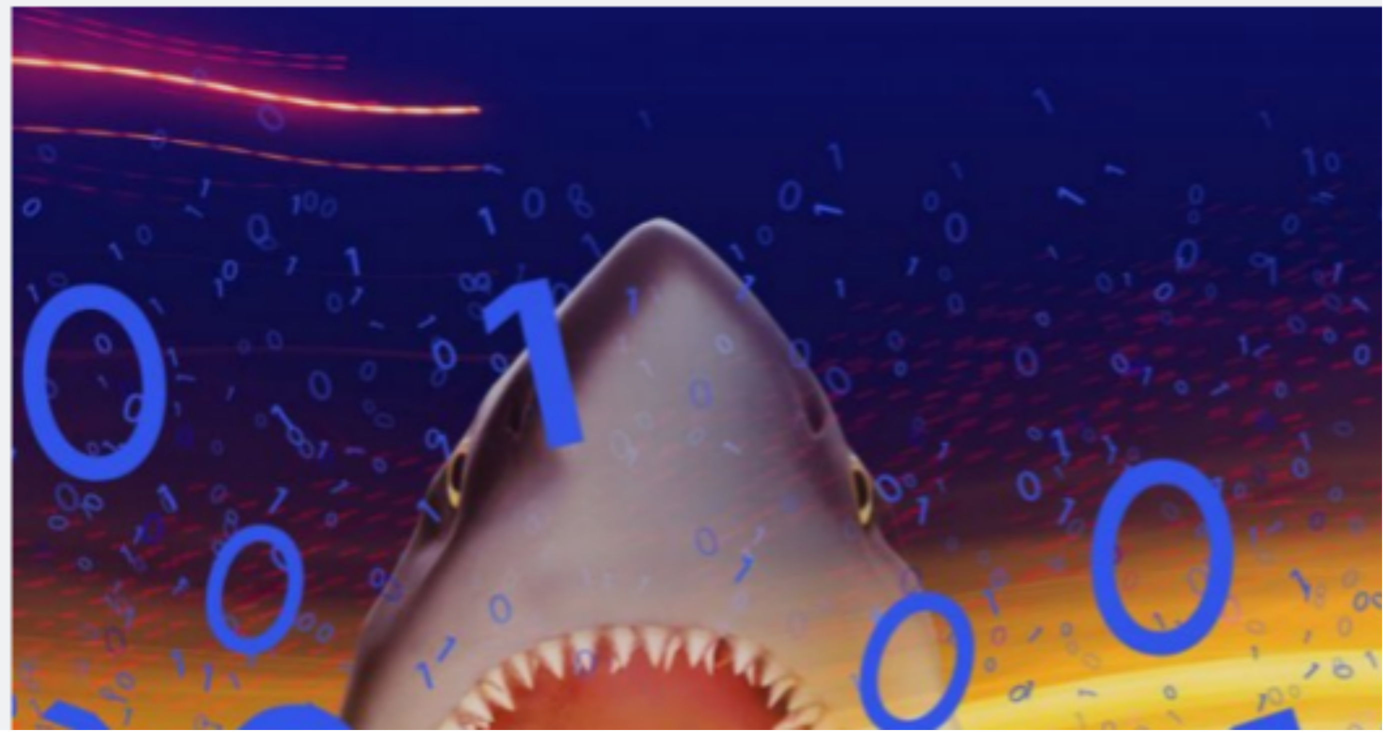
ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE

DIRTY PIPE —

Linux has been bitten by its most high-severity vulnerability in years

Dirty Pipe has the potential to smudge people using Linux and Linux derivitives.

DAN GOODIN - 3/7/2022, 6:39 PM



The Mac Security Blog

Search the Blog

Share

PRINT

APPLE

Apple still leaving critical vulnerabilities unpatched in macOS Sonoma

Posted on August 1st, 2024 by Joshua Long

As we first noted in November 2023, macOS Sonoma contains some very outdated open-source software components. (Free/libre open-source software is commonly abbreviated as FOSS or FLOSS.) This outdated software puts Mac users at serious risk. We've reached out to Apple multiple times about this, and Apple still hasn't responded. Here's what we know.

Popular Stories

- Porn blackmail "sextortion" emails: Have you been hacked? A new scam
- How to Install macOS Sonoma (or Sequoia) on Unsupported Macs, for Security Improvements
- The Complete Guide to Apple Watch Bands in 2024: Sizing, Styles, and More
- How to run Windows 11 for FREE on a Mac with an M1, M2, or M3 chip

Follow Intego

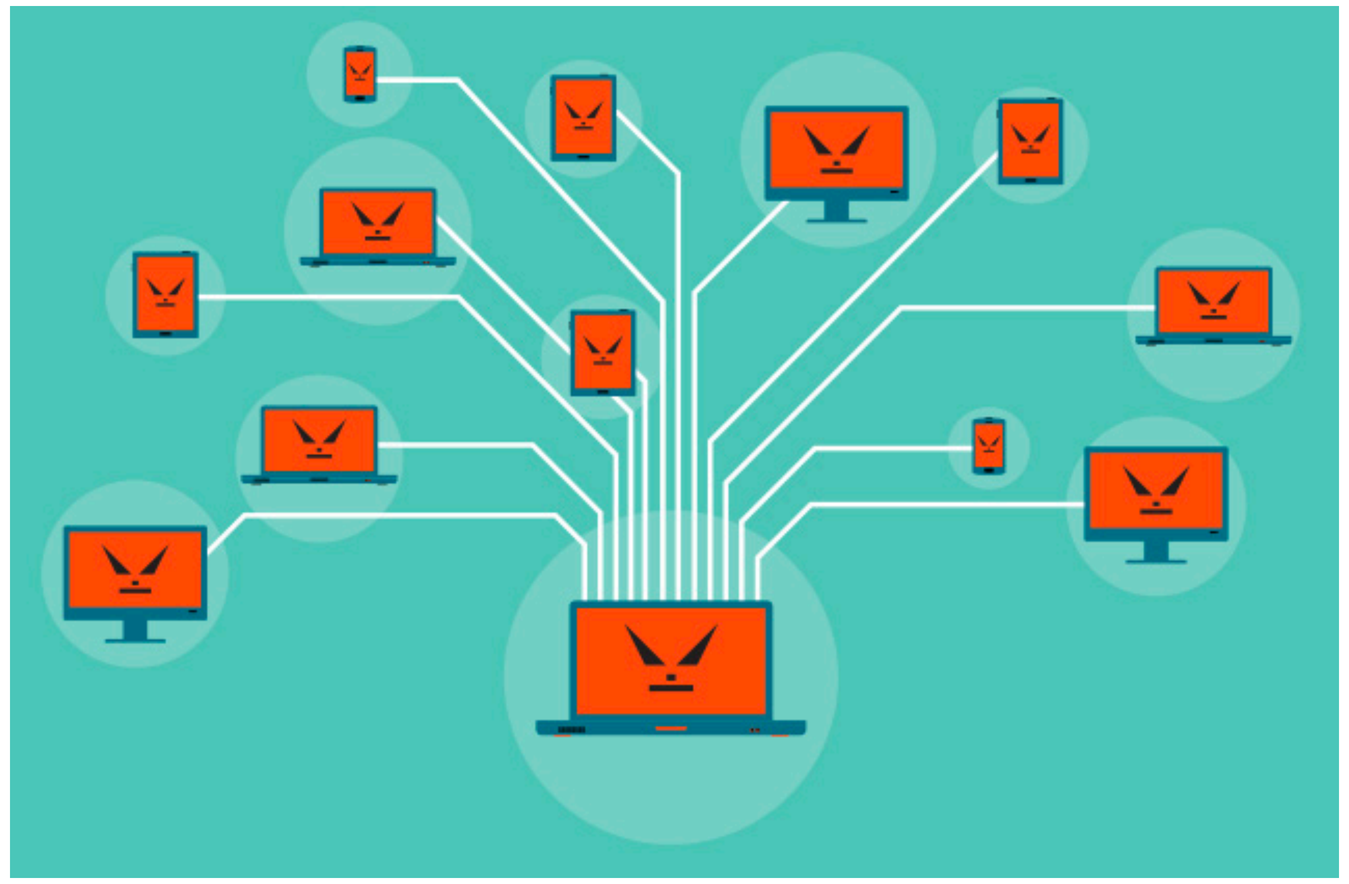
Recommended

SECURITY & PRIVACY

Fact 2: Many Bugs Are Exploitable (Causing Damage)

Ransomeware

e.g. WannaCry



Botnet

e.g. Mirai

Spyware

e.g. Pegasus

CIA Security Triad (+1)

- **Confidentiality**: An attacker cannot recover protected data.
- **Integrity**: An attacker cannot modify protected data.
- **Availability**: An attacker cannot stop/hinder computation.

Accountability/non-repudiation: Committed changes cannot be undone (as potential fourth fundamental property).

Fact 3: Software is Incredibly Complex

- Complexity
  - Software becomes more and more complicated.
  - Size is measured in terms of millions lines of code.
- Connectivity
  - The Internet makes it possible for attackers to exploit software remotely.
- Extensibility
  - Programs written by untrusted parties

Fact 3: Software is Incredibly Complex

- Complexity
  - Software becomes more and more complicated.
  - Size is measured in terms of millions lines of code.

- Connectivity
  - The Internet makes it possible for attackers to exploit software remotely.

- Extensibility
  - Programs written by untrusted parties

Do you trust computations provided by others?

Trusted Computing Base (TCB)

A set of hardware, firmware, and software that are critical to the security of a computer system.

- Bugs in the TCB may jeopardize the system's security
- E.g., a conventional e-voting machine: voting software + hardware

- Components outside of the TCB can misbehave without affecting the security of TCB.
- In general, a system with a smaller TCB is more trustworthy.
- A lot of security research is about how to move components outside of the TCB (i.e., making the TCB smaller)
  - E.g., Proof-Carrying Code removes the compiler outside of the TCB.

Definition: Threat Model

The abilities and resources of the attacker.

- Threat models enable structured reasoning about the attack surface.

- Awareness of entry points (and associated threats) to break into the target.
- Look at systems from an attacker's perspective:
  - Decompose application: identify structure
  - Determine and rank threats
  - Determine countermeasures and mitigations

Further reading:

https://owasp.org/www-community/Threat_Modeling

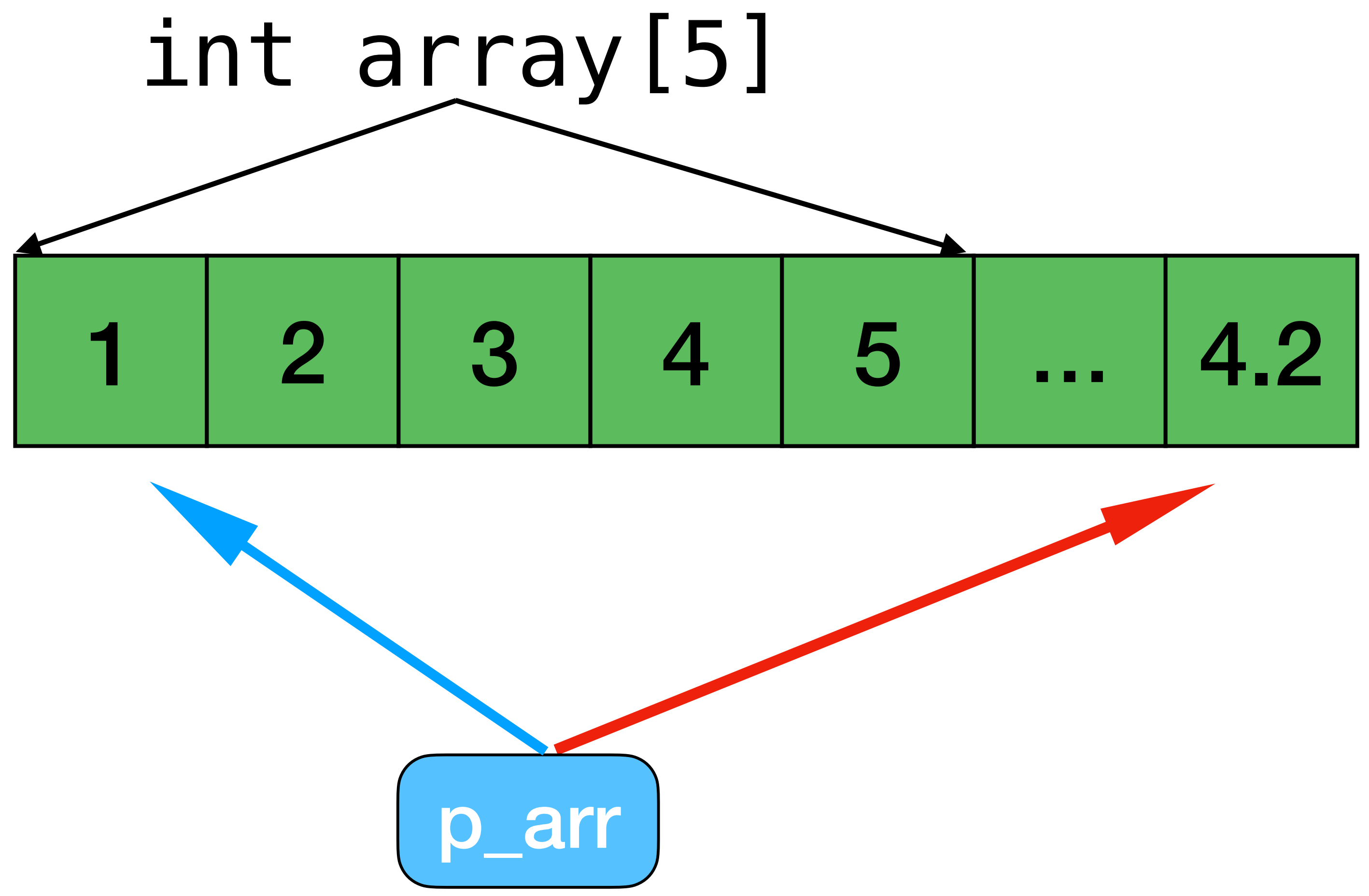
Memory Safety Taxonomy

- Spatial memory safety bugs
  - Buffer overflows / out-of-bound memory accesses
    - Stack buffer overflows
    - Heap buffer overflows
- Temporal memory safety bugs
  - Use-After-Free (UAF), the most common type
  - Double free
  - Invalid free
- Others
  - Null-pointer dereference
  - Format string bugs

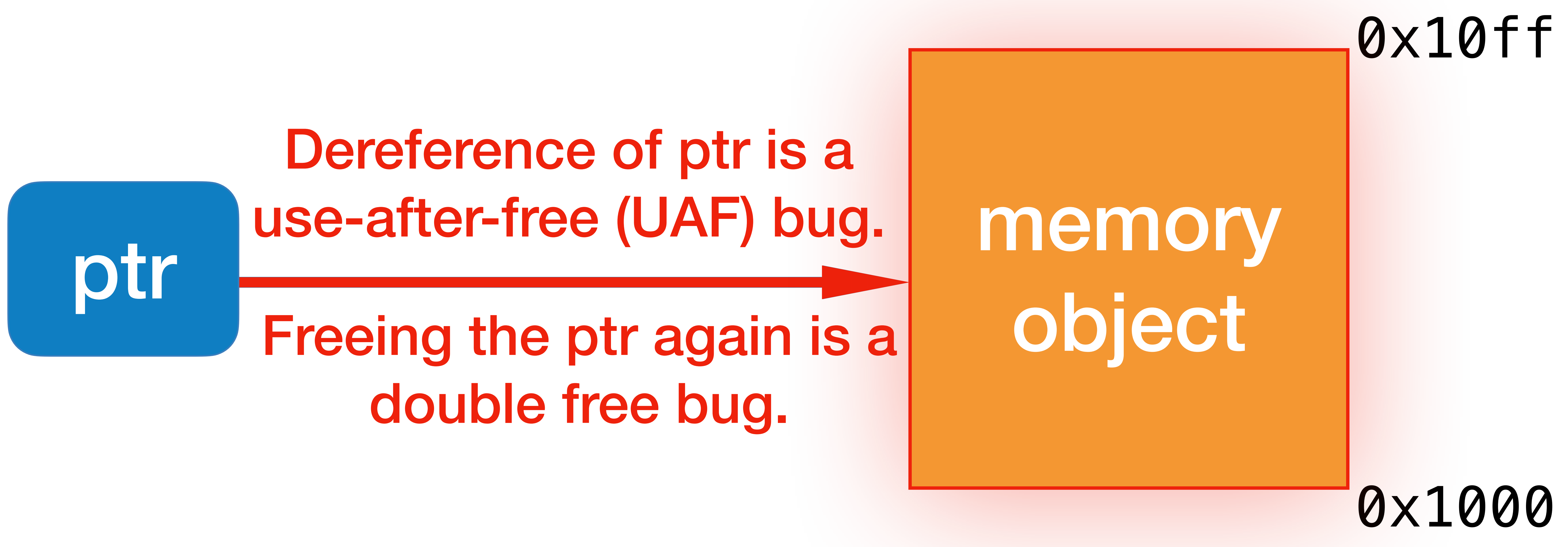
Null References: The Billion Dollar Mistake - Tony Hoare

Spatial Memory Safety Bugs: Buffer Overflows

Reading/writing a buffer out of its bounds.



Temporal Memory Safety Bugs



cwe.mitre.org/top25/archive/2024/2024_kev_list.html

Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

Top 25

Home > CWE Top 25 > 2024 CWE Top 10 KEV Weaknesses

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2024 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via: X | View in table format | KEV Key Insights | KEV Methodology

1 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 3 (up 2) ▲

2 Access of Resource Using Incompatible Type ('Type Confusion')
CWE-843 | CVEs in KEV: 6 | Rank Last Year: 8 (up 6) ▲

3 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-78 | CVEs in KEV: 6 | Rank Last Year: 5 (up 2) ▲

4 Improper Control of Generation of Code ('Code Injection')
CWE-94 | CVEs in KEV: 7 | Rank Last Year: 33 (up 29) ▲

5 Deserialization of Untrusted Data
CWE-502 | CVEs in KEV: 5 | Rank Last Year: 6 (up 1) ▲

6 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-22 | CVEs in KEV: 5 | Rank Last Year: 9 (up 3) ▲

7 Missing Authentication for Critical Function
CWE-306 | CVEs in KEV: 6 | Rank Last Year: 10 (up 3) ▲

8 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 4 | Rank Last Year: 11 (up 3) ▲

9 Use After Free
CWE-416 | CVEs in KEV: 5 | Rank Last Year: 1 (down 8) ▼

10 Improper Neutralization of Special Elements used in a Command ('Command Injection')
CWE-77 | CVEs in KEV: 4 | Rank Last Year: 15 (up 5) ▲

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

Home > CWE Top 25 > 2023 CWE Top 10 KEV Weaknesses

Home | About

2023 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via: | View in table format | KEV Key Insights | KEV Methodology

1 Use After Free
CWE-416 | Analysis score: 73.99 | # CVE Mappings in KEV: 44 | Avg. CVSS: 8.54

2 Heap-based Buffer Overflow
CWE-122 | Analysis score: 56.56 | # CVE Mappings in KEV: 32 | Avg. CVSS: 8.79

3 Out-of-bounds Write
CWE-787 | Analysis score: 51.96 | # CVE Mappings in KEV: 34 | Avg. CVSS: 8.19

4 Improper Input Validation
CWE-20 | Analysis score: 51.38 | # CVE Mappings in KEV: 33 | Avg. CVSS: 8.27

5 Improper Neutralization of Special Elements used in an OS Command
CWE-78 | Analysis score: 49.44 | # CVE Mappings in KEV: 25 | Avg. CVSS: 9.36

Long-standing Security Threats

1988 **Morris Worm**: infected 10% of Internet, exploiting an OOB bug as one key step

2012 **Heartbleed**: leaking secret data in servers/clients, OOB read in the OpenSSL library

2021 **NSO Zero-click**: remote code execution in iPhone, exploiting an OOB as a key step

Reports from Microsoft, Google, Apple, etc. consistently show that about ***65%–70%*** of their vulnerabilities are caused by memory safety bugs [1].

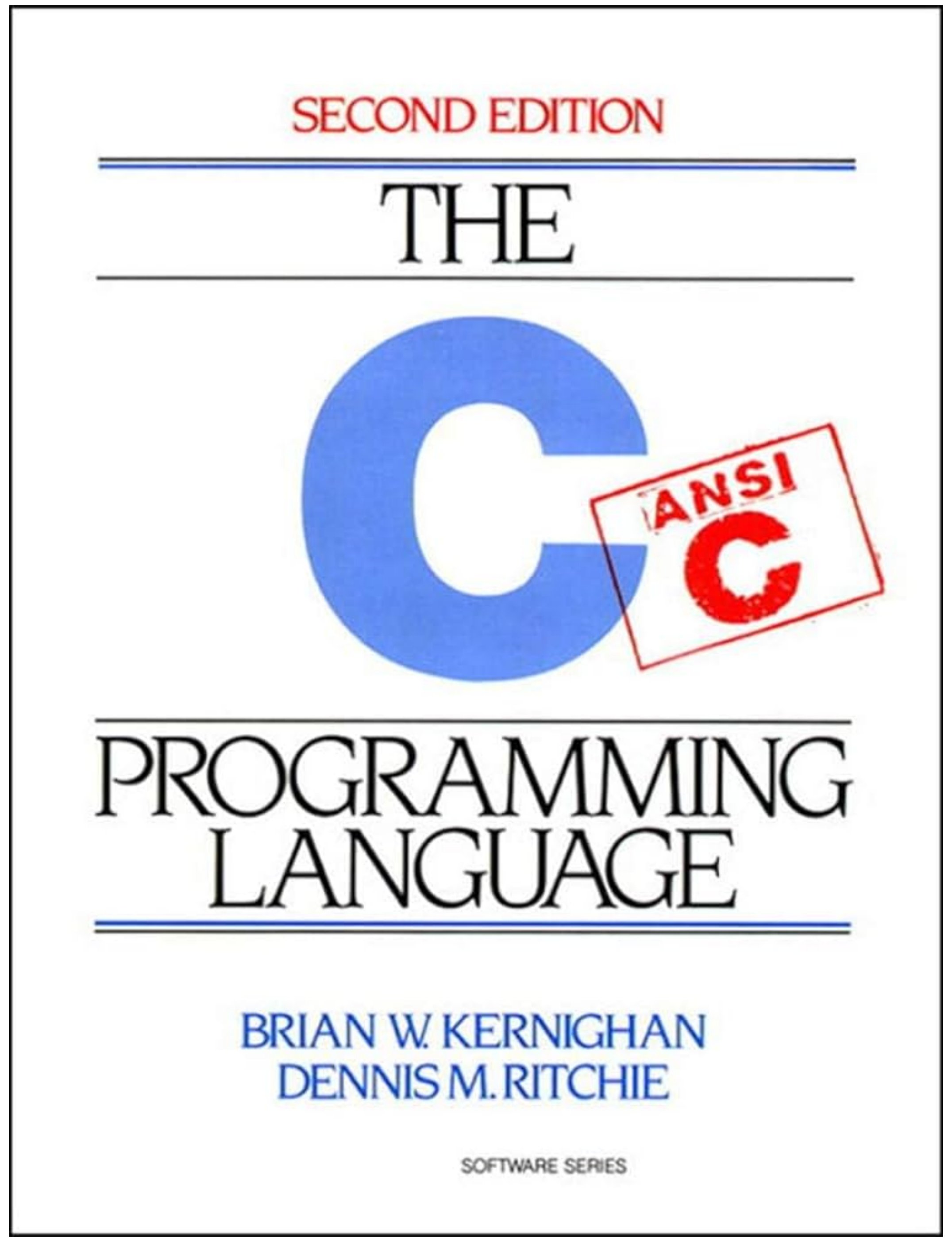
[1] Alex Gaynor. What science can tell us about C and C++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/

Exploits Against Memory

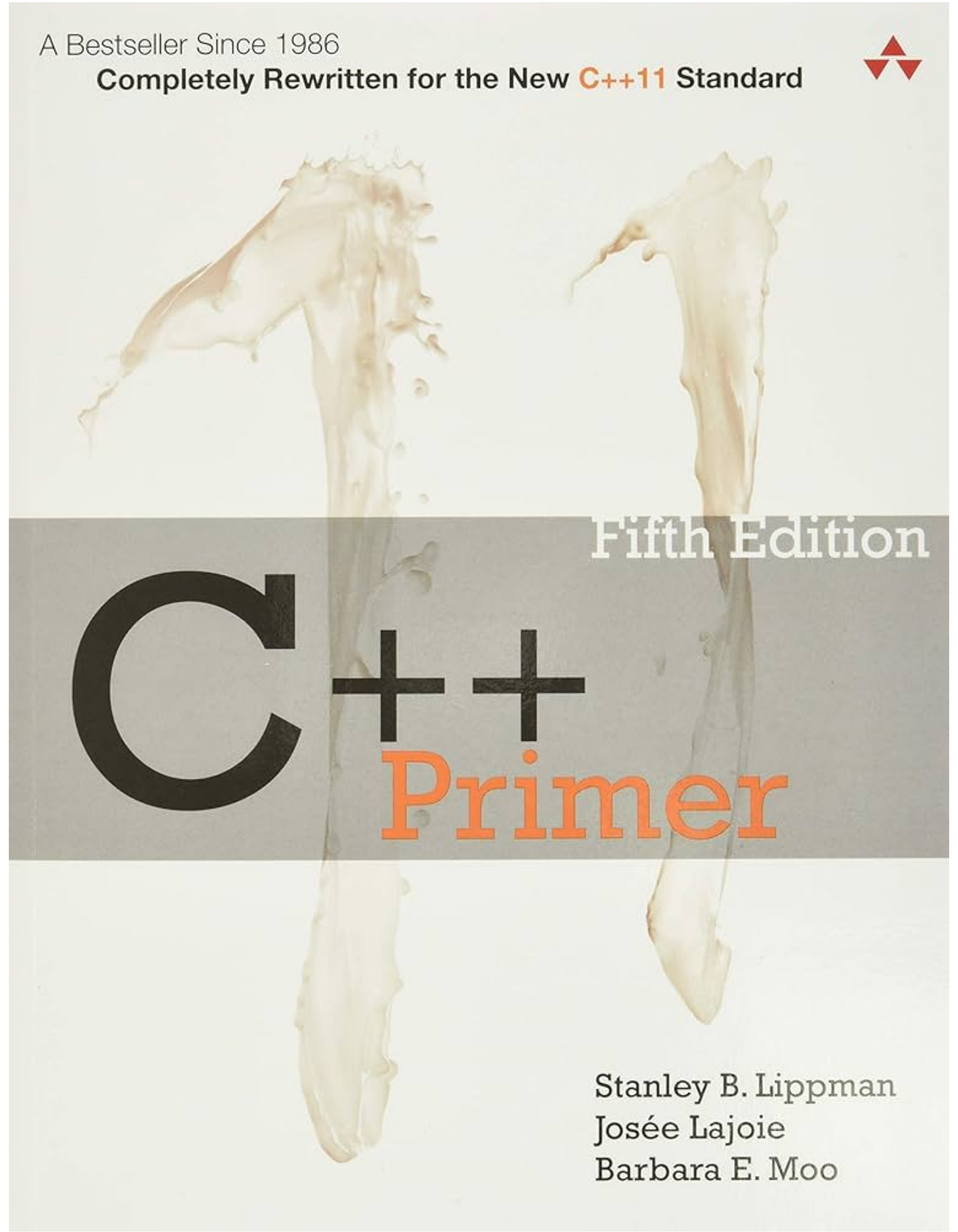
1995	Stack smashing to hijack the return address: Shellcoding
1997	return-to-libc
Early 2000	Use-After-Free (UAF)
2007	Return-oriented Programming
Today	Most commonly exploited: Heap buffer overflows and UAF

Why are there so many memory safety vulnerabilities?

Programming in C is Simple

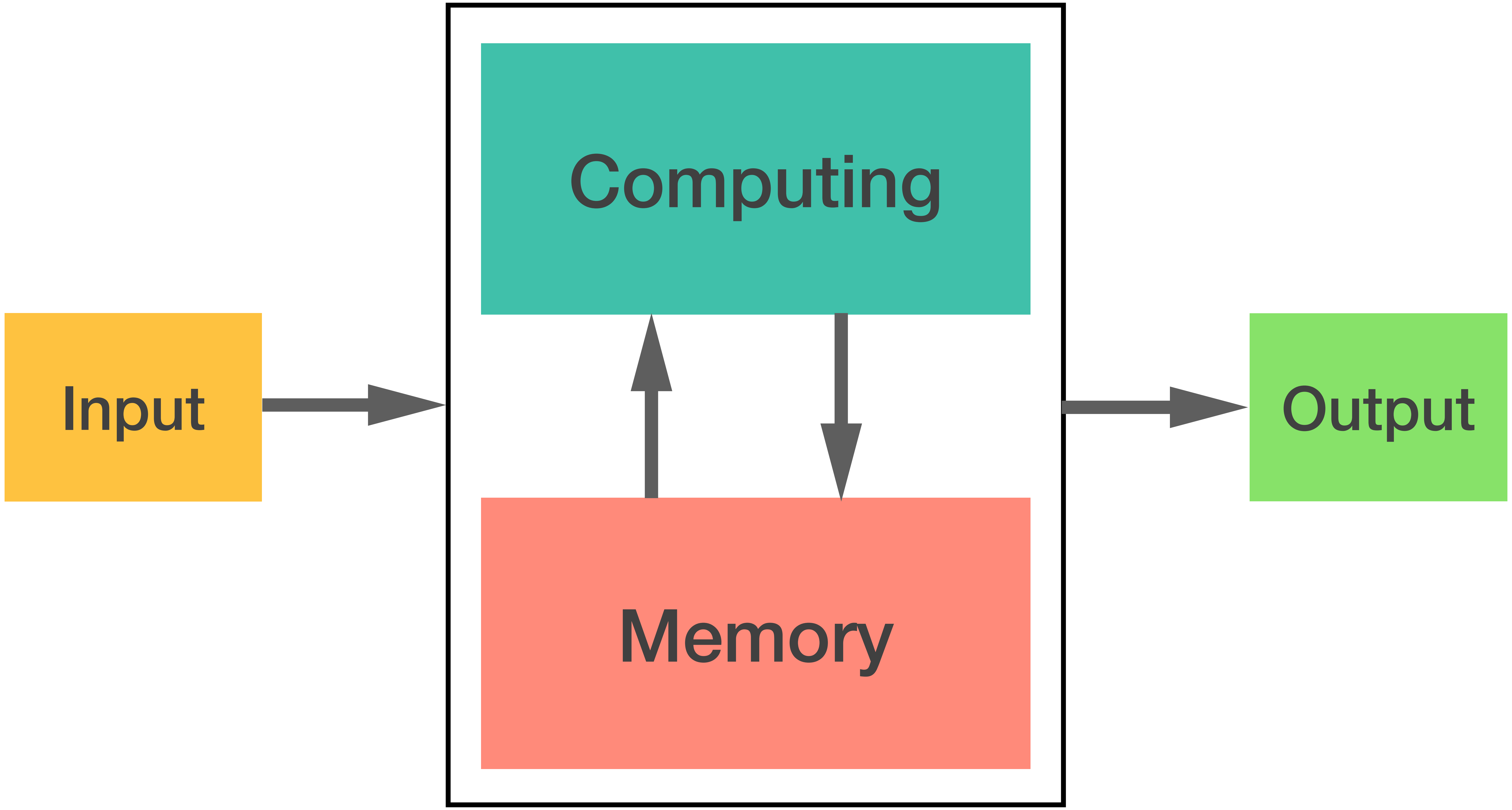


~200 pages



~1,000 pages

Architecture of Modern Computers



Programming *Correctly* in C is (Extremely) Hard

Simple and primitive language features

- Basic data types (char, integer, boolean, etc.)
- struct
- Pointers
- Basic control flow (conditional branches, loops, etc.)

Pointer: Capability to manipulate memory.

- For C, pointer is usually implemented as a virtual address.

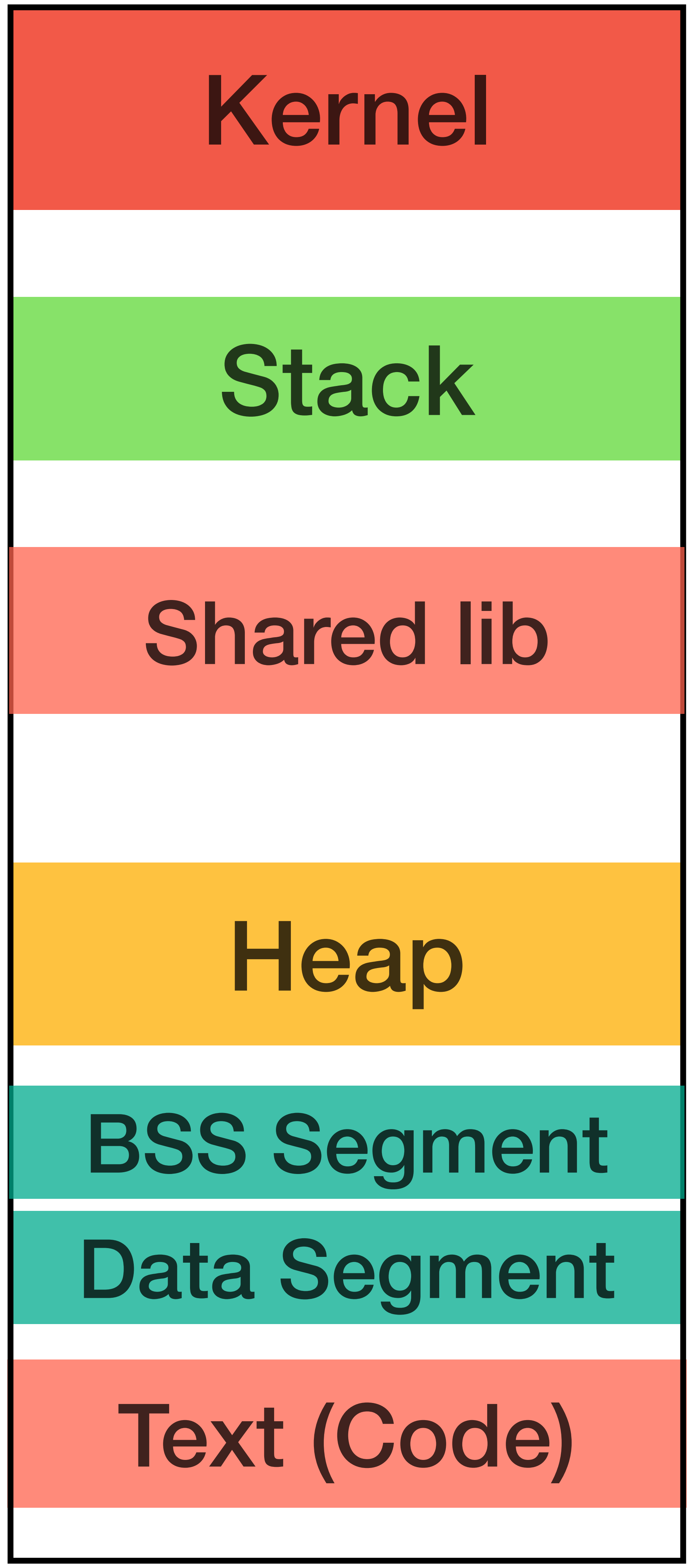
C pointers can do almost arbitrary memory manipulation!

- The correctness is at the discretion of programmers.

Address Space of a C Program

What do programs need in memory?

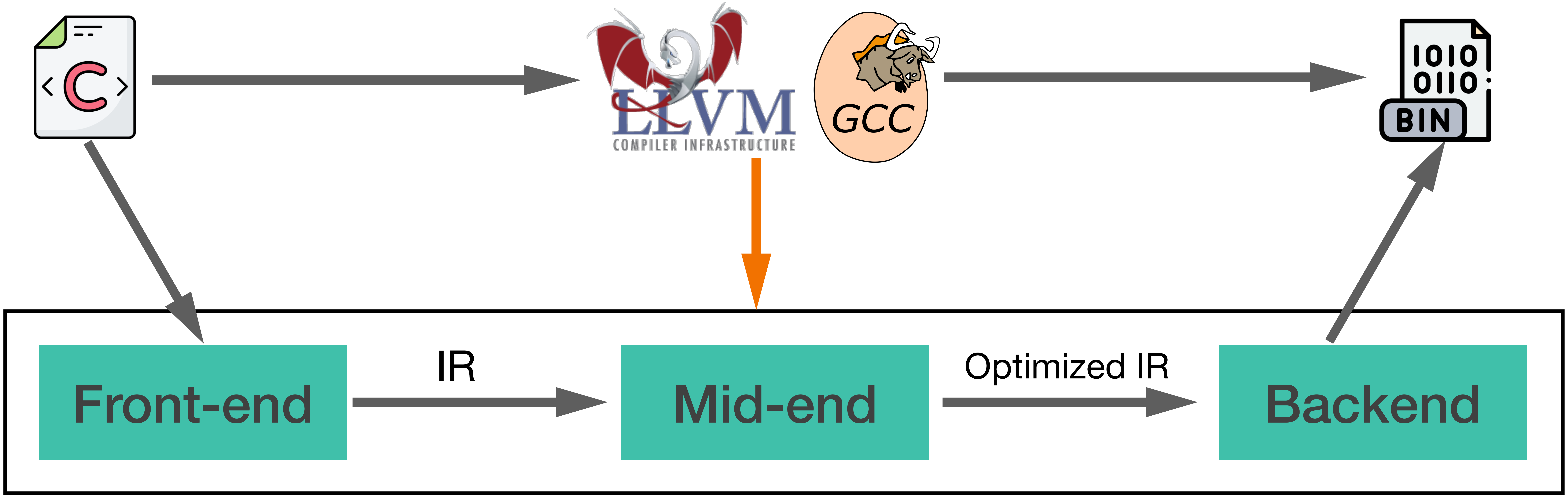
- Code
- Data Segment
  - Initialized data
- BSS Segment
  - Uninitialized data
- Heap
- Shared libraries
- Stack
- Kernel



Future Lectures on Memory Safety Defenses

- Run-time mitigations
  - Address Space Layout Randomization
  - Stack canaries and shadow stacks
  - Control-flow Integrity
  - Memory Isolation
- Testing
  - Memory sanitizing
  - Fuzzing
- Safe implementations
  - Pointer-based memory safety
- Memory-safe languages
  - Safe dialects of C
  - New systems languages

Life of a C Program: Compilation

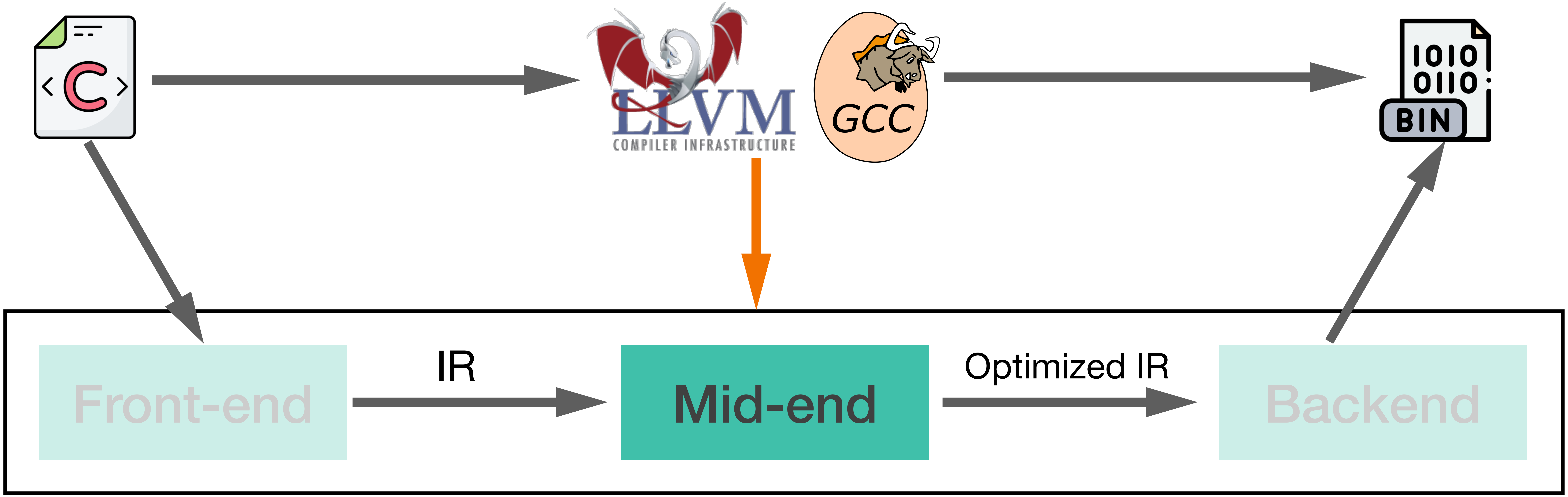


- Parsing
- Semantic Analysis
- Intermediate Representation (IR) Code Generation

- IR Optimizations

- Native CodeGen
- Linking

Compilers Come to the Rescue!



- Parsing
- Semantic Analysis
- Intermediate Representation (IR) Code Generation

- **IR Optimizations**

- Native CodeGen
- Linking

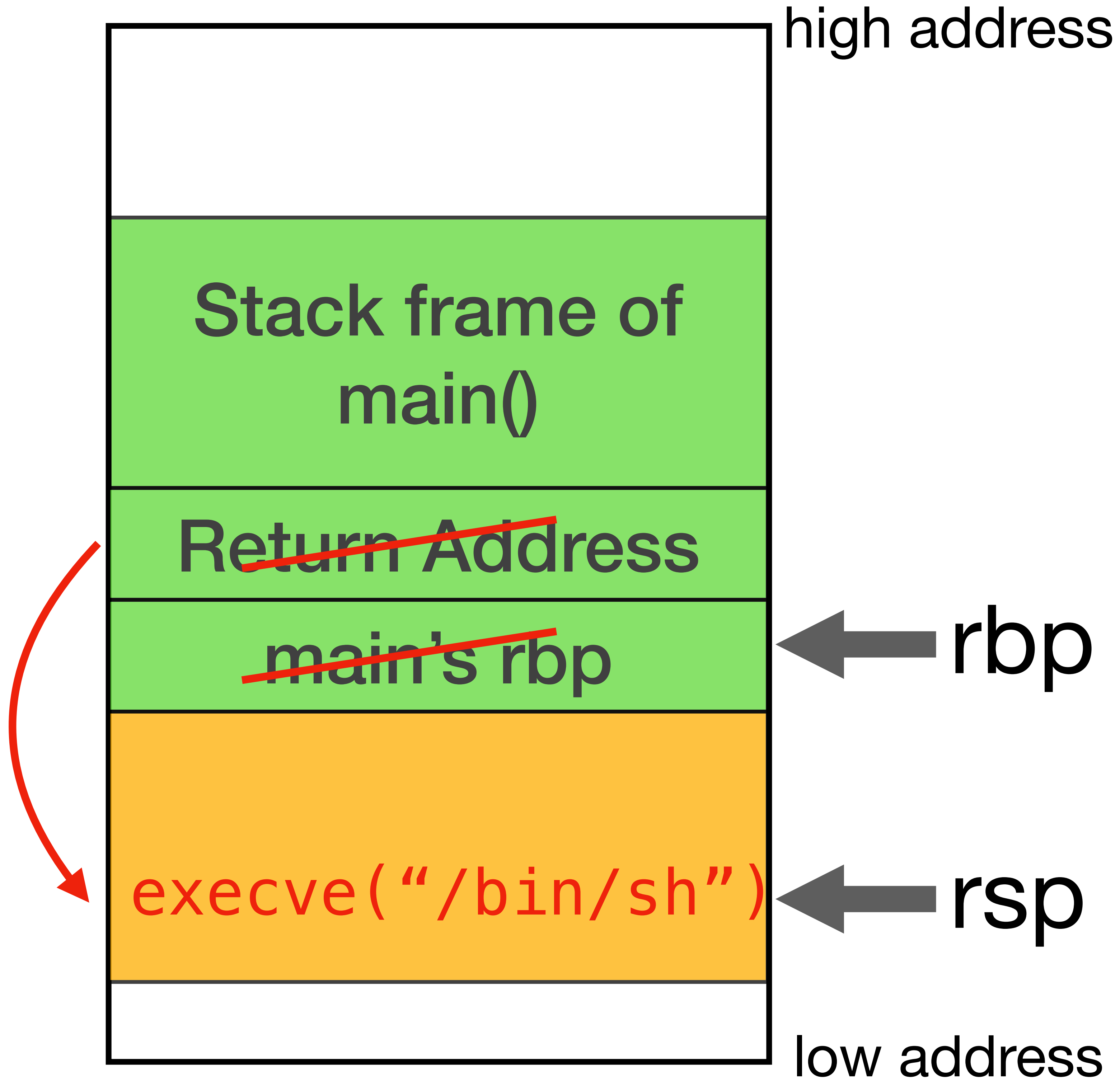
Future Lectures on Memory Safety Defenses

- Run-time mitigations
  - Address Space Layout Randomization
  - **Stack canaries and shadow stacks**
  - **Control-flow Integrity**
  - **Memory Isolation**
- Testing
  - **Memory sanitizing**
  - Fuzzing
- Safe implementations
  - **Pointer-based memory safety**
- Memory-safe languages
  - **Safe dialects of C**
  - **New systems languages**

Q & A

Address Space Layout Randomization

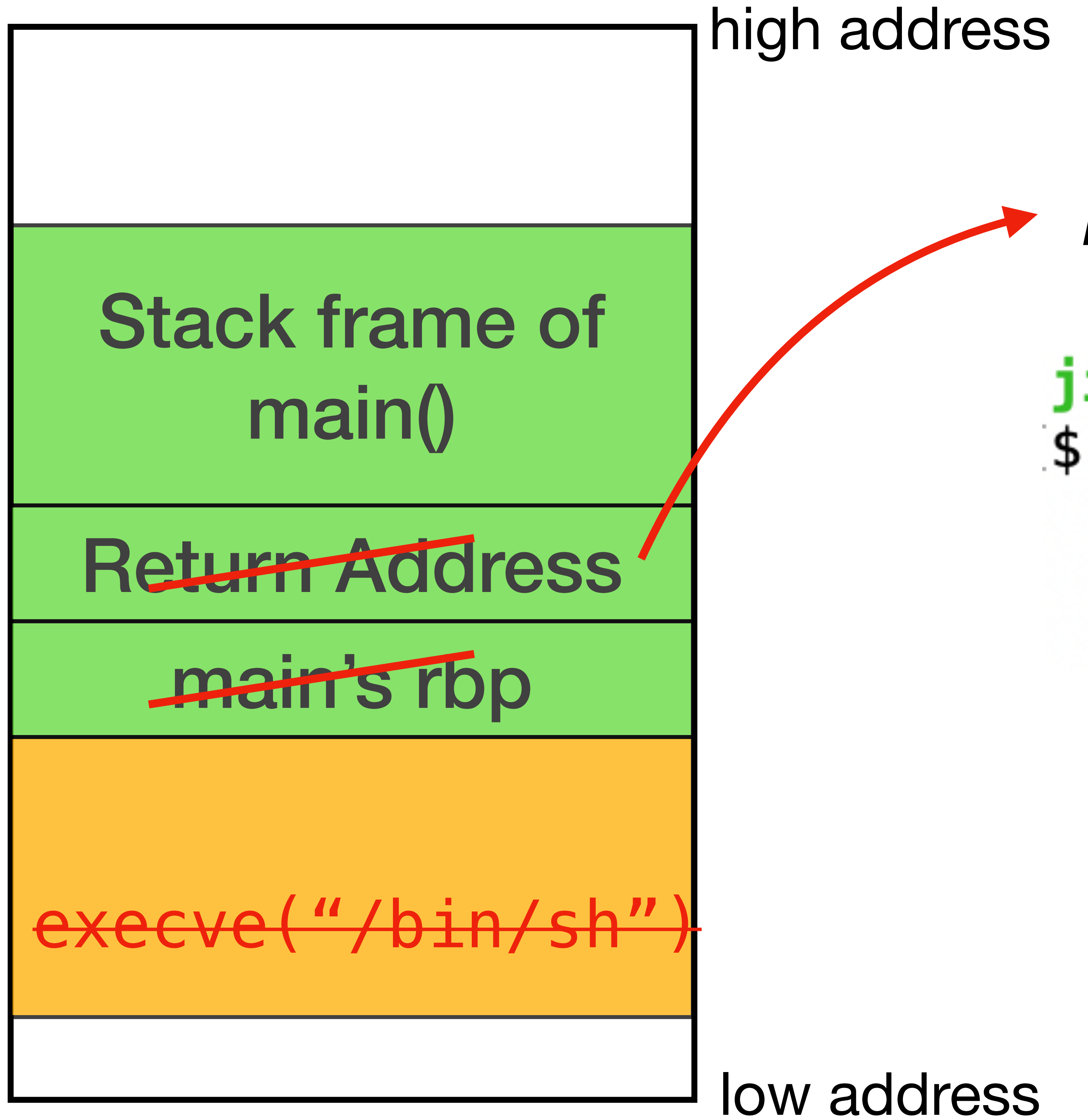
Smashing the Stack: Injecting Shell Code



- This brings up a shell.
- Attackers can execute *any* command in the shell.
- The shell has the same privilege as the process.

- Good news:
  - C/C++ stack is not executable by default.
- Bad news:
  - Code injection works in other cases, e.g. JIT.

Exploiting Existing and Executable Code



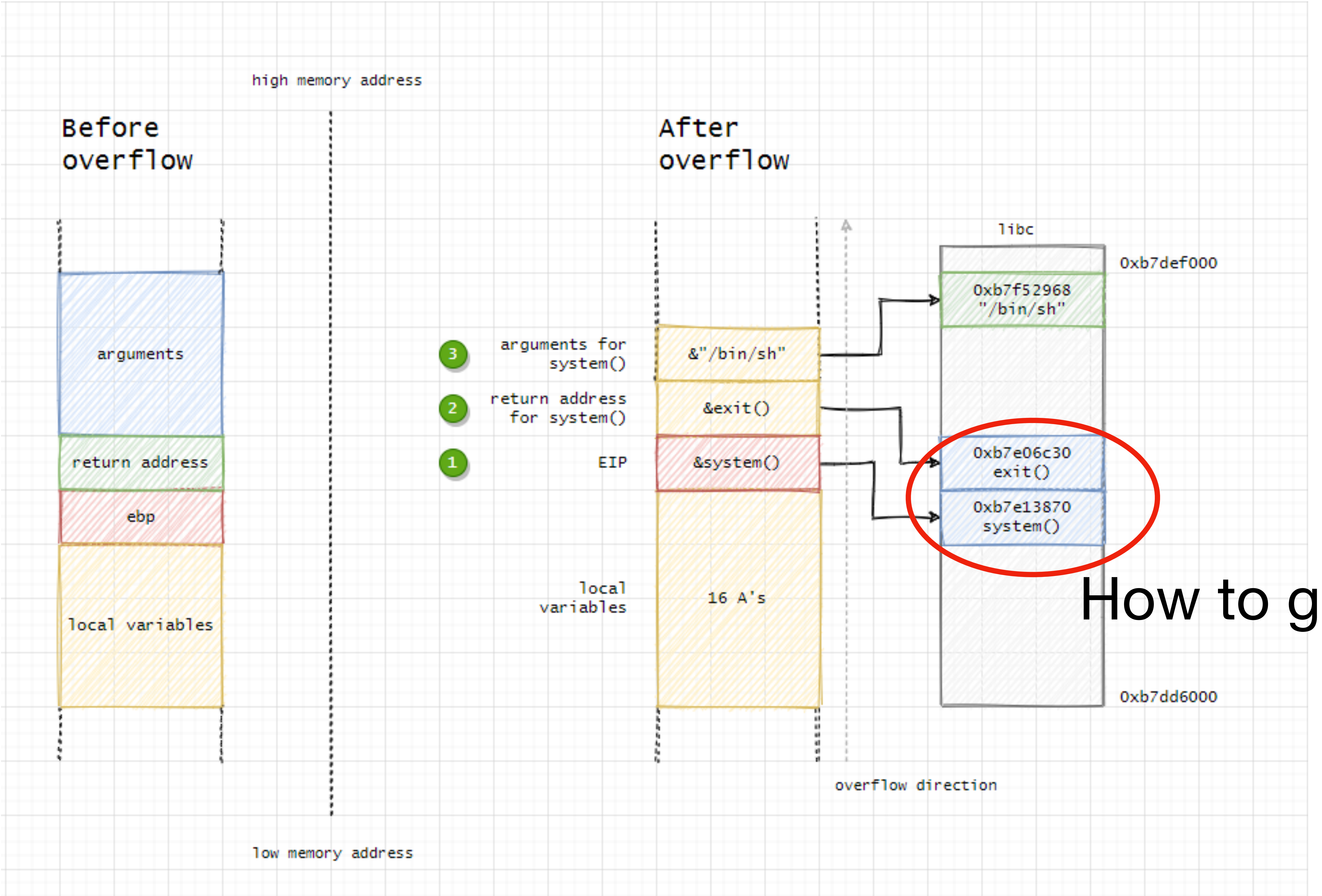
How about "returning" to some library code?

```
jie@gwsyssec: ~/courses/csci6545/lectures
$ ldd demo
        linux-vdso.so.1 (0x00007ffffadfd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f48a2c00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f48a2efc000)
```

```
(gdb) info proc mappings
process 74581
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555555000     0x1000        0x0  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555555000     0x555555556000     0x1000     0x1000  r-xp   /home/jie/courses/csci6545/lectures/demo
      0x555555556000     0x555555557000     0x1000     0x2000  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555557000     0x555555558000     0x1000     0x2000  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555558000     0x555555559000     0x1000     0x3000  rw-p   /home/jie/courses/csci6545/lectures/demo
      0x7ffff7c00000     0x7ffff7c28000    0x28000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7c28000     0x7ffff7dbd000   0x195000    0x28000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7dbd000     0x7ffff7e15000    0x58000   0x1bd000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e15000     0x7ffff7e16000     0x1000   0x215000  ---p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e16000     0x7ffff7e1a000     0x4000   0x215000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1a000     0x7ffff7e1c000     0x2000   0x219000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1c000     0x7ffff7e29000     0xd000        0x0  rw-p
      0x7ffff7fa6000     0x7ffff7fa9000     0x3000        0x0  rw-p
      0x7ffff7fbb000     0x7ffff7fbd000     0x2000        0x0  rw-p
      0x7ffff7fbd000     0x7ffff7fc1000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc1000     0x7ffff7fc3000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fc3000     0x7ffff7fc5000     0x2000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fc5000     0x7ffff7fef000    0x2a000     0x2000  r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fef000     0x7ffff7ffa000     0xb000    0x2c000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x37000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x39000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0  rw-p   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0  --xp   [vsyscall]
```

Exploiting ret2libc on x86-32



How to get these addresses?

Stack memory layout of a 32-bit vulnerable program

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

Life of a C Program: Execution

Loading → **Execution** → **Termination**

Loading

- Initializing memory layout
- (Optional) Dynamic linking, e.g. `libc`
- Environment initialization, e.g., stack setup
- Setting program counter (PC) to `_start()`

Execution

- `_start()` calls `main()`
- `main() runs the program`

Termination

- `main()` returns,
- `_start()` calls `exit()`
- cleanup and shutdown

How to Get Target Addresses?

- Examining the binary at run-time
  - Debugger (GDB/LLDB/etc.)
  - Systems convention
    - On x86-64/Linux, `main()` usually starts around `0x400000`
- In Assignment 1, the program was compiled by

```
clang lucky.c -fno-pic -no-pie -o lucky
```

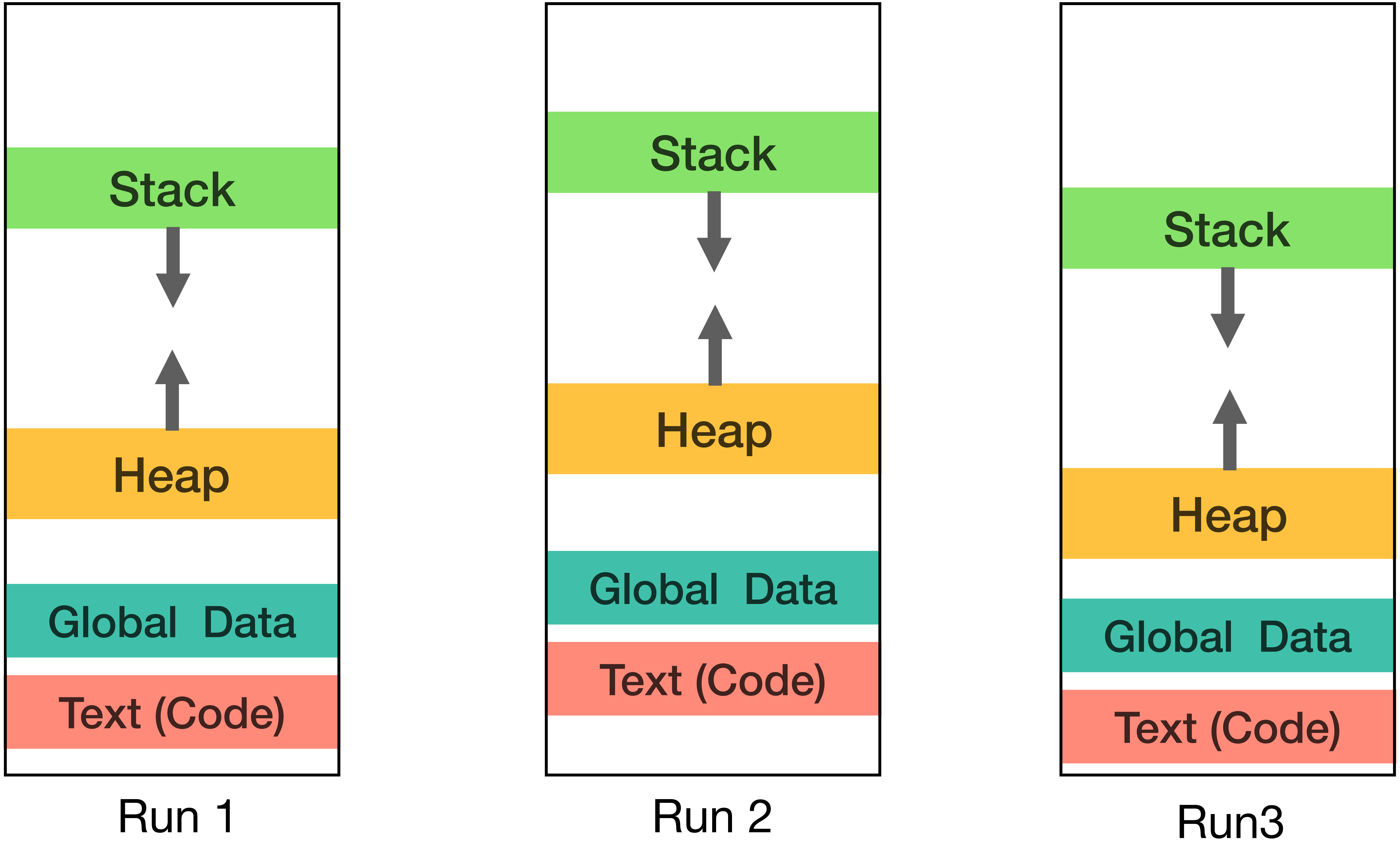
  - `pic`: position-independent code (usually for shared libraries)
  - `pie`: position-independent executable (for executables)
  - These options determines whether the code addresses of `lucky` executable are fixed or randomized during loading.

Address Space Layout Randomization (ASLR)

Introducing randomness into memory regions of a program

- During program initialization, done by the program loader
- Can also happen during static linking time
- Making it hard to figure out attacked target addresses

Address Space Layout Randomization (ASLR)



Address Space Layout Randomization (ASLR)

- **When** to randomize address space?
  - Only at loading time or also at run-time?
  - What should the randomization frequency be?
- **What** to randomize?
  - Which memory regions to randomize?
  - Should we randomize each memory objects?
- **How** to randomize?

Memory Mapping of vim

```
$ cat /proc/147967/map
map_files/ maps
jie@fedora: /home/jie
$ cat /proc/147967/maps
564b3aef8000-564b3aefd000 r--p 00000000 00:20 160559                     /usr/bin/vim
564b3aefd000-564b3b235000 r-xp 00005000 00:20 160559                     /usr/bin/vim
564b3b235000-564b3b2a1000 r--p 0033d000 00:20 160559                     /usr/bin/vim
564b3b2a1000-564b3b2b5000 r--p 003a8000 00:20 160559                     /usr/bin/vim
564b3b2b5000-564b3b2e9000 rw-p 003bc000 00:20 160559                     /usr/bin/vim
564b3b2e9000-564b3b2f8000 rw-p 00000000 00:00 0
564b3b349000-564b3b75f000 rw-p 00000000 00:00 0                          [heap]
7fe39c600000-7fe3aa11d000 r--p 00000000 00:20 21612                      /usr/lib/locale/locale-archive
7fe3aa12a000-7fe3aa12e000 rw-p 00000000 00:00 0
7fe3aa12e000-7fe3aa130000 r--p 00000000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa130000-7fe3aa133000 r-xp 00002000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa133000-7fe3aa134000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa134000-7fe3aa135000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa135000-7fe3aa136000 rw-p 00000000 00:00 0
7fe3aa136000-7fe3aa138000 r--p 00000000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa138000-7fe3aa1a8000 r-xp 00002000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1a8000-7fe3aa1d0000 r--p 00072000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d0000-7fe3aa1d1000 r--p 00099000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d1000-7fe3aa1d2000 rw-p 0009a000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d2000-7fe3aa1fa000 r--p 00000000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa1fa000-7fe3aa363000 r-xp 00028000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa363000-7fe3aa3b1000 r--p 00191000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b1000-7fe3aa3b5000 r--p 001de000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b5000-7fe3aa3b7000 rw-p 001e2000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b7000-7fe3aa3c1000 rw-p 00000000 00:00 0
7fe3aa3c1000-7fe3aa3c3000 r--p 00000000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c3000-7fe3aa3c6000 r-xp 00002000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c6000-7fe3aa3c7000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c7000-7fe3aa3c8000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c8000-7fe3aa3c9000 rw-p 00006000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c9000-7fe3aa3cb000 r--p 00000000 00:20 37395                      /usr/lib64/libacl.so.1.1.2302
7fe3aa3cb000-7fe3aa3d0000 r-xp 00002000 00:20 37395                      /usr/lib64/libacl.so.1.1.2302
7fe3aa3d0000-7fe3aa3d1000 r--p 00007000 00:20 37395                      /usr/lib64/libacl.so.1.1.2302
7fe3aa3d1000-7fe3aa3d2000 r--p 00008000 00:20 37395                      /usr/lib64/libacl.so.1.1.2302
7fe3aa3d2000-7fe3aa3d3000 rw-p 00000000 00:00 0
7fe3aa3d3000-7fe3aa3e0000 r--p 00000000 00:20 38583                      /usr/lib64/libsodium.so.26.1.0
7fe3aa3e0000-7fe3aa41f000 r-xp 0000d000 00:20 38583                      /usr/lib64/libsodium.so.26.1.0
7fe3aa41f000-7fe3aa430000 r--p 0004c000 00:20 38583                      /usr/lib64/libsodium.so.26.1.0
7fe3aa430000-7fe3aa431000 r--p 0005d000 00:20 38583                      /usr/lib64/libsodium.so.26.1.0
7fe3aa431000-7fe3aa432000 rw-p 0005e000 00:20 38583                      /usr/lib64/libsodium.so.26.1.0
7fe3aa432000-7fe3aa438000 r--p 00000000 00:20 38657                      /usr/lib64/libtinfo.so.6.4
7fe3aa438000-7fe3aa44c000 r-xp 00006000 00:20 38657                      /usr/lib64/libtinfo.so.6.4
7fe3aa44c000-7fe3aa45a000 r--p 0001a000 00:20 38657                      /usr/lib64/libtinfo.so.6.4
7fe3aa45a000-7fe3aa45e000 r--p 00027000 00:20 38657                      /usr/lib64/libtinfo.so.6.4
7fe3aa45e000-7fe3aa45f000 rw-p 0002b000 00:20 38657                      /usr/lib64/libtinfo.so.6.4
7fe3aa45f000-7fe3aa465000 r--p 00000000 00:20 38547                      /usr/lib64/libselinux.so.1
7fe3aa465000-7fe3aa481000 r-xp 00006000 00:20 38547                      /usr/lib64/libselinux.so.1
7fe3aa481000-7fe3aa488000 r--p 00022000 00:20 38547                      /usr/lib64/libselinux.so.1
7fe3aa488000-7fe3aa489000 r--p 00028000 00:20 38547                      /usr/lib64/libselinux.so.1
7fe3aa489000-7fe3aa48a000 rw-p 00029000 00:20 38547                      /usr/lib64/libselinux.so.1
7fe3aa48a000-7fe3aa48c000 rw-p 00000000 00:00 0
7fe3aa48c000-7fe3aa49c000 r--p 00000000 00:20 38203                      /usr/lib64/libm.so.6
7fe3aa49c000-7fe3aa513000 r-xp 00010000 00:20 38203                      /usr/lib64/libm.so.6
7fe3aa513000-7fe3aa56d000 r--p 00087000 00:20 38203                      /usr/lib64/libm.so.6
7fe3aa56d000-7fe3aa56e000 r--p 000e0000 00:20 38203                      /usr/lib64/libm.so.6
7fe3aa56e000-7fe3aa56f000 rw-p 000e1000 00:20 38203                      /usr/lib64/libm.so.6
7fe3aa57a000-7fe3aa57b000 ---p 00000000 00:00 0
7fe3aa57b000-7fe3aa582000 rw-p 00000000 00:00 0
7fe3aa582000-7fe3aa584000 rw-p 00000000 00:00 0
7fe3aa584000-7fe3aa585000 r--p 00000000 00:20 37086                      /usr/lib64/ld-linux-x86-64.so.2
7fe3aa585000-7fe3aa5ac000 r-xp 00001000 00:20 37086                      /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5ac000-7fe3aa5b6000 r--p 00028000 00:20 37086                      /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5b6000-7fe3aa5b8000 r--p 00032000 00:20 37086                      /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5b8000-7fe3aa5ba000 rw-p 00034000 00:20 37086                      /usr/lib64/ld-linux-x86-64.so.2
7ffdc775f000-7ffdc7780000 rw-p 00000000 00:00 0                          [stack]
7ffdc77ce000-7ffdc77d2000 r--p 00000000 00:00 0                          [vvar]
7ffdc77d2000-7ffdc77d4000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Memory Mapping of vim

```
$ cat /proc/147967/map
map_files/ maps
jie@fedora: /home/jie
$ cat /proc/147967/maps
564b3aef8000-564b3aefd000 r--p 00000000 00:20 160559                     /usr/bin/vim
564b3aefd000-564b3b235000 r-xp 00005000 00:20 160559                     /usr/bin/vim
564b3b235000-564b3b2a1000 r--p 0033d000 00:20 160559                     /usr/bin/vim
564b3b2a1000-564b3b2b5000 r--p 003a8000 00:20 160559                     /usr/bin/vim
564b3b2b5000-564b3b2e9000 rw-p 003bc000 00:20 160559                     /usr/bin/vim
564b3b2e9000-564b3b2f8000 rw-p 00000000 00:00 0
564b3b349000-564b3b75f000 rw-p 00000000 00:00 0                          [heap]
7fe39c600000-7fe3aa11d000 r--p 00000000 00:20 21612                      /usr/lib/locale/locale-archive
7fe3aa12a000-7fe3aa12e000 rw-p 00000000 00:00 0
7fe3aa12e000-7fe3aa130000 r--p 00000000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa130000-7fe3aa133000 r-xp 00002000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa133000-7fe3aa134000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa134000-7fe3aa135000 r--p 00005000 00:20 37425                      /usr/lib64/libattr.so.1.1.2502
7fe3aa135000-7fe3aa136000 rw-p 00000000 00:00 0
7fe3aa136000-7fe3aa138000 r--p 00000000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa138000-7fe3aa1a8000 r-xp 00002000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1a8000-7fe3aa1d0000 r--p 00072000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d0000-7fe3aa1d1000 r--p 00099000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d1000-7fe3aa1d2000 rw-p 0009a000 00:20 38428                      /usr/lib64/libpcre2-8.so.0.11.2
7fe3aa1d2000-7fe3aa1fa000 r--p 00000000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa1fa000-7fe3aa363000 r-xp 00028000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa363000-7fe3aa3b1000 r--p 00191000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b1000-7fe3aa3b5000 r--p 001de000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b5000-7fe3aa3b7000 rw-p 001e2000 00:20 37490                      /usr/lib64/libc.so.6
7fe3aa3b7000-7fe3aa3c1000 rw-p 00000000 00:00 0
7fe3aa3c1000-7fe3aa3c3000 r--p 00000000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c3000-7fe3aa3c6000 r-xp 00002000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c6000-7fe3aa3c7000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c7000-7fe3aa3c8000 r--p 00005000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
7fe3aa3c8000-7fe3aa3c9000 rw-p 00006000 00:20 160557                     /usr/lib64/libgpm.so.2.1.0
```

binary

shared libs

Memory Mapping of vim

```
7fe3aa3c7000-7fe3aa3c8000 r--p 00005000 00:20 160557                    /usr/lib64/libgpm.so.2.1.0
7fe3aa3c8000-7fe3aa3c9000 rw-p 00006000 00:20 160557                    /usr/lib64/libgpm.so.2.1.0
7fe3aa3c9000-7fe3aa3cb000 r--p 00000000 00:20 37395                     /usr/lib64/libacl.so.1.1.2302
7fe3aa3cb000-7fe3aa3d0000 r-xp 00002000 00:20 37395                     /usr/lib64/libacl.so.1.1.2302
7fe3aa3d0000-7fe3aa3d1000 r--p 00007000 00:20 37395                     /usr/lib64/libacl.so.1.1.2302
7fe3aa3d1000-7fe3aa3d2000 r--p 00008000 00:20 37395                     /usr/lib64/libacl.so.1.1.2302
7fe3aa3d2000-7fe3aa3d3000 rw-p 00000000 00:00 0
7fe3aa3d3000-7fe3aa3e0000 r--p 00000000 00:20 38583                     /usr/lib64/libsodium.so.26.1.0
7fe3aa3e0000-7fe3aa41f000 r-xp 0000d000 00:20 38583                     /usr/lib64/libsodium.so.26.1.0
7fe3aa41f000-7fe3aa430000 r--p 0004c000 00:20 38583                     /usr/lib64/libsodium.so.26.1.0
7fe3aa430000-7fe3aa431000 r--p 0005d000 00:20 38583                     /usr/lib64/libsodium.so.26.1.0
7fe3aa431000-7fe3aa432000 rw-p 0005e000 00:20 38583                     /usr/lib64/libsodium.so.26.1.0
7fe3aa432000-7fe3aa438000 r--p 00000000 00:20 38657                     /usr/lib64/libtinfo.so.6.4
7fe3aa438000-7fe3aa44c000 r-xp 00006000 00:20 38657                     /usr/lib64/libtinfo.so.6.4
7fe3aa44c000-7fe3aa45a000 r--p 0001a000 00:20 38657                     /usr/lib64/libtinfo.so.6.4
7fe3aa45a000-7fe3aa45e000 r--p 00027000 00:20 38657                     /usr/lib64/libtinfo.so.6.4
7fe3aa45e000-7fe3aa45f000 rw-p 0002b000 00:20 38657                     /usr/lib64/libtinfo.so.6.4
7fe3aa45f000-7fe3aa465000 r--p 00000000 00:20 38547                     /usr/lib64/libselinux.so.1
7fe3aa465000-7fe3aa481000 r-xp 00006000 00:20 38547                     /usr/lib64/libselinux.so.1
7fe3aa481000-7fe3aa488000 r--p 00022000 00:20 38547                     /usr/lib64/libselinux.so.1
7fe3aa488000-7fe3aa489000 r--p 00028000 00:20 38547                     /usr/lib64/libselinux.so.1
7fe3aa489000-7fe3aa48a000 rw-p 00029000 00:20 38547                     /usr/lib64/libselinux.so.1
7fe3aa48a000-7fe3aa48c000 rw-p 00000000 00:00 0
7fe3aa48c000-7fe3aa49c000 r--p 00000000 00:20 38203                     /usr/lib64/libm.so.6
7fe3aa49c000-7fe3aa513000 r-xp 00010000 00:20 38203                     /usr/lib64/libm.so.6
7fe3aa513000-7fe3aa56d000 r--p 00087000 00:20 38203                     /usr/lib64/libm.so.6
7fe3aa56d000-7fe3aa56e000 r--p 000e0000 00:20 38203                     /usr/lib64/libm.so.6
7fe3aa56e000-7fe3aa56f000 rw-p 000e1000 00:20 38203                     /usr/lib64/libm.so.6
7fe3aa57a000-7fe3aa57b000 ---p 00000000 00:00 0
7fe3aa57b000-7fe3aa582000 rw-p 00000000 00:00 0
7fe3aa582000-7fe3aa584000 rw-p 00000000 00:00 0
7fe3aa584000-7fe3aa585000 r--p 00000000 00:20 37086                     /usr/lib64/ld-linux-x86-64.so.2
7fe3aa585000-7fe3aa5ac000 r-xp 00001000 00:20 37086                     /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5ac000-7fe3aa5b6000 r--p 00028000 00:20 37086                     /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5b6000-7fe3aa5b8000 r--p 00032000 00:20 37086                     /usr/lib64/ld-linux-x86-64.so.2
7fe3aa5b8000-7fe3aa5ba000 rw-p 00034000 00:20 37086                     /usr/lib64/ld-linux-x86-64.so.2
7ffdc775f000-7ffdc7780000 rw-p 00000000 00:00 0                         [stack]
7ffdc77ce000-7ffdc77d2000 r--p 00000000 00:00 0                         [vvar]
7ffdc77d2000-7ffdc77d4000 r-xp 00000000 00:00 0                         [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                 [vsyscall]
```

runtime linker / loader

kernel-provided

Address Space Layout Randomization (ASLR)

- **When** to randomize address space?
  - Only at loading time or also at run-time?
  - What should the randomization frequency be?
- **What** to randomize?
  - Which memory regions to randomize?
  - Should we randomize each memory objects?
- **How** to randomize?
  - How many bits to randomize?

Case Study: PaX's ASLR on x86-32 Systems

- **When** to randomize address space?
  - At loading time
- **What** to randomize?
  - Stack
  - `mmap()` area (shared libs + partial heap)
  - Main executable
- **How** to randomize?

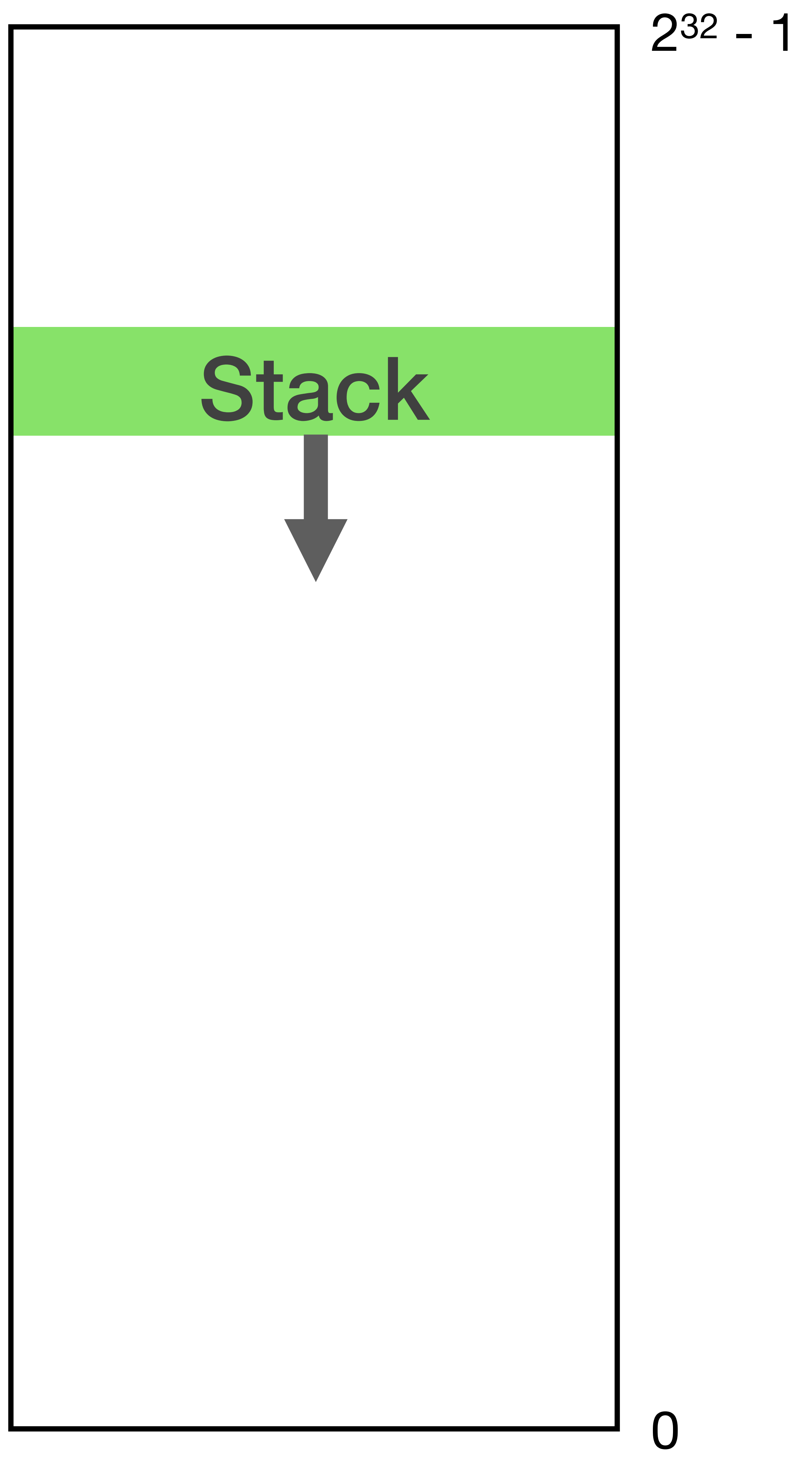
Case Study: PaX's ASLR on x86-32 Systems

- Randomized bits: number of bits ASLR can vary for a memory region
- Attacked bits: number of bits attackers can bypass (e.g., partial info leak)

- Rs: number of randomized bits for the stack
- Rm: number of randomized bits for the `mmap()` area
- Rx: number of randomized bits for the main executable
- As: number of bits of stack randomness attacked in one attempt
- Am: number of bits of `mmap()` randomness attacked in one attempt
- Ax: number of bits of main executable randomness attacked in on attempt
- Probability of success within x number of attempts:
  - Brute-force attacks: $Pb(x) = x / 2^n$
  - Random guess attacks: $Pb(x) = 1 - (1 - 2^{-n})^x$

where $n = Rs-As + Rm-Am + Rx-Ax$, i.e., the number of randomized bits to find.

Probability of Success

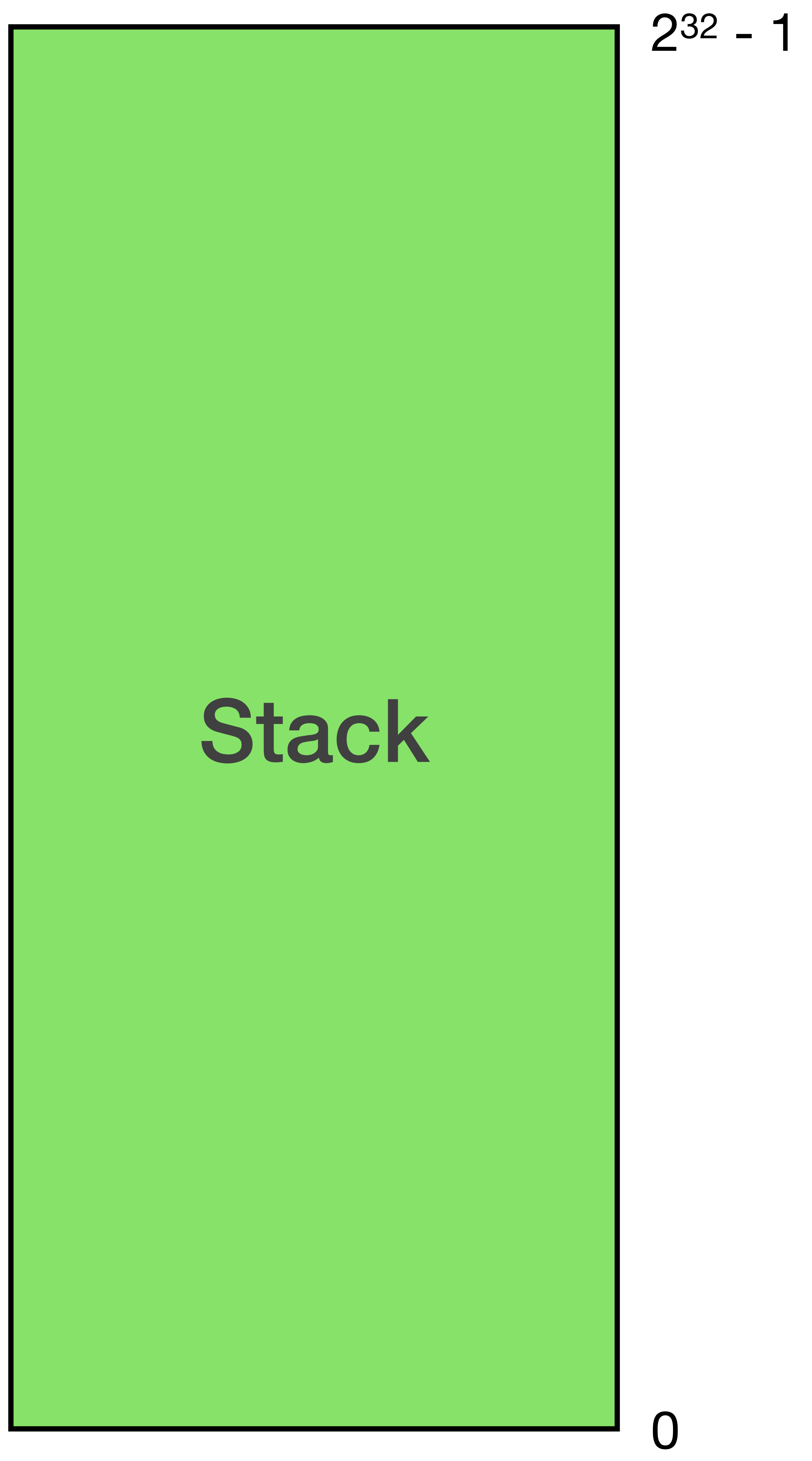


- Assume the stack
  - 1 byte large
  - Can stay anywhere in the address space
- How many randomized bits do we have?

  32
- What's the probability of success with one guess?

  $1 / 2^{32}$

Probability of Success

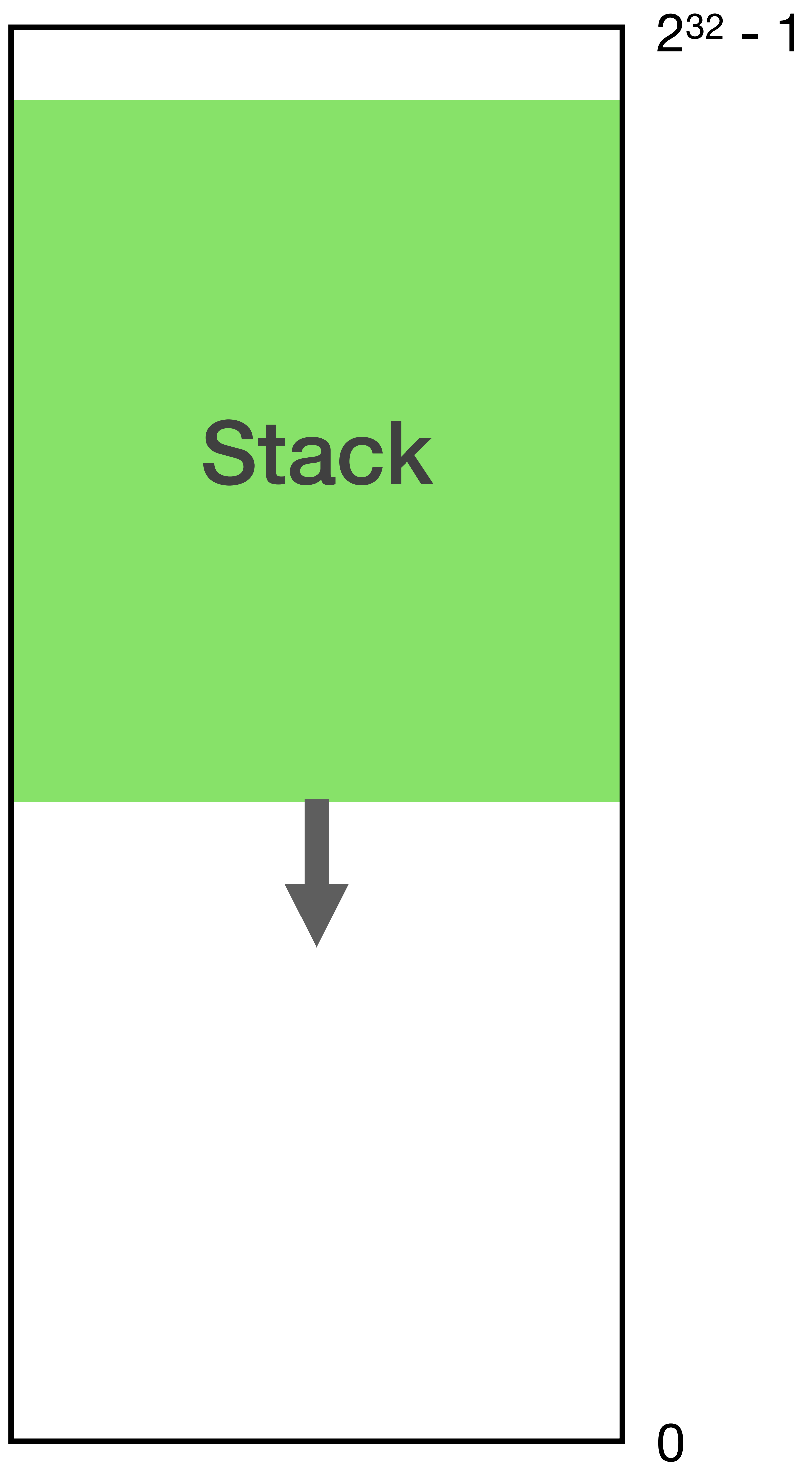


- Assume the stack
  - 2^{32} bytes (4 GB) large
  - Can stay anywhere in the address space
- How many randomized bits do we have?

  0
- What's the probability of success with one guess?

  100%

Probability of Success

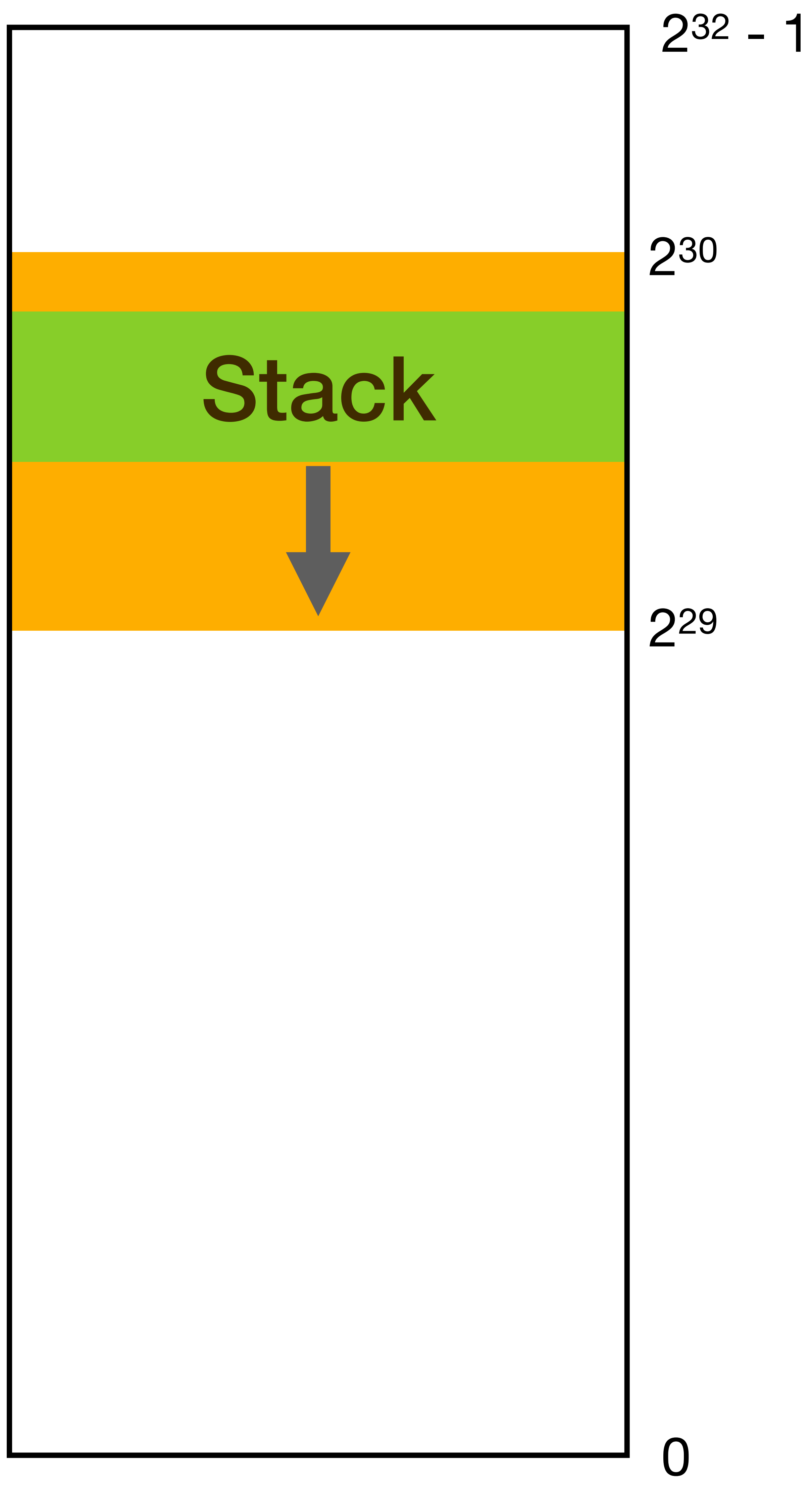


- Assume the stack
  - 2^{31} bytes (2G) large
  - Can stay anywhere in the address space
- How many randomized bits do we have?

  31
- What's the probability of success with one guess?

  $1 / 2^{31}$

Probability of Success



- Assume the stack
  - 2^{21} bytes (2 MB) large
  - Restricted to address 2^{29} to 2^{30} (512 MB)
- How many randomized bits do we have?

  2^{29} to $(2^{30} - 2^{21})$ ⟶ ~ 29
- What's the probability of success with on guess?

  $1 / 2^{29}$

  Considering alignment requirements, we most likely will only have 25 randomized bits, assuming a 2^4 = 16-bytes alignment.

Case Study: PaX's ASLR on x86-32 Systems

- Randomized bits: number of bits ASLR can vary for a memory region
- Attacked bits: number of bits attackers can bypass (e.g., partial info leak)

- Rs: number of randomized bits for the stack
- Rm: number of randomized bits for the mmap() area
- Rx: number of randomized bits for the main executable
- As: number of bits of stack randomness attacked in one attempt
- Am: number of bits of `mmap()` randomness attacked in one attempt
- Ax: number of bits of main executable randomness attacked in on attempt
- Probability of success within x number of attempts:
  - Brute-force attacks: $Pb(x) = x / 2^n$
  - Random guss attacks: $Pb(x) = 1 - (1 - 2^{-n})^x$

where $n = Rs - As + Rm - Am + Rx - Ax$, i.e., the number of randomized bits to find.

How Effective/Robust is ASLR?

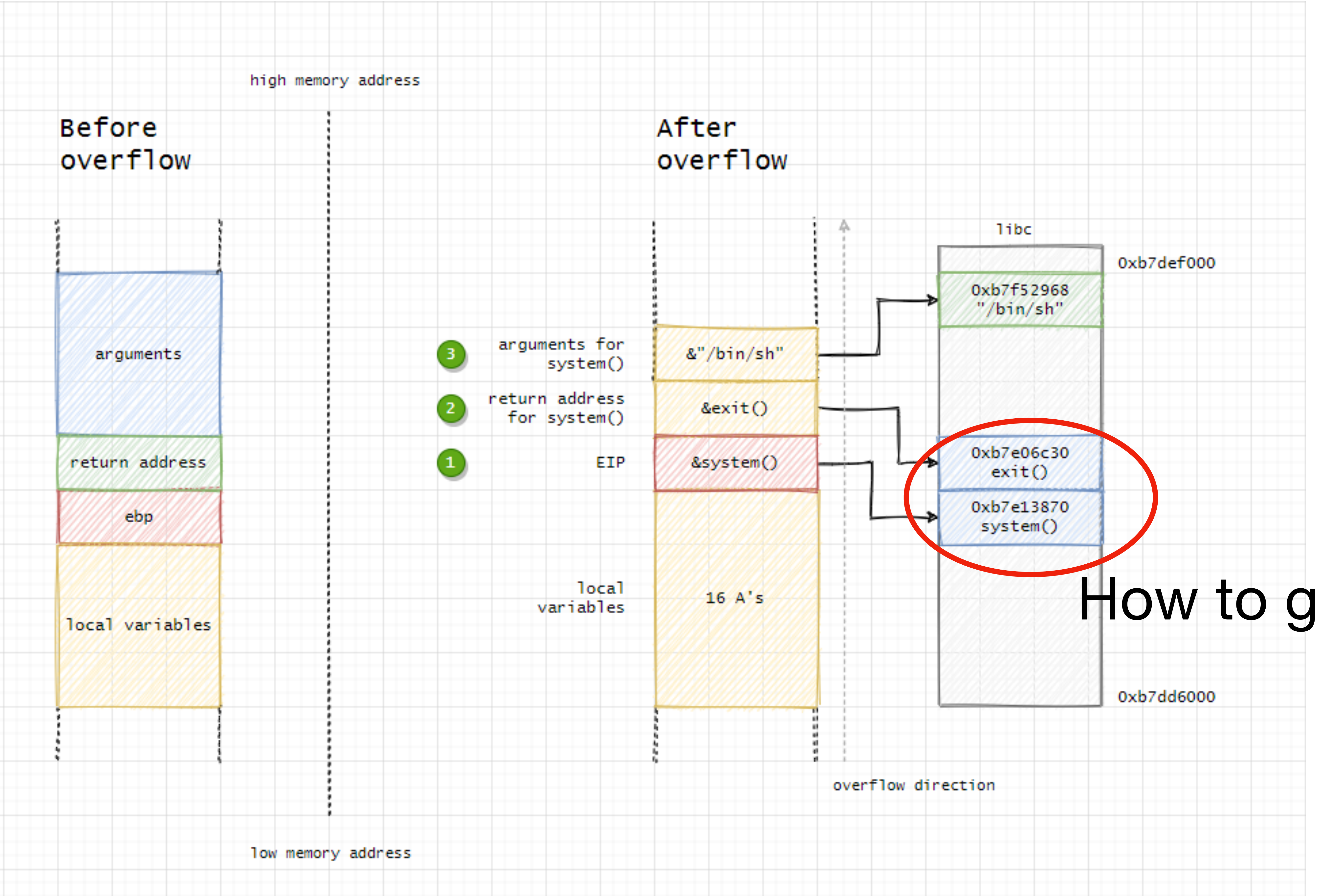
Attacking PaX ASLR

Attack the Apache http server with `ret2libc`

- Server takes requests for connections from remote users
  - Creating a new child process to handle the request
- A stack buffer overflow bug was exploited when handling user input
  - The vulnerable buffer's address is stored as a local variable on stack
- ASLR setting:
  - Starting address of each memory region is randomized
  - Randomized bits: 16 bits for `mmap()` and code, and 24 bits for stack
    - Kernel maintains a `delta_mmap` variable as the offset to the start address of the `mmap()` region, which is `0x40000000`.
- Attacking goal: Invoke `system()` with argument to launch a shell

Shacham, Hovav, et al. "On the effectiveness of address-space randomization." ACM conference on Computer and Communications Security. 2004.

Exploiting ret2libc on x86-32



How to get these addresses?

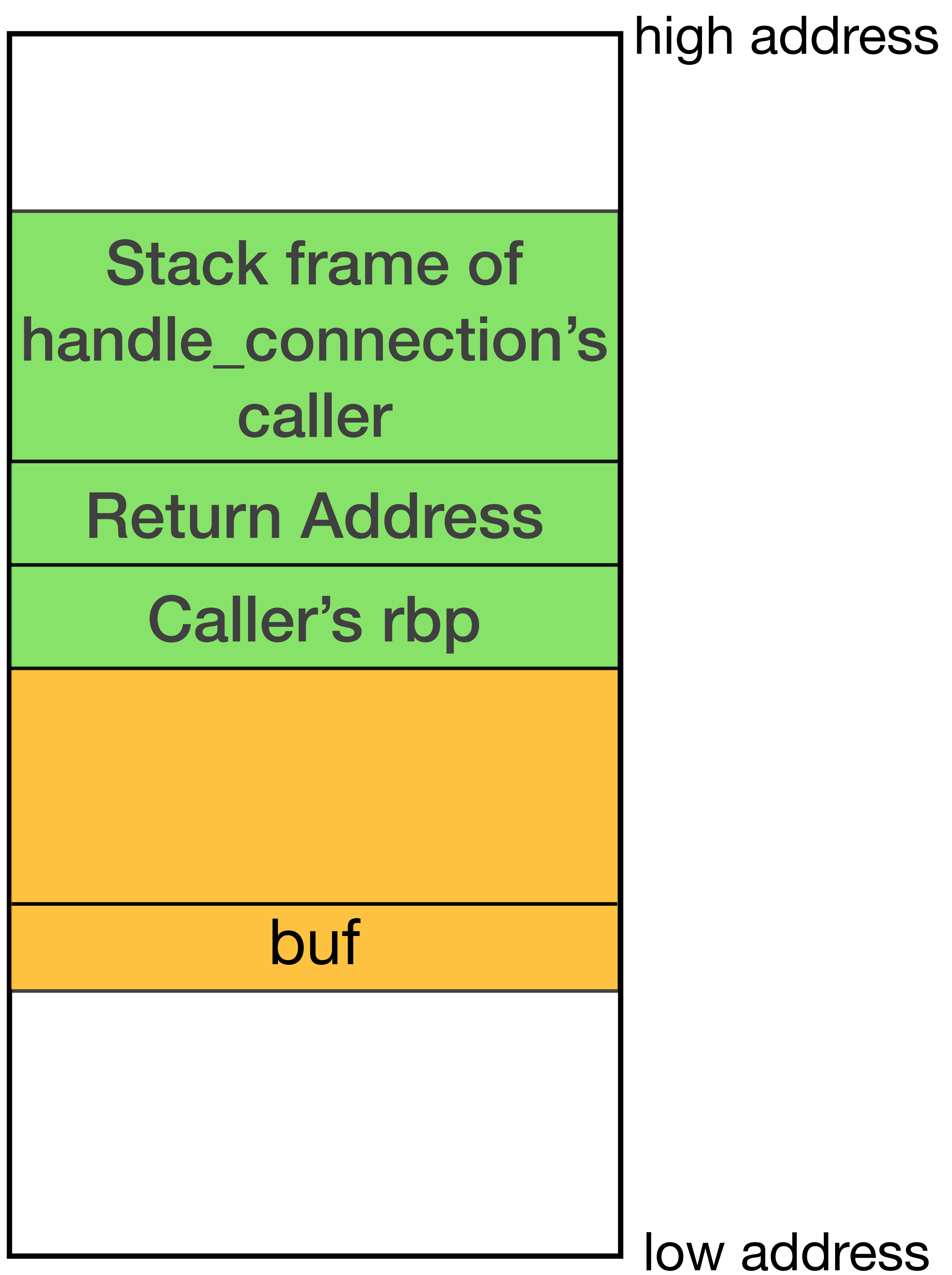
Stack memory layout of a 32-bit vulnerable program

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

Attacking PaX ASLR

```
void handle_connection(...) {
    char buf[64];
    ...
    strcpy(buf, s);  // Buffer overflow
    ...
}
```

- Attacking goal:
  - Figure out `system()`'s address
  - Supply the argument and its address to `system()`

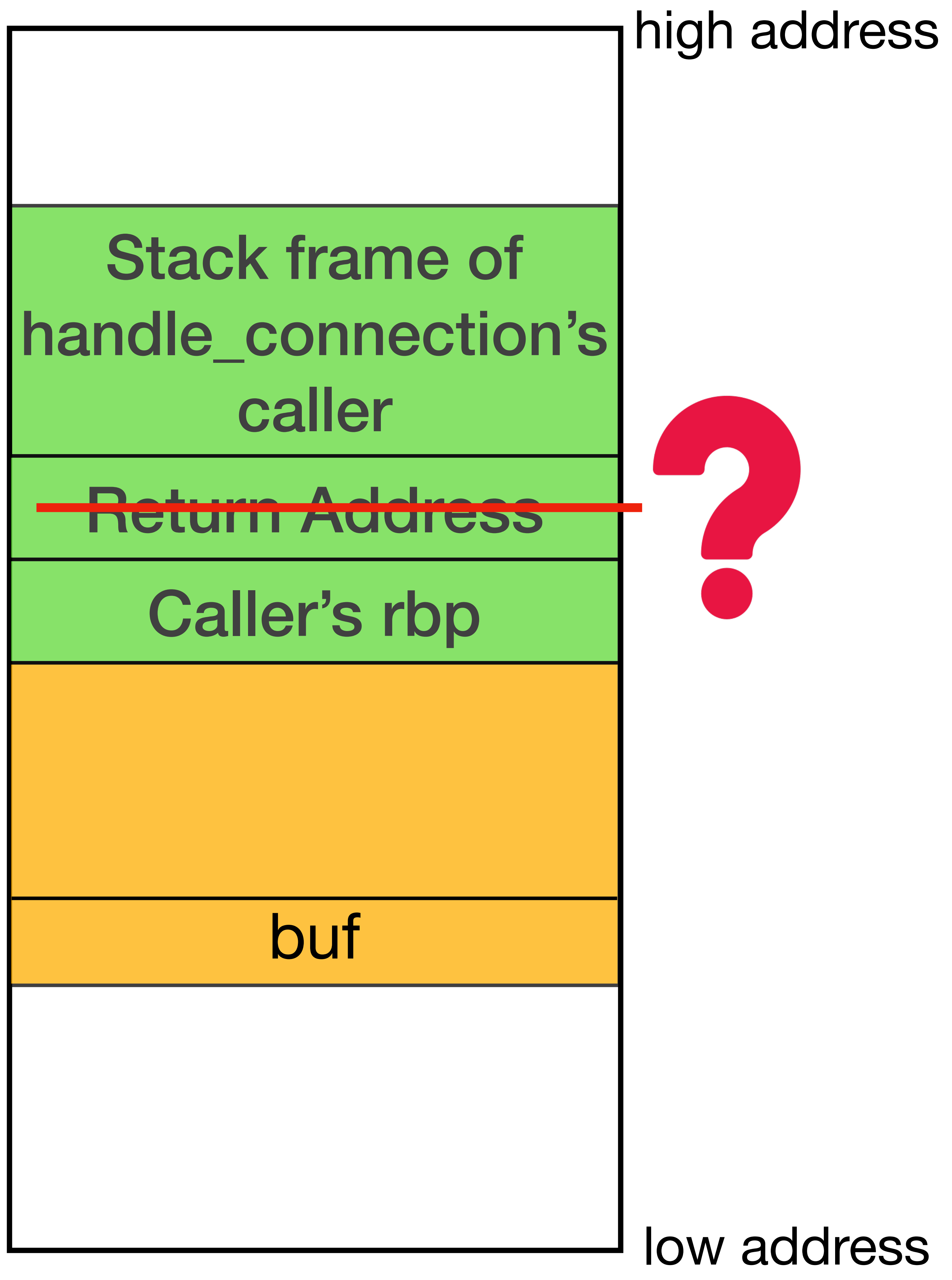


Attacking PaX ASLR

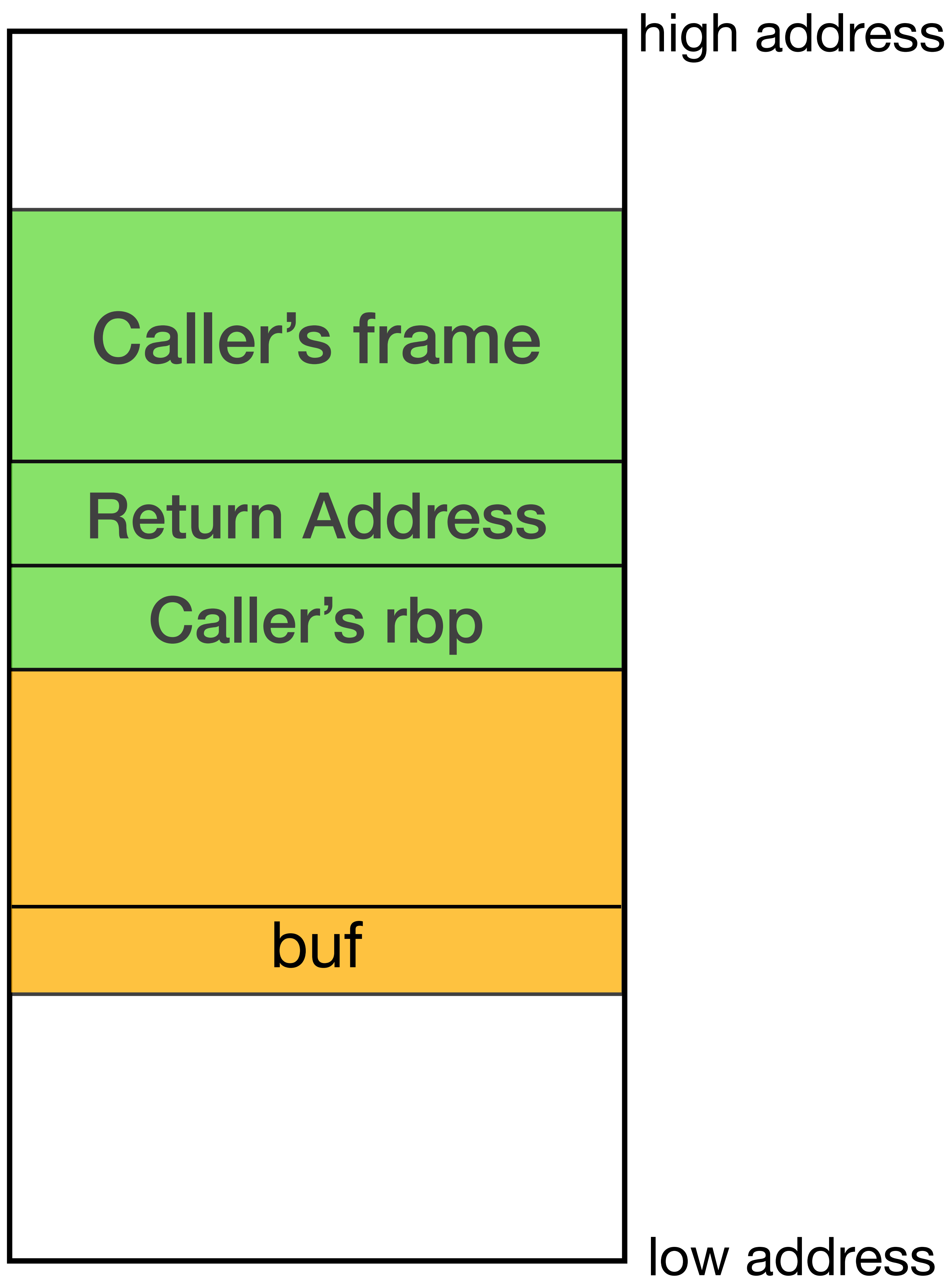
Address = `0x40000000` + delta_mmap + offset_in_lib

Attacking steps: Brute-force guessing `usleep()`'s address with argument 16 seconds.

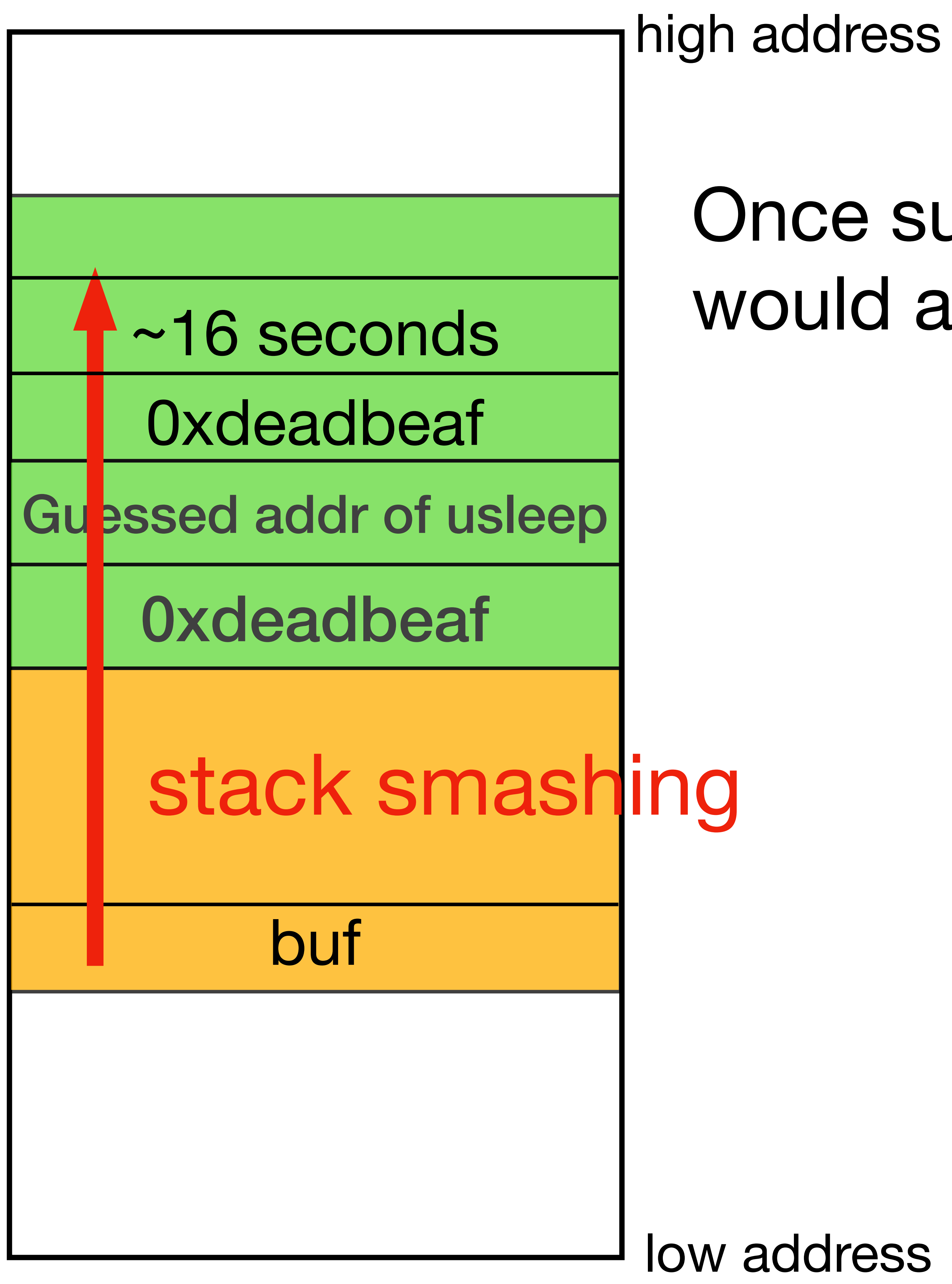
- If succeeded, server will hang for 16s.
- If failed, connection will terminate instantly.



Exploit Step 1: Figure Out `delta_mmap`



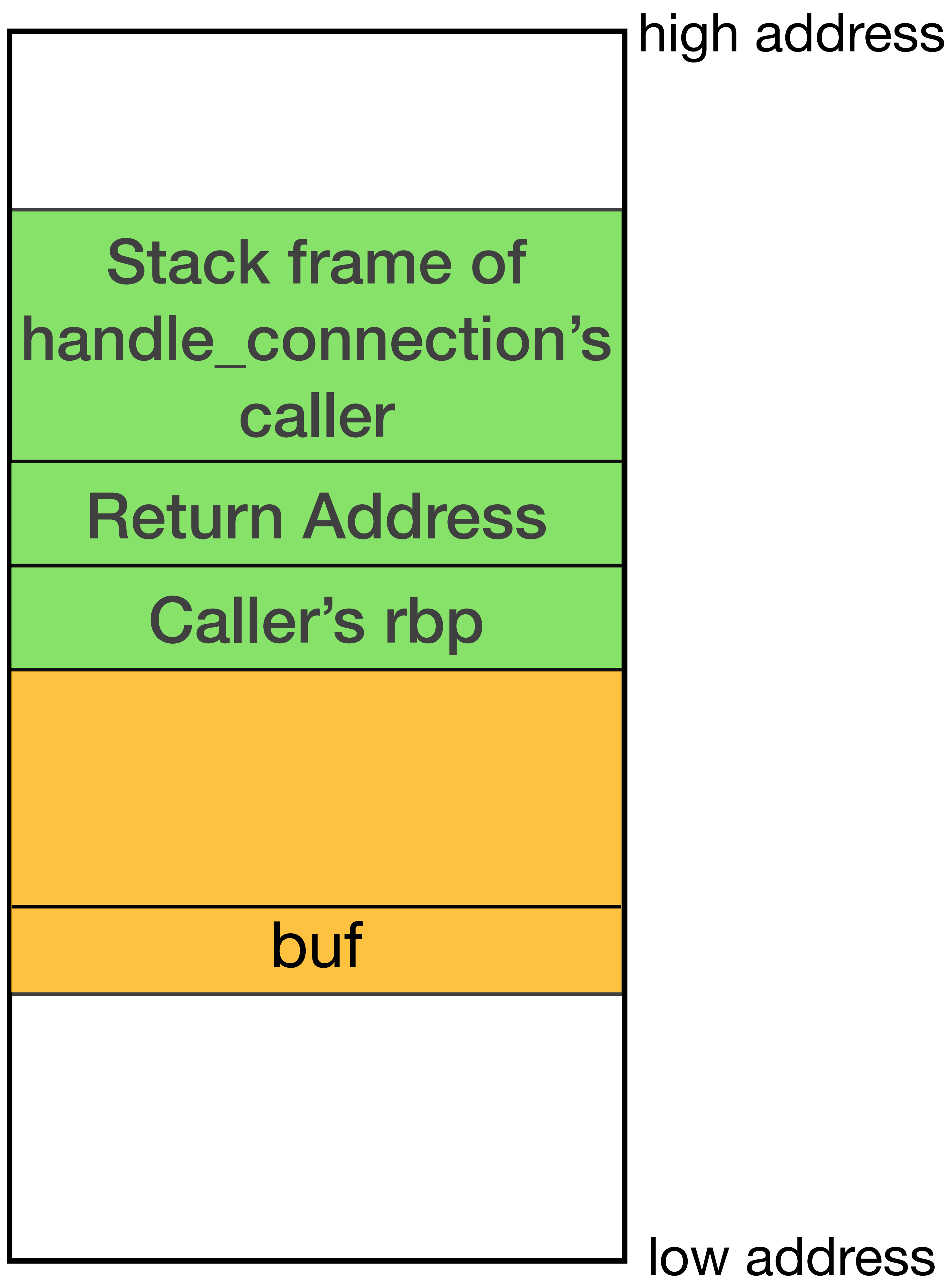
Stack before attack



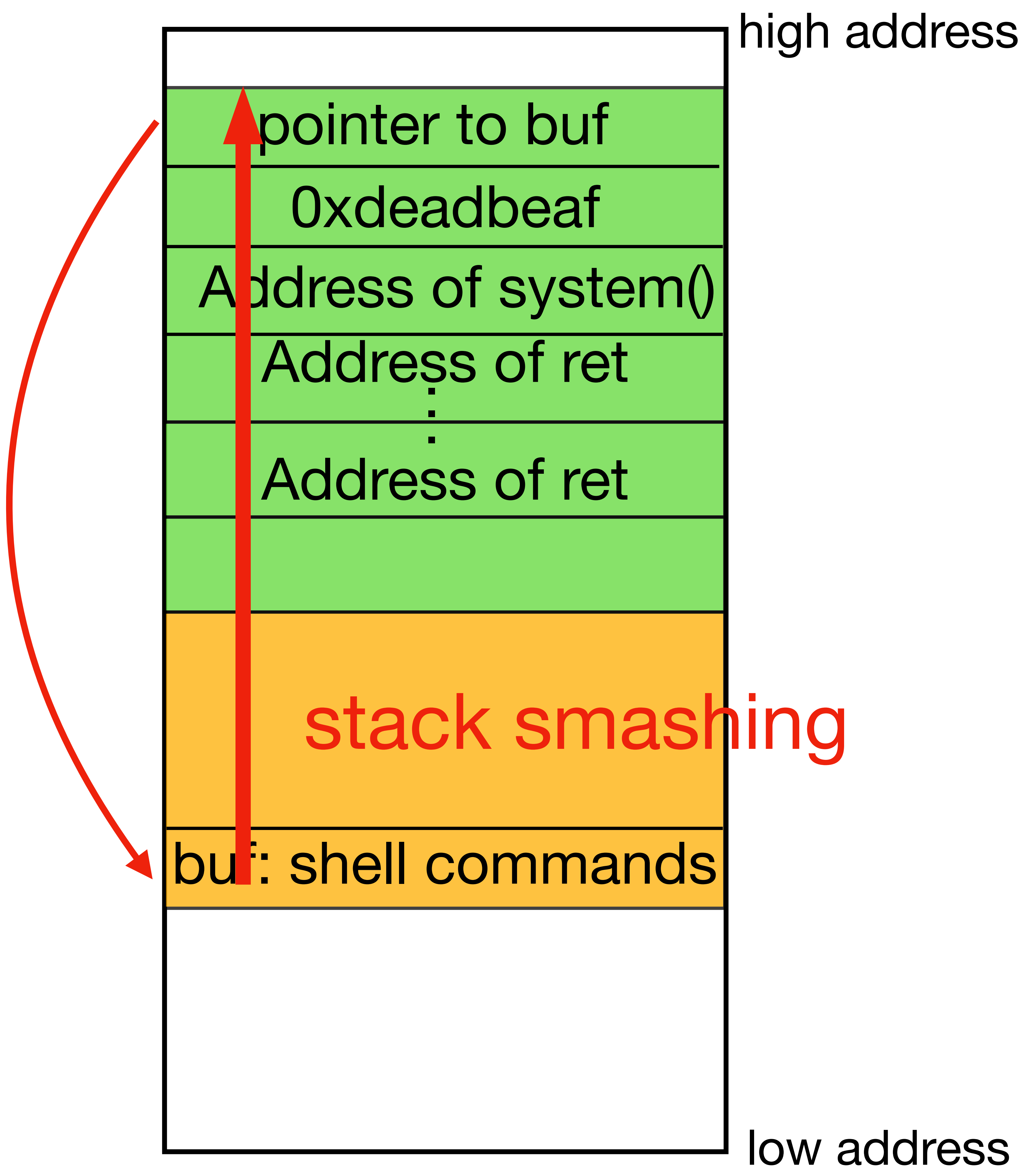
Stack after attack

Once succeeded, attackers would attain `delta_mmap`.

Exploit Step 2: Injecting `system()`'s Address



Stack before attack



Stack after attack

How Hard/Easy is the Attack?

- 16 bits of randomization for delta_mmap
  - Only need to try at most 2^16 = 65,536 times
- Experimental setup
  - Exploit executed on a 2.4 GHz Pentium 4 Linux machine
  - Against a PaX ASLR protected Linux running on Athlon 1.8 GHz machine
  - Running 10 trials
- Experimental results

Average	Max	Min
216 s	810 s	29 s

How to Improve ASLR?

- Use 64-bit systems
  - Limited to 36 bits to randomize on Intel x86-64, and 34 bits on Mac M chips
- More frequent randomization during execution
  - Randomize each new process
  - Randomize memory objects
  - Can be complicated and expensive
- Randomization at compile time
  - Randomize each function