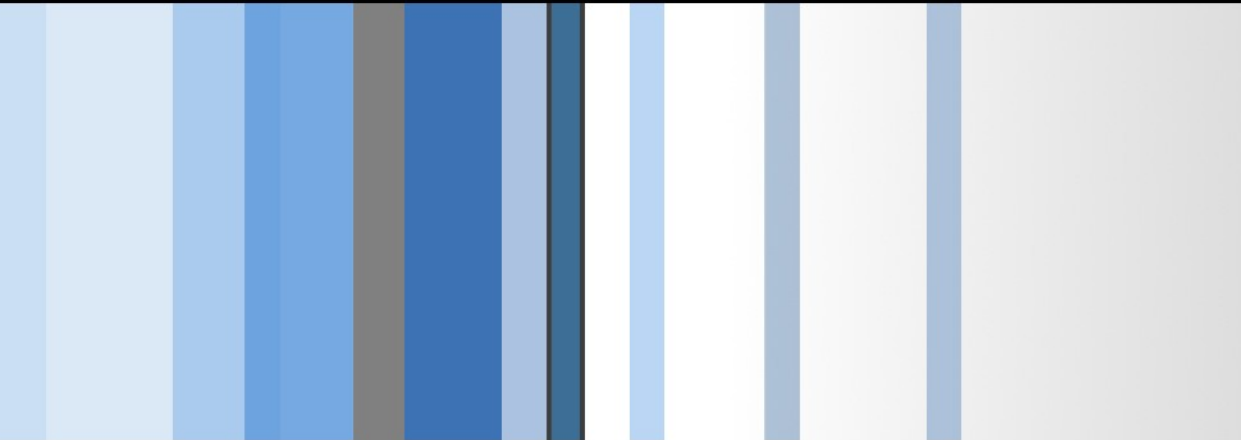


CSC 472 Software Security

ROP (3) & Dynamic Linking & Return-to-libc Attack

Dr. Si Chen (schen@wcupa.edu)



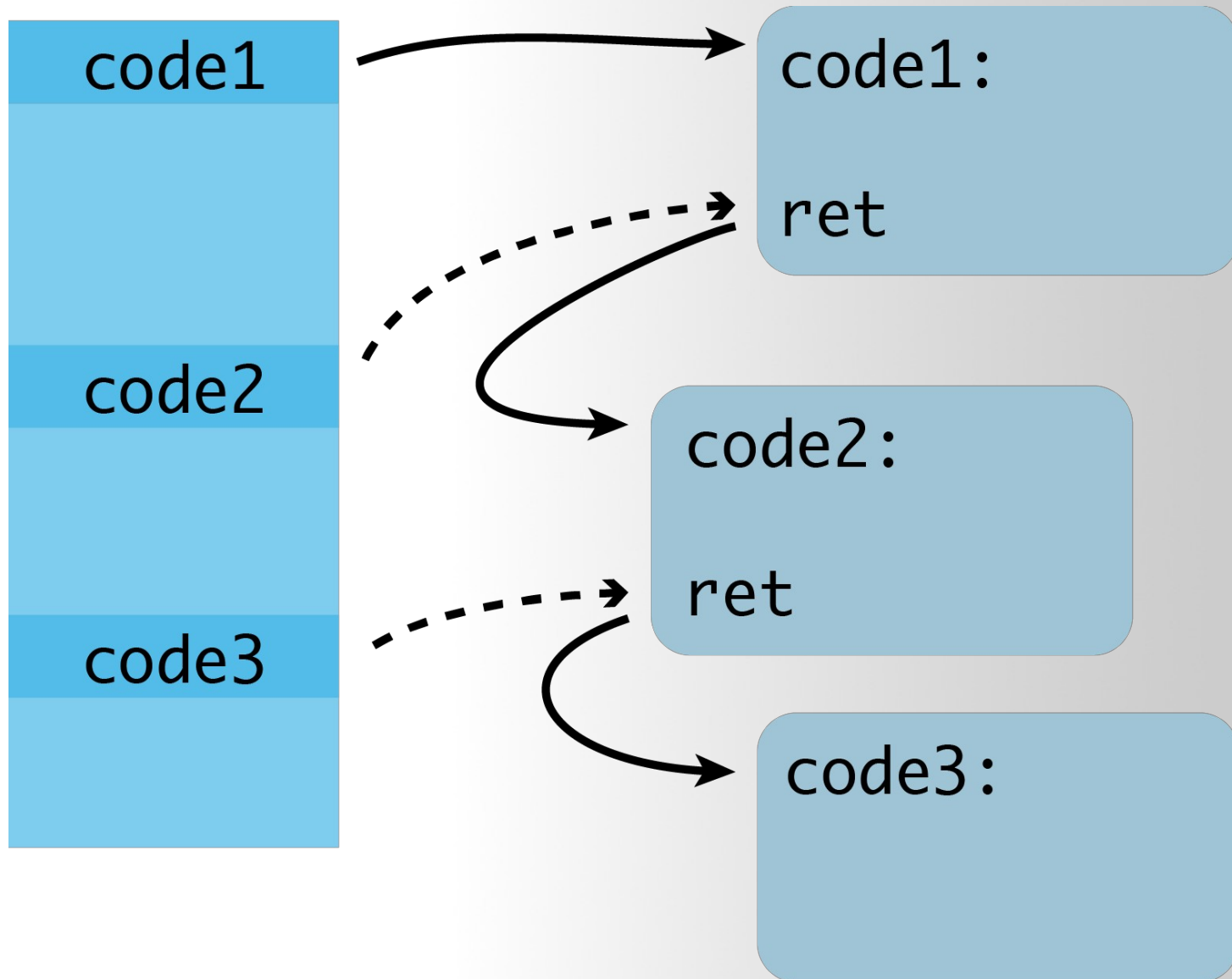
Review

Return-oriented programming (ROP)

Return-Oriented Programming

is A LOT LIKE a ransom
note, BUT instead of CUTTING
cut Letters FROM Magazines,
YOU ARE CUTTING OUT
instructions FROM text
segments

ROP: The Main Idea



```

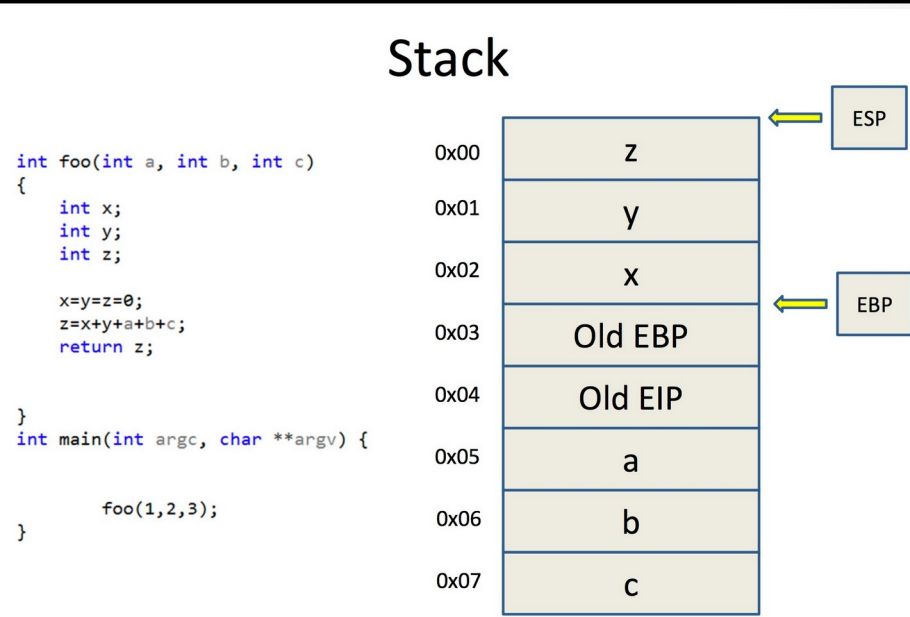
1  #include <stdio.h>
2  #include <string.h>
3
4  char string[100];
5
6  void exec_string() {
7      system(string);
8  }
9
10 void add_bin(int magic) {
11     if (magic == 0xdeadbeef) {
12         strcat(string, "/bin");
13     }
14 }
15
16 void add_bash(int magic1, int magic2) {
17     if (magic1 == 0xcafebabe && magic2 == 0x0badf00d) {
18         strcat(string, "/bash");
19     }
20 }
21
22 void vulnerable_function(char *string) {
23     char buffer[100];
24     gets(buffer);
25 }
26
27 int main(int argc, char** argv) {
28     string[0] = 0;
29     vulnerable_function(argv[1]);
30     return 0;
31 }

```

Execution Path

- add_bin()
 - magic == 0xdeadbeef
- add_bash()
 - magic1 == 0xcafebabe
 - magic2 == 0x0badf00d
- exec_string()
- Spawn shell

Return Chaining



First Function Address

Next Function Address

Arguments1 for First Function

Arguments2 for First Function

...

Return Chaining

The previous ROP chain does not work, because argument **0xdeadbeef** is still on the stack, we need to find a way to "**clean**" it

Add_bin()

→ magic == 0xdeadbeef

Add_bash()

Exec_string()

Execution Path

- add_bin()
 - magic == 0xdeadbeef
- add_bash()
 - magic1 == 0xcafebabe
 - magic2 == 0x0badf00d
- exec_string()
- Spawn shell

Solution: use a **pop, ret** gadget to push the argument **0xdeadbeef** into a **register** to remove it from the stack

Dummy Character "A"s

Address for Add_bin()

Address for pop_ret

0xdeadbeef

Address for Add_bash()

Multiple Dummy Character 'A' s
Address of add_bin()
Address of pop, ret gadget
0xdeadbeef
Address of add_bash()
Address of pop, pop, ret gadget
0xcafebabe
0x0badf00d
Address of exec_string()

ELF (Executable Linkable Format)

ELF executable for Linux

Executable and Linkable Format (ELF)

Linux	Windows	
ELF file	.exe (PE)	
.so (Shared object file)	.dll (Dynamic Linking Library)	
.a	.lib (static linking library)	
.o (intermediate file between compilation and linking, object file)	.obj	

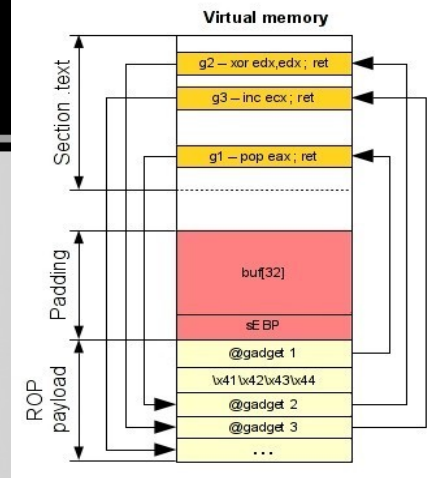
ELF executable for Linux

```
[quake0day@quake0day-wcu Downloads]$ file a
a: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=da2dba
f2eda3d2b639f8dac80396a994d2df0e, not stripped
```

- ELF32-bit LSB
- Dynamically linked

Section (or Segment) in ELF

```
1 int global_init_var = 84;
2 int golbal_uninit_var;
3
4 void func1(int i)
5 {
6     printf("%d\n", i);
7 }
8
9 int main(void)
10 {
11     static int static_var = 85;
12     static int static_var2;
13
14     int a = 1;
15     int b;
16     func1(static_var + static_var2 + a + b);
17     return 0;
18 }
```



ELF file /
Object file

File Header

.text section

.data section

.bss (Block started by
symbol) section

→ objdump -h rop2

rop: file format elf32-i386

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.interp	00000013	00000154	00000154	00000154	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
1	.note.ABI-tag	00000020	00000168	00000168	00000168	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
2	.note.gnu.build-id	00000024	00000188	00000188	00000188	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	.gnu.hash	00000020	000001ac	000001ac	000001ac	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.dynsym	00000090	000001cc	000001cc	000001cc	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
5	.dynstr	000000a2	0000025c	0000025c	0000025c	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
6	.gnu.version	00000012	000002fe	000002fe	000002fe	2**1
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
7	.gnu.version_r	00000030	00000310	00000310	00000310	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
8	.rel.dyn	00000040	00000340	00000340	00000340	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
9	.rel.plt	00000018	00000380	00000380	00000380	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
10	.init	00000023	00000398	00000398	00000398	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
11	.plt	00000040	000003c0	000003c0	000003c0	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
12	.plt.got	00000010	00000400	00000400	00000400	2**3
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
13	.text	000002e2	00000410	00000410	00000410	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
14	.fini	00000014	000006f4	000006f4	000006f4	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
15	.rodata	00000008	00000708	00000708	00000708	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
16	.eh_frame_hdr	0000005c	00000710	00000710	00000710	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
17	.eh_frame	0000018c	0000076c	0000076c	0000076c	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
18	.init_array	00000004	00001ed4	00001ed4	00000ed4	2**2
	CONTENTS, ALLOC, LOAD, DATA					
19	.fini_array	00000004	00001ed8	00001ed8	00000ed8	2**2
	CONTENTS, ALLOC, LOAD, DATA					
20	.dynamic	000000f8	00001edc	00001edc	00000edc	2**2
	CONTENTS, ALLOC, LOAD, DATA					
21	.got	0000002c	00001fd4	00001fd4	00000fd4	2**2
	CONTENTS, ALLOC, LOAD, DATA					
22	.data	00000008	00002000	00002000	00001000	2**2
	CONTENTS, ALLOC, LOAD, DATA					
23	.bss	00000084	00002020	00002020	00001008	2**5
	ALLOC					
24	.comment	00000024	00000000	00000000	00001008	2**0
	CONTENTS, READONLY					

→ ~ size rop
text data bss dec hex filename
1948 308 132 2388 954 rop

→ ~ objdump -h rop2

rop2: file format elf32-i386

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.note.ABI-tag	00000020	080480f4	080480f4	000000f4	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
1	.note.gnu.build-id	00000024	08048114	08048114	00000114	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
2	.rel.plt	00000078	08048138	08048138	00000138	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	.init	00000023	080481b0	080481b0	000001b0	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
4	.plt	00000078	080481d8	080481d8	000001d8	2**3
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
5	.text	00066f81	08048250	08048250	00000250	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
6	__libc_freeres_fn	00000ba7	080af1e0	080af1e0	000671e0	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
7	__libc_thread_freeres_fn	00000127	080afd90	080afd90	00067d90	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
8	.fini	00000014	080afeb8	080afeb8	00067eb8	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
9	.rodata	00018898	080afee0	080afee0	00067ee0	2**5
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
10	.eh_frame	000129b0	080c8778	080c8778	00080778	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
11	.gcc_except_table	000000ac	080db128	080db128	00093128	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
12	.tdata	00000010	080dc6e0	080dc6e0	000936e0	2**2
	CONTENTS, ALLOC, LOAD, DATA, THREAD LOCAL					
13	.tbss	00000020	080dc6f0	080dc6f0	000936f0	2**2
	ALLOC, THREAD LOCAL					
14	.init_array	00000008	080dc6f0	080dc6f0	000936f0	2**2
	CONTENTS, ALLOC, LOAD, DATA					
15	.fini_array	00000008	080dc6f8	080dc6f8	000936f8	2**2
	CONTENTS, ALLOC, LOAD, DATA					
16	.data.rel.ro	000018d4	080dc700	080dc700	00093700	2**5
	CONTENTS, ALLOC, LOAD, DATA					
17	.got	00000028	080ddfd4	080ddfd4	00094fd4	2**2
	CONTENTS, ALLOC, LOAD, DATA					
18	.got.plt	00000048	080de000	080de000	00095000	2**2
	CONTENTS, ALLOC, LOAD, DATA					
19	.data	00000f20	080de060	080de060	00095060	2**5
	CONTENTS, ALLOC, LOAD, DATA					
20	__libc_subfreeres	00000024	080def80	080def80	00095f80	2**2
	CONTENTS, ALLOC, LOAD, DATA					
21	__libc_IIO_vtables	00000354	080defc0	080defc0	00095fc0	2**5
	CONTENTS, ALLOC, LOAD, DATA					
22	__libc_atexit	00000004	080df314	080df314	00096314	2**2
	CONTENTS, ALLOC, LOAD, DATA					
23	__libc_thread_subfreeres	00000004	080df318	080df318	00096318	2**2
	CONTENTS, ALLOC, LOAD, DATA					
24	.bss	00000cdc	080df320	080df320	0009631c	2**5
	ALLOC					
25	__libc_freeres_ptrs	00000014	080dfff0	080dfff0	0009631c	2**2
	ALLOC					
26	.comment	00000024	00000000	00000000	0009631c	2**0
	CONTENTS, READONLY					

→ ~ size rop2
text data bss dec hex filename

rop2.c

```
→ ~ gcc -m32 -fno-stack-protector -static -znoexecstack -o rop2 ./rop2.c
```

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

void vuln() {
    char buffer[128];
    char * second_buffer;
    uint32_t length = 0;
    puts("Reading from STDIN");
    read(0, buffer, 1024);

    if (strcmp(buffer, "Cool Input") == 0) {
        puts("What a cool string.");
    }
    length = strlen(buffer);
    if (length == 42) {
        puts("LUE");
    }
    second_buffer = malloc(length);
    strncpy(second_buffer, buffer, length);
}

int main() {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    puts("This is a big vulnerable example!");
    printf("I can print many things: %x, %s, %d\n", 0xdeadbeef, "Test String",
        42);
    write(1, "Writing to STDOUT\n", 18);
    vuln();
}
```

Since the binary is not big enough to give us a decent number of ROP gadgets, we will cheat a bit and compile the binary as a **statically linked ELF**.

This should include library code in the final executable and bulk up the size of the binary.

Linux System Call

- If we take a look at the syscall reference, we can see that some parameters are expected in the eax, ebx, ecx, and edx registers.
 - eax - holds the number of the syscall to be called
 - ebx - a pointer to the string containing the file name to be executed
 - ecx - a pointer to the array of string pointers representing argv
 - edx - a pointer to the array of string pointers representing envp
- For our purposes, the value that each of the registers should contain are:

eax = 0xb
ebx = "/bin/sh"
ecx = memory address -> 0
edx = memory address -> 0

ROPgadget

```
[quake0day@quake0day-wcu ~]$ ROPgadget --binary main
```

```
Gadgets information
```

```
=====
```

```
0x080cc211 : aaa ; add byte ptr [eax], al ; cmp al, 0xb5 ; clc ; jmp dword ptr [ecx]
```

```
0x080cc239 : aaa ; add byte ptr [eax], al ; mov ch, 0xf8 ; jmp dword ptr [ecx]
```

```
0x080cc225 : aaa ; add byte ptr [eax], al ; pop eax ; mov ch, 0xf8 ; call dword ptr [edi]
```

```
0x08096d2c : aaa ; add esp, 0x70 ; pop ebx ; pop esi ; pop edi ; ret
```

```
0x0809c8a7 : aaa ; div bh ; add esp, 0x10 ; pop ebx ; pop esi ; pop edi ; ret
```

```
0x080ace79 : aaa ; push 1 ; push 1 ; call eax
```

```
0x0804e4ab : aaa ; push dword ptr [ebx] ; mov eax, dword ptr [esp + 0x20] ; call eax
```

```
0x0807a21a : aad 0x2d ; ret 0
```

```
0x080aac19 : aad 0x89 ; ret 0xe283
```

```
→ ~ ROPgadget --binary rop2 --ropchain
```

Dynamic Linking

rop2.c

```
→ ~ gcc -m32 -fno-stack-protector -static -znoexecstack -o rop2 ./rop2.c
```

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

void vuln() {
    char buffer[128];
    char * second_buffer;
    uint32_t length = 0;
    puts("Reading from STDIN");
    read(0, buffer, 1024);

    if (strcmp(buffer, "Cool Input") == 0) {
        puts("What a cool string.");
    }
    length = strlen(buffer);
    if (length == 42) {
        puts("LUE");
    }
    second_buffer = malloc(length);
    strncpy(second_buffer, buffer, length);
}

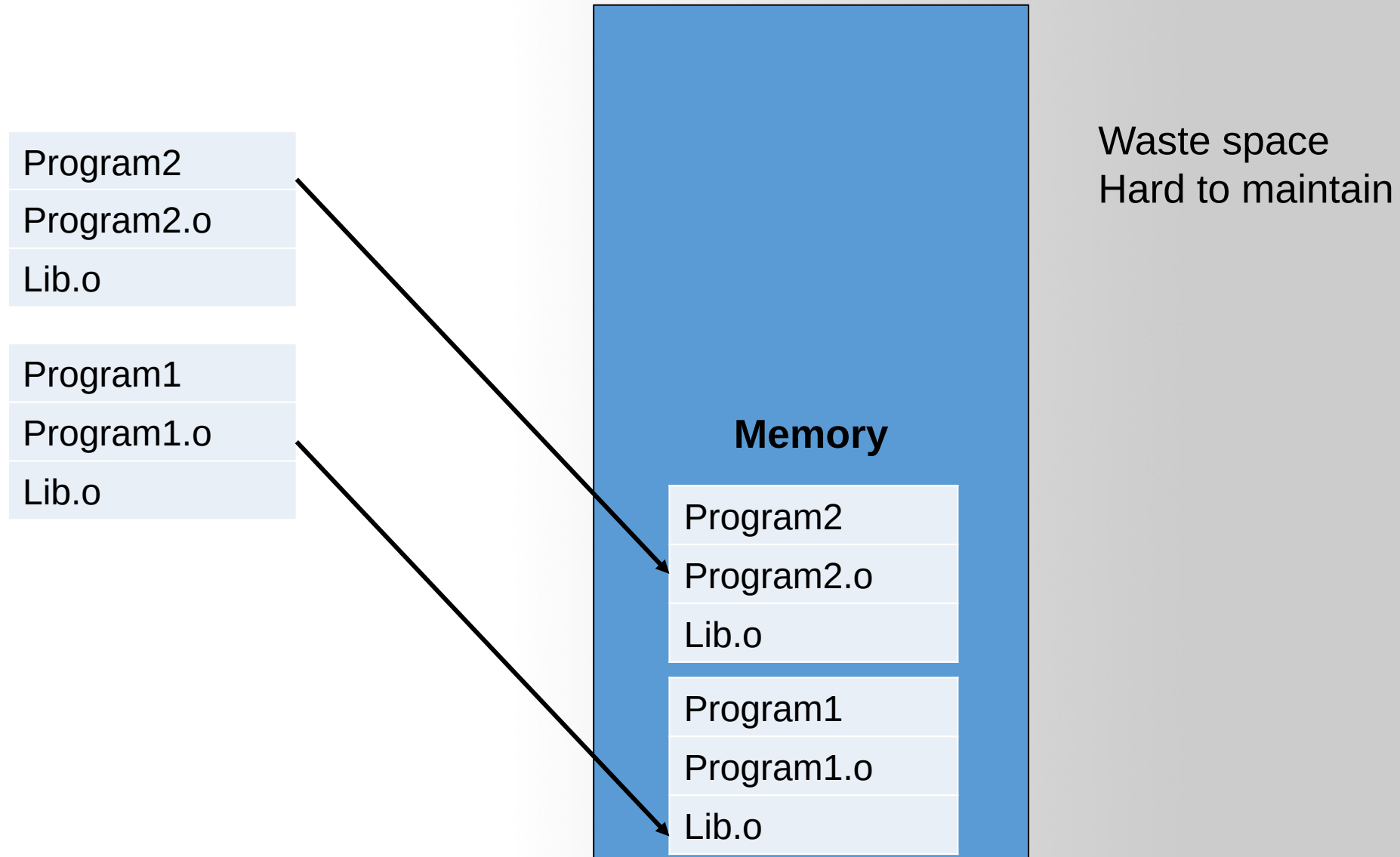
int main() {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    puts("This is a big vulnerable example!");
    printf("I can print many things: %x, %s, %d\n", 0xdeadbeef, "Test String",
        42);
    write(1, "Writing to STDOUT\n", 18);
    vuln();
}
```

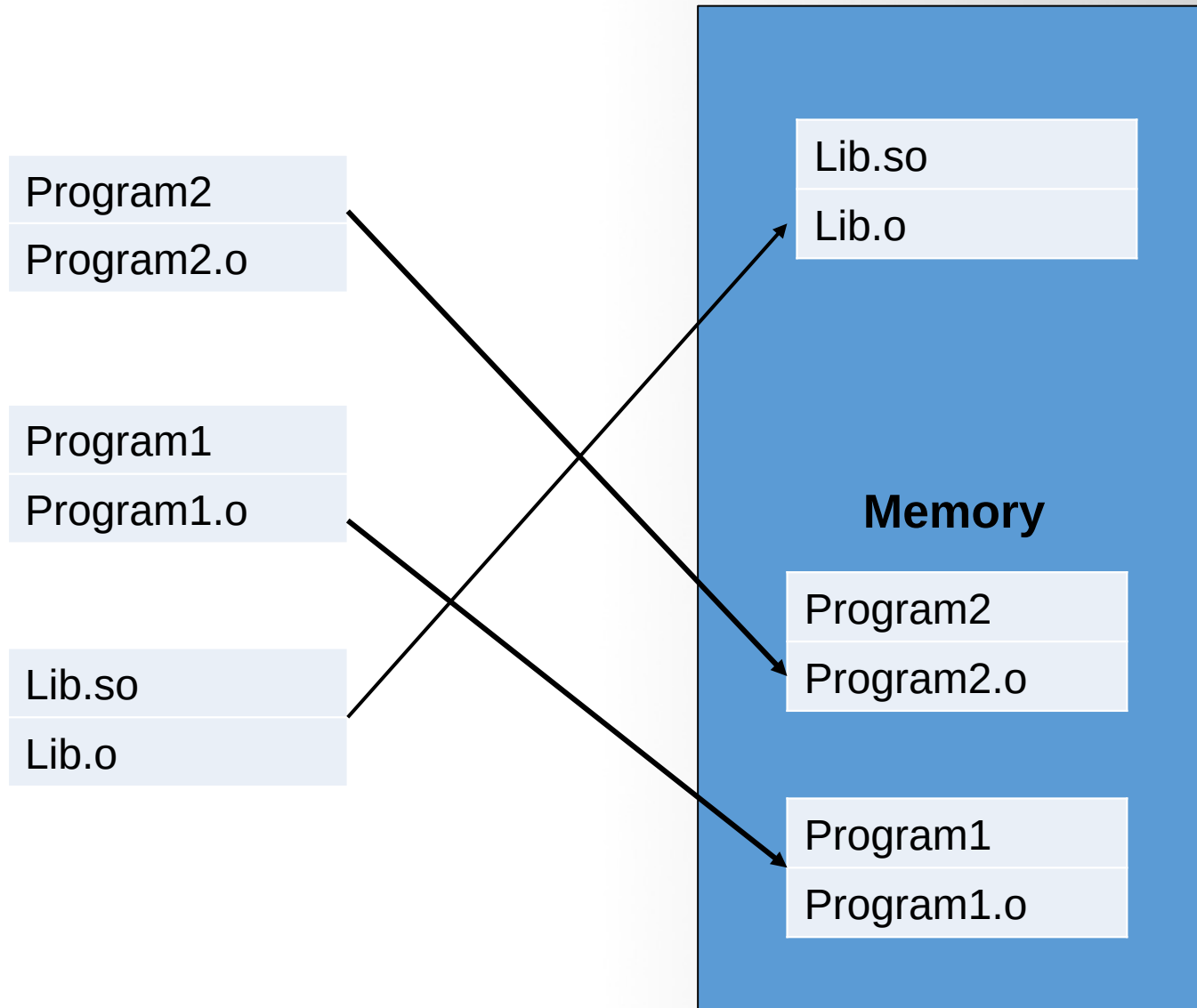
Since the binary is not big enough to give us a decent number of ROP gadgets, we will cheat a bit and compile the binary as a **statically linked ELF**.

This should include library code in the final executable and bulk up the size of the binary.

Drawbacks of Static Linking



Dynamic Linking



Dynamic Linking in Linux and Windows

Linux	Windows	
ELF file	.exe (PE)	
.so (Shared object file)	.dll (Dynamic Linking Library)	
.a	.lib (static linking library)	
.o (intermediate file between compilation and linking, object file)	.obj	

Shared library

```
[quake0day@quake0day-wcu Downloads]$ ldd ./a
linux-gate.so.1 (0xb77c5000)
libc.so.6 => /usr/lib/libc.so.6 (0xb75dd000)
/lib/ld-linux.so.2 (0xb77c7000)
```

- ELF is loaded by **ld-linux.so.2** in charge of memory mapping, load shared library etc..
- You can call functions in **libc.so.6**

Return-Oriented Programming (ROP)

What happens if the **binary we have to attack is not large enough** to provide us the gadgets we need?

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.

- 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

reveal_address.c

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <unistd.h>

int main() {
    puts("This program helps visualise where libc is loaded.\n");
    int pid = getpid();
    char command[500];
    puts("Memory Layout: ");
    sprintf(command, "cat /proc/%d/maps", pid);
    system(command);
    puts("\nFunction Addresses: ");
    printf("System@libc 0x%lx\n", dlsym(RTLD_NEXT, "system"));
    printf("PID: %d\n", pid);
    puts("Press enter to continue.");
    read(0, command, 1);
}
```

reveal_address 32 bit version

```
➔ ~ ./reveal_address32
```

This program helps visualise where libc is loaded.

Memory Layout:

56555000-56556000	r-xp	00000000	08:00	42372	/root/reveal_address32
56556000-56557000	r--p	00000000	08:00	42372	/root/reveal_address32
56557000-56558000	rw-p	00001000	08:00	42372	/root/reveal_address32
56558000-5657a000	rw-p	00000000	00:00	0	[heap]
f7de8000-f7fba000	r-xp	00000000	08:00	640006	/lib32/libc-2.27.so
f7fba000-f7fbb000	---p	001d2000	08:00	640006	/lib32/libc-2.27.so
f7fbb000-f7fbd000	r--p	001d2000	08:00	640006	/lib32/libc-2.27.so
f7fbd000-f7fbe000	rw-p	001d4000	08:00	640006	/lib32/libc-2.27.so
f7fbe000-f7fc1000	rw-p	00000000	00:00	0	
f7fc1000-f7fc4000	r-xp	00000000	08:00	640009	/lib32/libdl-2.27.so
f7fc4000-f7fc5000	r--p	00002000	08:00	640009	/lib32/libdl-2.27.so
f7fc5000-f7fc6000	rw-p	00003000	08:00	640009	/lib32/libdl-2.27.so
f7fcf000-f7fd1000	rw-p	00000000	00:00	0	
f7fd1000-f7fd4000	r--p	00000000	00:00	0	[vvar]
f7fd4000-f7fd6000	r-xp	00000000	00:00	0	[vdso]
f7fd6000-f7ffc000	r-xp	00000000	08:00	640002	/lib32/ld-2.27.so
f7ffc000-f7ffd000	r--p	00025000	08:00	640002	/lib32/ld-2.27.so
f7ffd000-f7ffe000	rw-p	00026000	08:00	640002	/lib32/ld-2.27.so
ffffd000-ffffe000	rw-p	00000000	00:00	0	[stack]

Function Addresses:

System@libc 0xf7e24d10

PID: 27150

Press enter to continue.

/lib32/libc-2.27.so

Memory

/lib32/libc-2.27.so

reveal_address 64 bit version

```
➔ ~ ./reveal_address64  
This program helps visualise where libc is loaded.
```

```
Memory Layout:  
555555554000-555555555000 r-xp 00000000 08:00 42371 /root/reveal_address64  
555555754000-555555755000 r--p 00000000 08:00 42371 /root/reveal_address64  
555555755000-555555756000 rw-p 00001000 08:00 42371 /root/reveal_address64  
555555756000-555555777000 rw-p 00000000 00:00 0 [heap]  
7ffff77e0000-7ffff79c7000 r-xp 00000000 08:00 571 /lib/x86_64-linux-gnu/libc-2.27.so  
7ffff79c7000-7ffff7bc7000 --p 001e7000 08:00 571 /lib/x86_64-linux-gnu/libc-2.27.so  
7ffff7bc7000-7ffff7bcb000 r--p 001e7000 08:00 571 /lib/x86_64-linux-gnu/libc-2.27.so  
7ffff7bcb000-7ffff7bcd000 rw-p 001eb000 08:00 571 /lib/x86_64-linux-gnu/libc-2.27.so  
7ffff7bcd000-7ffff7bd1000 rw-p 00000000 00:00 0  
7ffff7bd1000-7ffff7bd4000 r-xp 00000000 08:00 588 /lib/x86_64-linux-gnu/libdl-2.27.so  
7ffff7bd4000-7ffff7dd3000 --p 00003000 08:00 588 /lib/x86_64-linux-gnu/libdl-2.27.so  
7ffff7dd3000-7ffff7dd4000 r--p 00002000 08:00 588 /lib/x86_64-linux-gnu/libdl-2.27.so  
7ffff7dd4000-7ffff7dd5000 rw-p 00003000 08:00 588 /lib/x86_64-linux-gnu/libdl-2.27.so  
7ffff7dd5000-7ffff7dfc000 r-xp 00000000 08:00 547 /lib/x86_64-linux-gnu/ld-2.27.so  
7ffff7fe9000-7ffff7fee000 rw-p 00000000 00:00 0  
7ffff7ff7000-7ffff7ffa000 r--p 00000000 00:00 0 [vvar]  
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0 [vdso]  
7ffff7ffc000-7ffff7ffd000 r--p 00027000 08:00 547 /lib/x86_64-linux-gnu/ld-2.27.so  
7ffff7ffd000-7ffff7ffe000 rw-p 00028000 08:00 547 /lib/x86_64-linux-gnu/ld-2.27.so  
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0  
7ffff7ffde000-7ffff7fff000 rw-p 00000000 00:00 0  
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [stack]  
[vsyscall]
```

```
Function Addresses:  
System@libc 0x7ffff782f440  
PID: 27160  
Press enter to continue.
```

/lib/x86_64-linux-
gnu/libc-2.27.so

Memory

/lib/x86_64-linux-gnu/libc-2.27.so

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")

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

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

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

Dummy Characters

Address for System() in libc

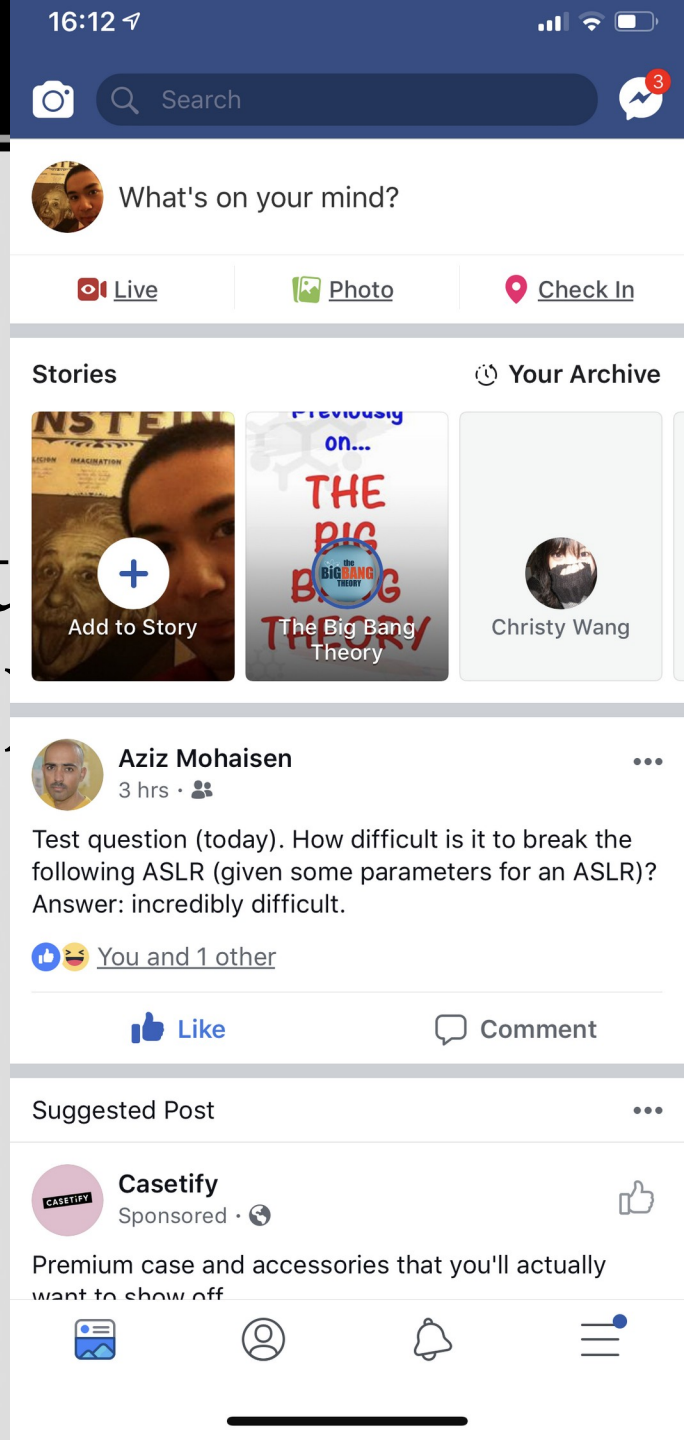
Address for Exit() function in libc (if you want to exit the program gracefully)

Address for Command String ("e.g. /bin/sh")

Hands-on: ret2libc on WolfCTF

- Open the demo in your browser terminal — GDB + GEF ready, no setup.
- ret2libc.c is pre-built with NX on (no execstack) and -no-pie.
- Walk the exploit end to end:
 - find the 76-byte offset to the saved return address (cyclic pattern),
 - locate system(), exit() and "/bin/sh" in the process's own loaded libc,
 - forge the call frame [&system][&exit][&"/bin/sh"] and pop a shell.
- Demo: portal.wolfctf.com/labs/csc472-ret2libc-demo

ASLR in Depth (not really...)



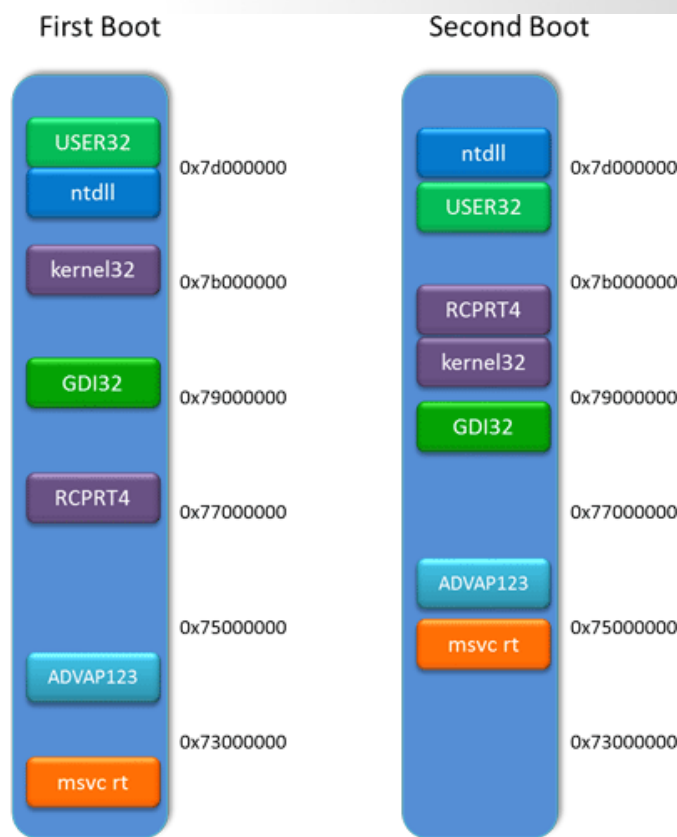
Shutdown ASLR

```
[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.



ASLR in Practice — what moves, what doesn't

- Randomized every run: the stack, the heap, the mmap region, and shared libraries (libc).
- A no-PIE executable is NOT randomized: its code, PLT and GOT stay at fixed addresses.
- 32-bit libc has low entropy (only a handful of random bits) — even brute-forceable; 64-bit is far stronger.
- Lesson: hard-coded libc addresses break under ASLR — but fixed in-binary addresses (PLT/GOT) still work.

Defeating ASLR — leak, rebase, pwn

- You cannot guess libc's base, so leak one real libc address at runtime.
- e.g. call `puts(puts@GOT)` to print puts's resolved runtime address.
- $\text{libc_base} = \text{leaked_addr} - \text{offset}(\text{puts})$; then `system / "/bin/sh"` = `libc_base + their offsets`.
- Now the classic ret2libc frame works even with ASLR ON.
- Next class: ret2plt / ret2got reuse the binary's OWN fixed PLT/GOT — no libc leak needed.

Glossary of Terms

- **ASLR (Address Space Layout Randomization):** Security measure in modern OSes to randomize stack and libc addresses on each program execution.
- **Binary:** A binary is the output file from compiling a C or C++ file. Anything in the binary has a *constant address*.
- **Canary:** A canary is some (usually random) value that is used to verify that nothing has been overwritten. 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.
- **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.
- **Stack:** The stack is part of the memory for a binary. Local variables and pointers are often stored here. The stack can be randomized.
- **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.

Q & A