Class 13

CSC 472 Software Security
Return-to-libc Attack

Dr. Si Chen (schen@wcupa.edu)

Review

Return-oriented programming (ROP)

rop2.c

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

```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

- f 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

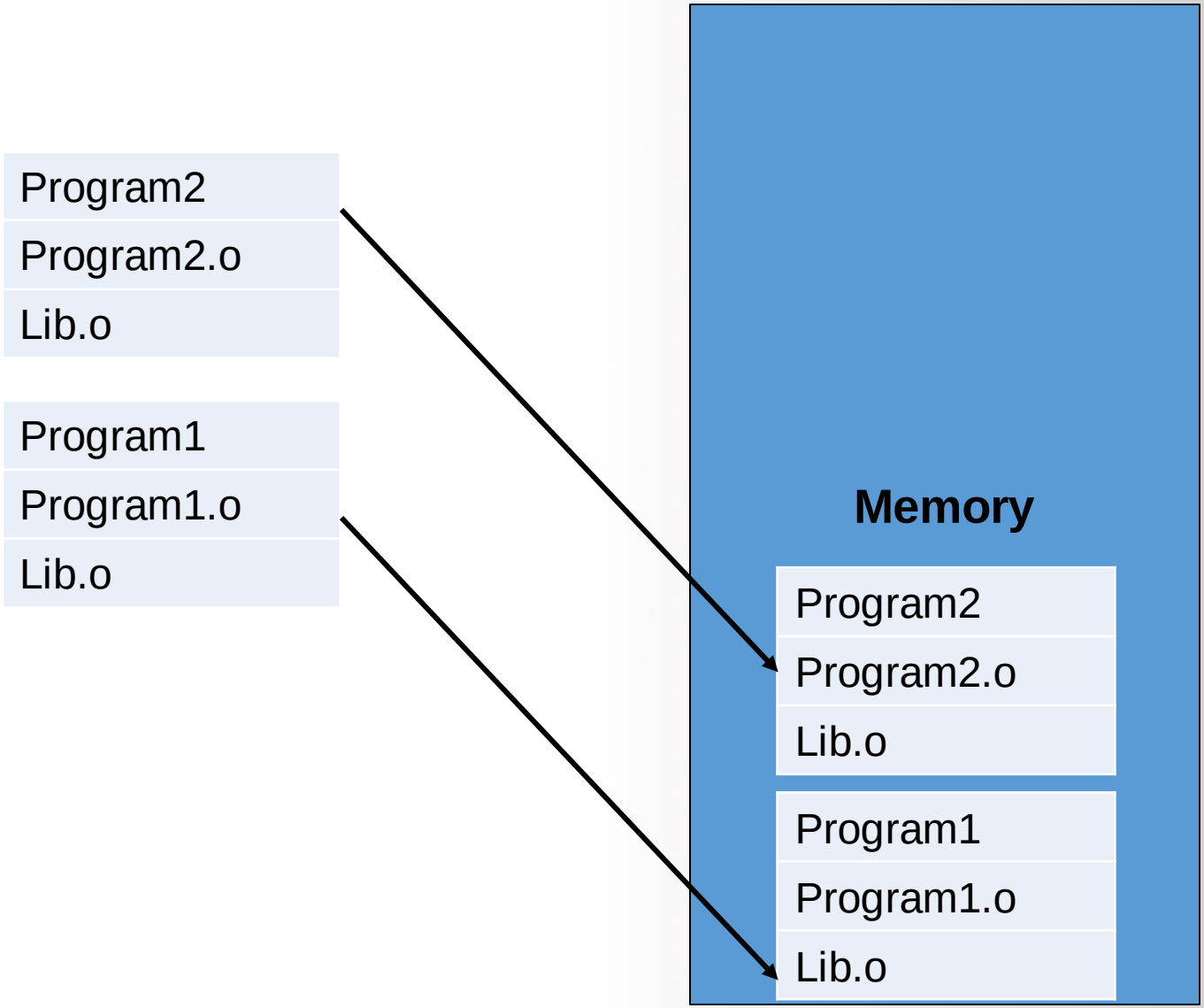
rop2.c

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

```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();
}
```

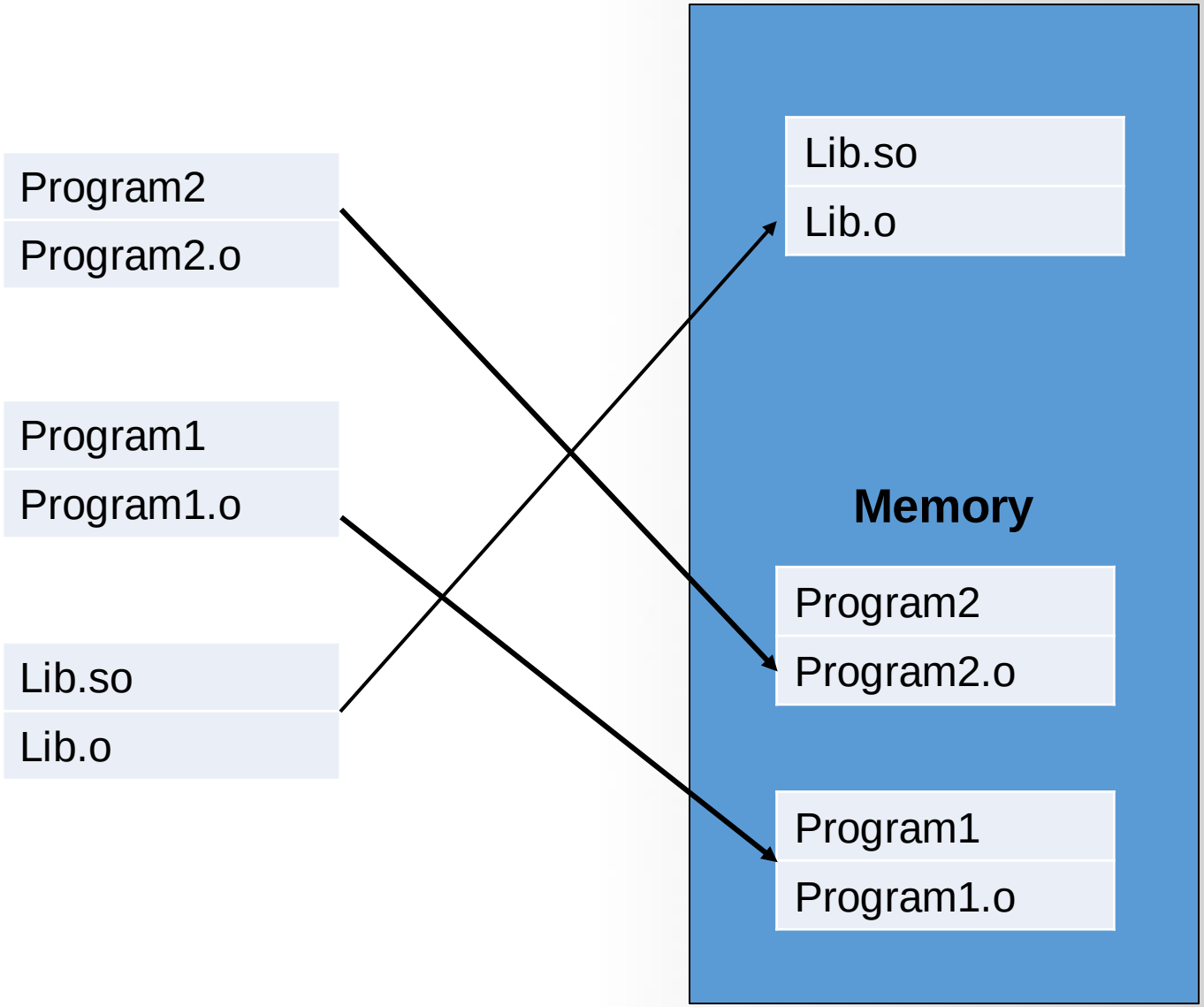
Since the binary is not big enough to give us a decent number of ROP gadgets, we will cheat a bit and compile the binary as a **statically linked ELF**.
This should include library code in the final executable and bulk up the size of the binary.

Drawbacks of Static Linking



Waste space
Hard to maintain

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 complication 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 Orientated 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.

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>

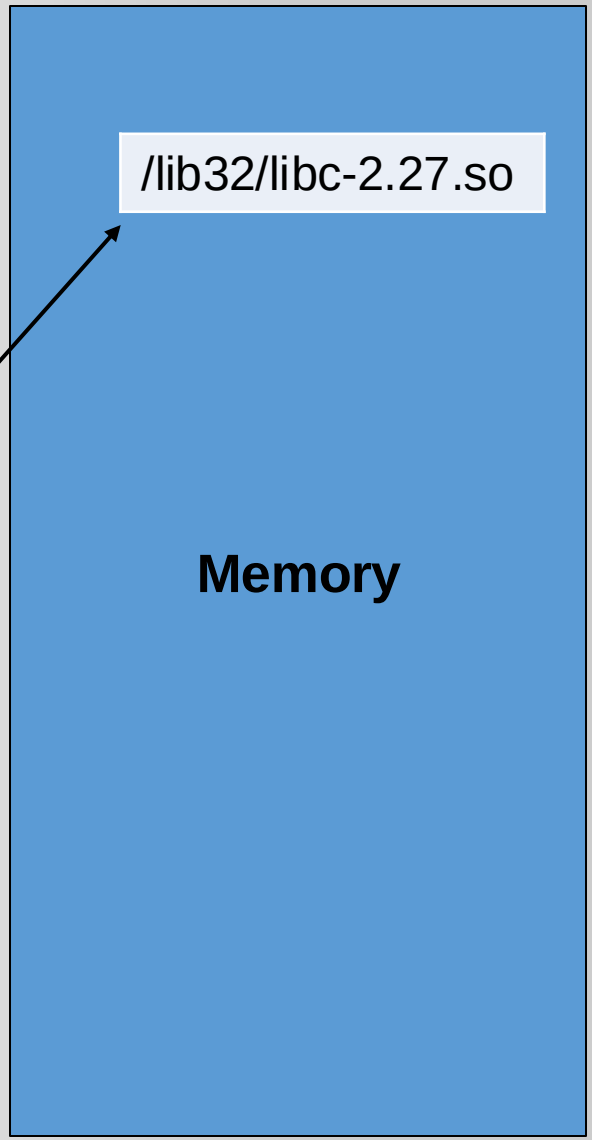
However, if we had these addresses into libc, **we could simplify our exploit to reuse useful functions**. One such useful function could be the system() function.
→ find System() function's address

reveal_address.c

```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_adddress 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
fffdd000-ffffe000 rw-p 00000000 00:00 0                                  [stack]

Function Addresses:
System@libc 0xf7e24d10
PID: 27150
Press enter to continue.
```



/lib32/libc-2.27.so

reveal_adddress 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
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]

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

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

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

Memory

Ret2lib Shellcode Structure

Function Address
Return Address (Old EIP)
Arguments

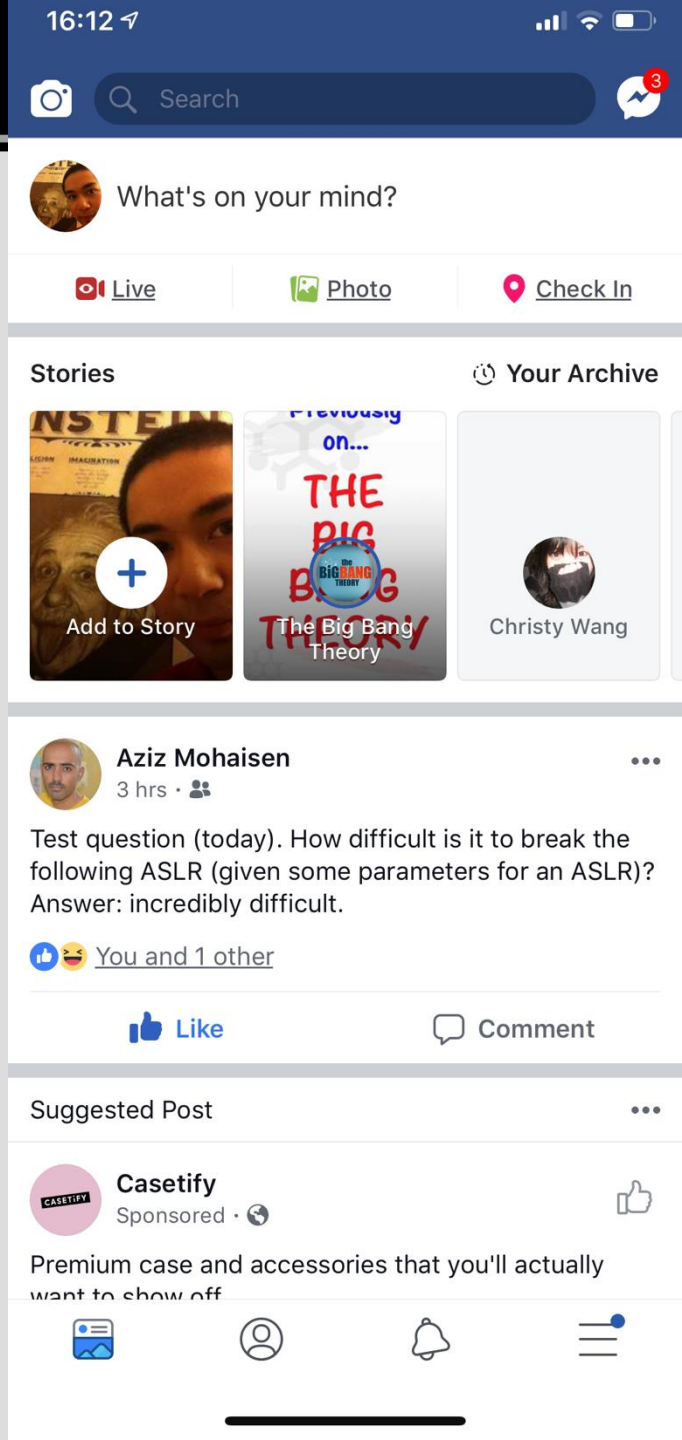
Dummy Characters
Address for System() in libc
Address for Exit() function in libc (if you want to exit the program gracefully)
Address for Command String (“e.g. /bin/sh”)

ret2lib.c

```c
#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”)

ASLR in Depth (not really…)



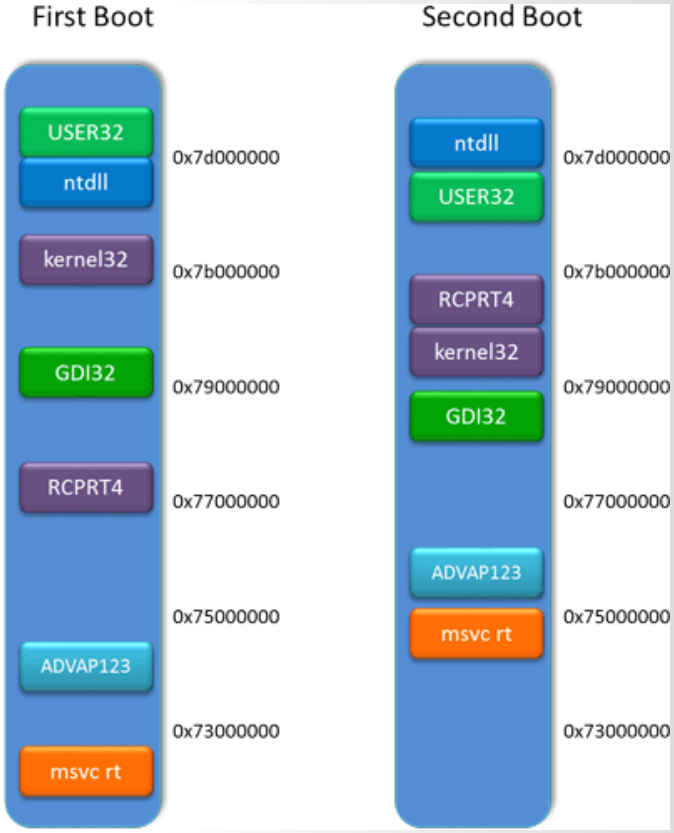
Shutdown ASLR

```
[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.



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 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.
- **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