Buffer Overflow Attack

Introduction

Memory has sequential blocks of memory in which static and dynamic variables are held. "Buffer overflow" is the overriding of these memory blocks by loading more than the amount of data that can be carried by unbanked memory blocks. Typically, these fields are accessed by copying a lot of characters from one memory area to another. This event causes memory overflow in the system [1].

Aim of the project is taking access of the root of the Seed-Ubuntu OS via buffer over-flow attack and achieving buffer-overflow attack in detail. For that purpose, buffer-overflow related configurations done for first Task1 in Seed-Ubuntu.

Furthermore, in task 1 exploiting the vulnerability, buffer-overflow attack with fixed memory addresses. In task 2 Address Randomization, buffer-overflow attack with randomized memory, and in task 3, Stack Guard, preventing from buffer-overflow attack done.

Configuration

Very initial part of the project is installing Seed-Ubuntu, according to TA's instruction check list Seed-Ubuntu installation successfully achieved on the virtual machine. There is no issue faced while installing and running Seed-Ubuntu on virtual machine.

In the project, first root access taken with below shell command [Figure 1] for disabling the non-executable stack and Stack-Guard protections.

```
$ su root
    Password (enter root password)
  # gcc -o stack -z execstack -fno-stack-protector stack.c
  # chmod 4755 stack
  # exit
```

Shell Command [1]: Shell Command

- "stack.c" file was created and filled with code which is given from project description.

- "exploit.c" file was created and filled with appendix code snippet.

```
$ su root
    Password: (enter root password)
  # /sbin/sysctl -w kernel.randomize_va_space=2
```

Figure [2]: Address Randomization

Above code allows to enable memory randomization [figure 2, 2]. (For closing randomization "space" should be set to "0")

Project Task 1

Before start…

We take advantage of the "For instance, if the attack code needs to execute "exec("/bin/sh")", and there exists code in libc that executes "exec(arg)" where "arg" is a string pointer argument, then the attacker need only change a pointer to point to "/bin/sh" and jump to the appropriate instructions in the libc library" [7]

GNU GDB debugger was used in this project. This links shows how to use GNU GDB. GDB debugger a lot helps to solve many issues.

After project configuration and environment setup, in this task exploit.c file edited (Appendix (exploit.c)).

Finding Return Address

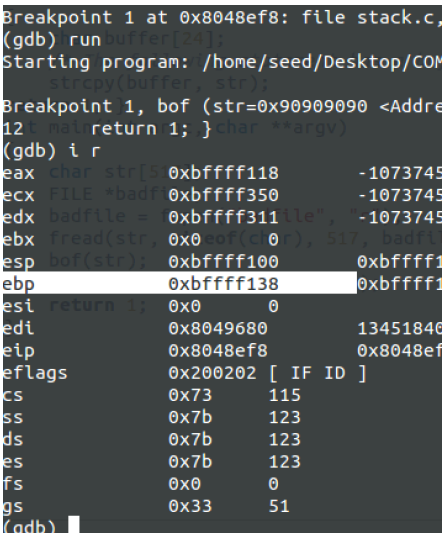
Before starting project, return address of the "buf" function. For that purpose, as mentioned GNU – GDB debugger was used for extended use of the GNU – GDB debugger. GDB output and commands were compiled as below shown. "buf" function allows to us do buffer-overflow attack. "strcpy(buffer, str);" in "buf" function allows to inserting malicious code snippet into stack.

"gdb stack" starts GNU-GDB

"break 12" buf function "return 0" line. (break point added here to getting return address of the "buf" function.)

"run" command starts debugging.

Result (figure 3) of the debug shown in next page...



Terminology of the results [3].

EBP - used to access data on stack
- when this register is used to specify an address, SS is used implicitly [3]

EIP - program counter (instruction pointer), relative to CS [3]

ESP - stack pointer, relative to SS [3]

According to above terminology, we need to take "ebp" address of the results.

This hexadecimal number "0xbfff138" gives us exactly location. This number will be used in the filling exploid.c code snippet.

Finding Buffer Size

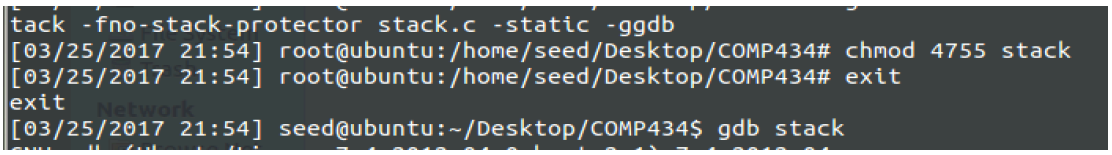
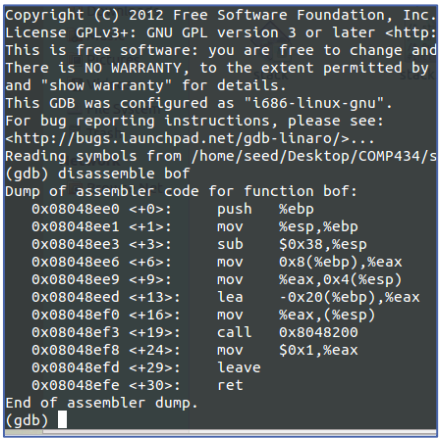


Figure [4]: -static –ggdb usage

Above screenshot (figure 4) "–static –ggdb" allows to detailed debug of the code snippet.

After entering this shell commend "gdb stack" to terminal opens GNU-GDB debugger. In debugger typing "disassemble bof" allows to show assembly version of the "bof" method in stack.c.

Left hand side GNU-GDB output shows assembly version of the "bof" method. In this output, 0x38 is allocation size of the buffer size. When we convert it to the decimal it returns us the buffer size which is 56.



Figure[5]-->

0xbfff138 + 56 gives return address of the "buf" where buffer attack held.

Creating "exploid.c"

After finding return address of the "buf" method. We can start implement our exploit.c file. "exploit.c" file's task is creating content for badfile [2]. Shell code and buffer size already implemented for us.

Starting implementation of the exploit.c file is below code sniped.

```
40    int num = sizeof(buffer) - (sizeof(shellcode) + 1);
```

This code calculates the shortest area to shellcode fit. It increases to correctly work of our shell code probability. In detail, this operation increases no-op amount [6].

Below code allows to make our buffer point out to long type address since our buffer is buffer.

```
41    addrpointer = (long*)(pointer);
```

Then we initialize our return address what we found from GNU-GDB debugger

```
47    return_address = 0xbffff138 + 56;
```

For return address we insert into address pointer with below code.

```
for (count = 0; count < addressLength; count++) /
*(addrpointer++) =return_address;
```

With the below code we finalize buffer-over flow attack.

```
for(count = 0; count < strlen(shellcode); count++) // Shell load
buffer[num+count] =  shellcode[count];
```

Above code allows to increase hit rate with adding shellcode at the very end of the code.

Compiling and Executing "exploid.c"

Warning! exploit and stack "c" files should be in the same directory.

With gcc –o exploit exploit.c method allows to compile our malicious c code. Then with ./exploit and ./stack buffer-over flow attack work perfectly.

```
[03/25/2017 23:16] seed@ubuntu:~/Desktop/COMP434$ gcc -o exploit exploit.c
[03/25/2017 23:16] seed@ubuntu:~/Desktop/COMP434$ ./exploit
[03/25/2017 23:17] seed@ubuntu:~/Desktop/COMP434$ ./stack
#
```

Project Task 2

In project 2, we kind a use brute force to get root access. First need to go back to configuration part and revisit to Figure 2 related part. And set “space” to “2”.

```
$ su root
    Password: (enter root password)
  # /sbin/sysctl -w kernel.randomize_va_space=2
```

Figure 2: RECALL

Again reopen terminal and go to directory where your exploit and stack c files present.

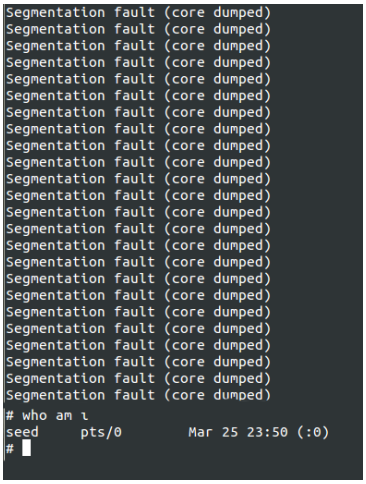
Run the below code.

>> $ sh -c "while [1]; do ./stack; done;"

Fallowing code depends on try-and error mechanism, while them main method return 0 this code call ./stack operation. When the program enters our shell it ./stack never called and our shell injected successfully. In the project part 1, I get the shell. And kernel randomization makes harder to do get root access without any permission with buffer-overflow attack. Each time our target buf target address is change so quickly. And with the not so small probability it increases to 15 – 45 minutes according to computer specs. I tried with my Mac book and finding to right address took 45 minutes, on the other hand with my desktop pc, it took 15 minutes.

Initially, I thought increasing offset which we found with GNU-GDB which is 56 which is minimum number we can assign as an offset of the code, but it did not change anything. I thought increasing offset increase the memory allocation on the stack. Bu it may not related or not.

Task 2 result shown below.



Project Task 3

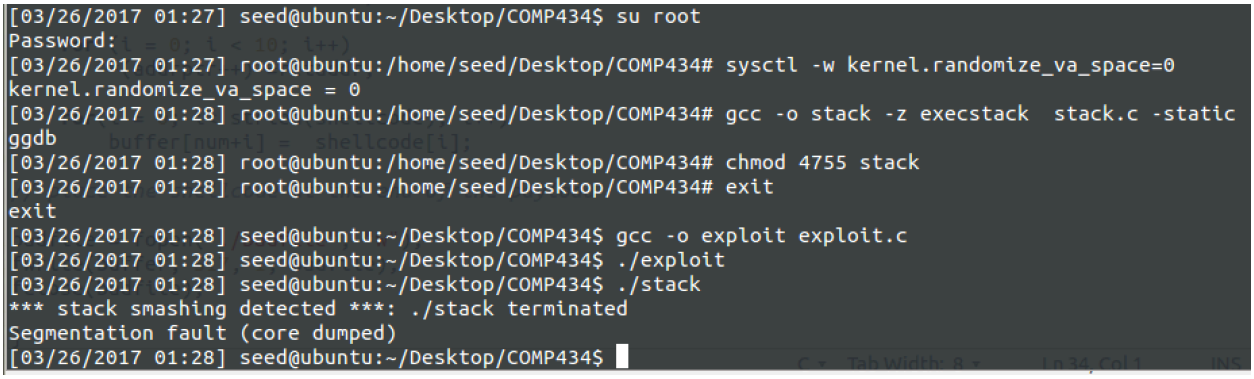


Figure 6: Opening stack protection result.

In this part we reopen stack shield and as a result we unable to access as root via buffer-overflow attack.

As a result you can see buffer-overflow attack no successful.

Conclusion

In this project I learnt basically how to real buffer-overflow works in real environment, I mean experimentally. Buffer overflows are worthy of this level of analysis because they represent a majority of the vulnerabilities, and a significant majority of the problems with the penetration of vulnerabilities. The results of this analysis show that a combination of the StackGuard defense and the non-executable stack defense serves to defeat many contemporary buffer overflow attacks [7]. In our case we unable to access root access reason of the StackGuard.

References

[1] Buffer Overflow Attack. (n.d.). Retrieved March 25, 2017, from http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html

[2] Comp 434/534 - *Project 2: Buffer Overflow Attack*, Koç University, comp 434 instructor and corresponded teaching assistants. (2017).

[3] Nelson, R. P. (1988). 80386: the 80386 book. Microsoft Press. Retrieved from http://www.hep.wisc.edu/~pinghc/x86AssmTutorial.htm

[4] Using the GNU Compiler Collection (GCC): Return Address. (n.d.). Retrieved March 26, 2017, from http://gcc.gnu.org/onlinedocs/gcc/Return-Address.html

[5] A step-by-step on the computer buffer overflow vulnerability tutorials on Intel x86 processor and C standard function call. (n.d.). Retrieved March 26, 2017, from http://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html

[6] Smashing the Stack for Fun and Profit by Aleph One. (n.d.). Retrieved March 26, 2017, from http://insecure.org/stf/smashstack.html

[7] Crispin Cowan, Perry Wagle, C. P., & Steve Beattie, and J. W. (n.d.). Attacks and Defenses for the Vulnerability ofthe Decade*. *Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology*. Retrieved from https://web.archive.org/web/20130309083252/http://tmp-www.cpe.ku.ac.th/~mcs/courses/2005_02/214573/papers/buffer_overflows.pdf

EXTRA: This project was brain-stormed with Mert Kıray who also takes this course. We discussed project stack architecture part together. And for the better understanding doing research on the web together.

Appendix (exploit.c)

```
int count, addressLength;

char *pointer;

long *addrpointer;

long return_address;

pointer = buffer;

int num = sizeof(buffer) - (sizeof(shellcode) + 1);

addrpointer = (long*) (pointer);

// return_address were founded from GNU. GDB.

// +56 is the optimization requirements...

// Check more info from report...

return_address = 0xbffff138 + 56;

// Above code >> This adress may optimized for fater result in adress_ranominazation...

address_Length = strlen(return_address); // Adress length to copy...

// Return adress size is 10 bit.

for (count = 0; count < address_Length; count++) // Return adress load...
```

```
*(addrpointer++) =return_address;
// We add shell code at end of the code, for the increasing hit rate.
for(count = 0; count < strlen(shellcode); count++) // Shell load
buffer[num+count] = shellcode[count];
```