

Computer Security - Midterm Assignment

Bianca Caissotti di Chiusano

i6245461

Task: Assume you have only(!) the binary of the attached program. Your assignment is to write a program that exploits the program, retrieves the secret message and decodes it. Good luck!

Part 1: Retrieving the secret message

The process of retrieving the secret message was done through a **Buffer overflow attack**. As a Mac user, the **Virtual Machine** and Linux distribution “Ubuntu” was used to simulate the process. Because we are using Ubuntu, it is important to first **disable the ASLR** (Address Space Layout Randomization), which otherwise would temporarily overwrite fixed memory addresses when executing the program.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

There are three segments: Stack, Heap and Code. The *main* function, along with *get_input* and *get_decoded_message* (as they are being called by the main function) are in the stack segment. Both *get_input* and *get_decoded_message* have return addresses to go back to the main function. These return addresses point to somewhere in the code segment.

In this segment, there is also *print_secret_message*, which is a method that exists but is not being called anywhere. This feature can be exploited by instead of jumping back from *get_input* to *main*, we jump to *get_secret_message*.

Thus, after disabling ASLR, we install gcc to run code in C:

```
sudo install gcc
```

Then we invoke the compiler with the following call:

```
gcc -fno-stack-protector -z execstack -g assignment.c -o assignment
```

This is called “Stack Smashing”, and it disables the stack protector, which will allow us to write in the stack in parts in which we would not be able to write otherwise. Through this call, we are also compiling *assignment.c* file and naming it “assignment”.

I then wrote:

```
gdb assignment -tui
```

At this point, as shown in the tutorial, because we are on Linux (Windows, for example, handles this differently) there could be two ways of proceeding, we could look for the memory address of *print_secret_message* with the assembler code, so the executable, or we can look at the memory address of our desired function at runtime. The latter is what will give us the correct memory address because the running process is handled differently than the executable.

Thus we need to initially set a breakpoint in the first line of the main function, and then start the program:

```
break main
start
```

And lastly, ask for the memory address of *print_secret_message* at runtime:

```
info address print_secret_message
```

When this is called the following memory address is given: `0x555555552db`

I replaced the XXXX on the given *exploit.c* file, which is also going to be used to perform the attack.

```
unsigned long long EIP = 0x555555552db;
```

As it can be seen, from the source code of *assignment.c* as well, the compiler already knows that memory is requested for 256 characters (1 character is 1 byte of space) and it will be placed in the stack.

```
char buf[256];
gets(buf); //unsafe function call
```

Thus as we are working in a 64 bit system, so 8 bytes, we need to write 256 xs buffer variable and another 8 to overwrite the pointer that points to the return address. Thus in total with the *exploit.c* we fill the input with 264 xs and feed it the found address to run the *print_secret_message* function which will give us the secret message.

```
// 256 Byte buffer + 8 byte stack base pointer (64 Bit system!)
for(i = 0; i < 264; i++)
    fwrite(&x, 1, 1, stdout);
```

So compile and run the exploit code along side the assignment code:

```
gcc exploit.c -o exploit
./exploit | ./assignment
```

Which will return the desired secret message:

```
OLZLJYLATLZZHNLPZHNYLLHISLULZZLZ
```

Part 2: Decoding the secret message

Now that I have the secret message, I will first perform **Frequency Analysis** and then use the **Ceasar Cipher**, which is a monoalphabetic substitution cipher.

In the folder provided for this assignment, are also two Java files. I implemented code to perform the two tasks just described.

When you compile and run Decode.java:

```
javac Decode.java
java Decode
```

You are asked to enter the encoded secret message.

The frequency analysis method returns the most frequent letter of the string, using recursion. In this case, it was the letter **L**.

Because we know that this message will be in English, we first try and substitute the most frequently occurred letter, so L, with the letter **E**, as this is the most commonly used letter in the English language.

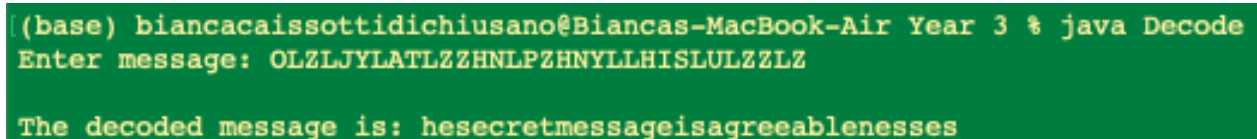
In the standard English alphabet starting with A and ending with Z, the letter E is in the 5th position. I hence, create a new alphabet that follows the same order, but with different positions, such that the most frequent letter from the frequency analysis, is now in the 5th position. Thus, for this assignment, now there is a new alphabet that starts with H and ends with G, with L being in the 5th position.

Standard alphabet, compared to the “new” alphabet:

A, B, C, D, **E**, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

H, I, J, K, **L**, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, A, B, C, D, E, F, G

By replacing the rest of the letters in the secret message from the standard alphabet to the new alphabet, we get the decoded message: **hesecretmessageisagreeablenesses**



```
(base) biancacaissottidichiusano@Biancas-MacBook-Air Year 3 % java Decode
Enter message: OLZLJYLATLZZHNLPZHNYLLHISLULZZLZ

The decoded message is: hesecretmessageisagreeablenesses
```

(Picture from my terminal)