CS253: Software Developments and Operations
Assignment 4 Solutions

Yatharth Goswami
Roll No. 191178

March 22, 2021

I Details of the Weakness

In this assignment we were asked to choose a bug and build a program to demonstrate the weakness. For this assignment, I chose to work on **CWE-190** or *Integer Overflow and wrap-arounds*. I also combined this exploit with **CWE-78** or *OS Command Injection*. So, I will put my understanding of both the attacks here.

1. **Integer Overflows:** These occur normally due to the fact that the integer variables declared in the programming languages are bounded to certain number of bytes and if an integer cannot be expressed in those many bits/bytes, then an overflow is said to occur. Generally, size of an integer depends on how the compiler interprets the 'int' construct in the language. For **C** language, this value is 32 bits. So for C lang, the maximum size of a signed int is 2147483647 and that of an unsigned int is 4294967295.

   Sometimes, it may happen that the code logic assumes the value of an integer to become more and more and such a program, not taking into account the integer limit check may be subject to unexpected errors on providing the right set of inputs. It may happen that an integer grows too large and exceeds the bounds set for the representation. As a result, it's value may wrap around to become a negative number and can cause the code to behave unexpectedly. This can allow an attacker to bypass some security checks, create buffer overflows and since integers are commonly used to index arrays, this can lead to kinds of errors like segfaults and also causes possibility of accessing and overwriting restricted or unknown regions in memory.

   Hence, these types of attacks can become critical in case when the targeted integer is used for control looping (may trigger infinite loops), checking for a security decision, or determining the size in memory allocation, copying etc.

2. **OS Command Injection:** This type of attack takes place when the developed software taken in an external input and using it to construct a part of or complete OS command but it doesn't properly take care of sanitising the input completely and this could lead to attacker using specially crafted attacks to surf on the system or execute dangerous commands on the remote system.

   This type of attack is mostly seen when the attacker does not have direct access to the operating system, such as a web application or some code running on a remote server. The problem becomes even severe, if the compromised program is not running with low privileges in which case the attacker can

run commands with higher privileges and can cause more havoc. The OS command injection can occur in two ways.

(a) In one case, the application may ask the user to just provide an argument for the command it is going to run. For example, the application might want to run (ls $\$path$) and asks the user to provide the path. In this case, attacker cannot stop 'ls' from executing but can provide additional commands using separators.

(b) In this case, complete command is provided by the attacker. The command is then sent to the Operating System to be run using commands like 'exec'.

II Attack Model

The attack model used for exploiting the program is pretty simple in this case. The vulnerable program is built in C++ language and would just ask for the user to give appropriate inputs and expose the weaknesses.

There is no specific OS requirement as such and the exploits would run on any Linux Based OS. The original attack was built and performed by me on an Ubuntu 18.04 machine though. The libraries used are just the standard C++ libraries and hence any external library should not be needed in this case. I recommend using the GNU-g++ compiler for the compilation of the source program. The user would be provided with the source code and the binary and would be asked to provide the right inputs through stdin to break the code and get the secret stored somewhere on the server. Assume that the provided source code is running on the remote server and it is only through this, that user can inject commands, that he wants to run on the main operating system/server.

III How to expose the exploit

The given program is viable to two serious issues, which I will be exploiting to get the secret text. Though the program looks seemingly fine, there are clever ways to bypass the checks and get to the required text. I have provided some of the inputs to test on in the files 'Test1.in' and 'Test2.in' containing respectively the number of monsters to fight with and the command to perform in the end. The script provided alongside 'automate.sh' will run the tests inside the test input files on the program and stores the output of the respective tests in a 'outputs' directory. You can compare these with the outputs in 'expected_outputs' directory which is

the result of the inputs on my machine. The first set of inputs in the file is the one that allows you to read the secret message. You can also apply the exploit manually using this test case

```
(2222222, `echo '.. dc' | rev`; `echo 'txt.terces tac' | rev`)
```

For checking the mitigated program, you can run similar script stored in it's directory. The only difference is that now you will be asked whether you want to check for OS Command injection as well or not and hence corresponding to these two conditions, the expected outputs are also present in two directories. It also creates two directories with the outputs of whether to check for OS Command Injection or not which can then be compared with the expected outputs. The 'expected_outputs' and 'expected_outputs1' folder respectively store the outputs when not considering OS Command injection and when considering it.

IV Explanation of exploit

On reading the source code on the first pass without thinking much about the vulnerabilities associated to it, it seems that on choosing to fight with monsters you will lose some of the bounty points you had initially and for getting to know the secret you are in requirement of more points than you had initially. So, you are required to increase your number of bounty points in one way or the other.

The first thought that immediately comes to mind is what happens if I fight negative number of monsters? Yes, it should subtract a negative number from your current points and hence increase your bounty points. But, the source code checks for the number supplied to be positive and hence this attack will not work. The other option is to exploit the range of int construct in C++ language, we use a very large number of fights such that the value of $1000*$(number of fights) will exceed the assigned range of 'int' and hence wraps around and becomes a negative value, we can use this to actually create negative value after checking for positive value of supplied number of fights. Let's see what actually happens on providing $x = 2222222$ as an input.

On multiplying 1000 with x, the number easily exceeds the given range for signed integers. In this case a wrap-around is experienced and can be explained using modular arithmetic. First the large number is taken modulo (2^{31}) for signed integers and then we add the minimum negative number (-2^{31}) to this modulo

obtained to obtain the final number.

$$
\begin{aligned}
x &= 2222222 \\
x \ mod(2^{31}) &= 74738352 \\
y &= x \ mod(2^{31}) \\
z &= -2^{31} + y = -2072745296 \\
1000 - z &= 2072746296
\end{aligned}
$$

$1000 - z$ is exactly the coins you will end up in the end. Hence, we broke this part using integer overflows. For the rest of the part, you had to traverse one step behind in the directory to get the contents of the file *script.txt*. Since, the commands *ls, cat* and *cd* are blacklisted by the source code, we cannot use them directly. However, we can do a clever trick to bypass this. We can use ‘echo‘ command together with ‘rev‘ command in linux for reversing the command we want to execute. In this way, the command we used will not contain the blacklisted commands and we can easily traverse the directory or traverse the directory. Using this methodology, I prepared the command

```
`echo '.. dc' | rev`; `echo 'txt.terces tac' | rev`
```

which changes to

```
`cd ..`; `cat secret.txt`
```

V Mitigation

These attacks can be easily mitigated by being a little more careful and applying extra checks and neutralising the commands input properly. For mitigating the integer overflow vulnerability we can apply extra checks while setting the bounty points as the value to be subtracted should be positive or a check like setting a defined range for maximum number of fights with the monsters will also work to stop this overflow based exploit.

The exploit based on OS Command Injection can be mitigated by blacklisting some other commands and special symbols like {echo, rev, &, |, ;, ’, ", backticks, newline}. Though this does not provide the most effective security still and a clever attacker might still come up with attacks for this which I may not be able to think of right now, but it atleast prevents the quite obvious attacks possible. In general, it is suggested to never call out to OS commands from the application and instead implement the required functionality using safer platform APIs.