CS-253 Assignment 4-Report

Soham Ghosal
Roll-180771

Weakness: CWE-190: Integer Overflow or Wraparound

- **Details of the weakness as I understand it:**

Quoting from cwe.mitre.org,"The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control."
I have tried to exploit this very weakness.
Let's try to analyse this weakness in a bit more detailed fashion. First, we define the term overflow.
Overflow- Phenomenon in which operations on two or more numbers/strings/ any entity exceeds the maximum(or minimum) value the data type can have. For this assignment, I have used C++. In C++, we have a bunch of data types, which are meant to contain integers. Some of these include short,int, long long. Short is 2 bytes, int is 4 bytes, whereas long long is 8 bytes.
From the knowledge of the binary number system, we know that a 2 byte signed (2's complement) integer will have a maximum range of [-2^(15)=-32768 to 2^(15)-1=32767]. Thus, whenever our result exceeds 32767 or goes below -32768, we say that overflow has occurred. (Note that such an overflow is only with respect to the "short" data type.)

The main arithmetic operations which might generate an overflow are:

1) Addition
2) Subtraction
3) Multiplication

In this assignment, I have used addition to forcefully generate an overflow and thus exploit this weakness.

Coming to wraparound, let's say we have the value 32767 stored in a short variable. When we add 1 to this, we expect 32768, but of course, overflow occurs. Now, by default, instead of catching this overflow, the short data type wraps around, that is it goes to the lowest possible value that could be stored in it, ie. -32768. This is called wraparound. Ofcourse, the g++ compiler has certain flags, which can catch wraparound/overflow{-fwrapv and -ftrapv}, but I have not used those flags here, so wraparound does occur.

- **Details regarding the attack model:**

Operating System: Any Unix based Operating System should work fine.

Compiler: g++
Buildsystem: Make
Others: Bash shell
Library: bits/stdc++.h , unistd.h

I have a bash script "run.sh", which compiles and runs the c++ file "code.cpp" with the required parameters. To run all these, I simply need to "make all". Here, the makefile comes in handy.

What I have tried to build is some kind of a basic submission portal. In the background, we have the main logic in the "code.cpp", which is obviously not available to the students. When a student wants to submit, he has to compile the program, and run the executable along with two other parameters, his name and his roll number. Also, another important rule is that a particular student can update his submission at most 3 times.

To keep track of how many times a student has updated his submission, I use <map> from c++ STL. Each student's unique roll number is mapped to an integer, which is the number of updates he has already done.

The student is supposed to run the code in the following fashion, with exactly 2 parameters

```
./code.out Soham 180771
```

But, it turns out no check is being done in the main code to limit the number of parameters, which is the basis for the exploit. Thus, if a student sends in a lot of random parameters, the counter corresponding to his name will keep on increasing, and eventually it will wrap around. A time will come when the counter reaches 0 again. Thus, the student will be able to update his submission yet again, which is undesirable.
This is the attack model, in short.

- **How to run the vulnerable system to expose the exploit**:

Instead of running the executable using ./code.out Soham 180771, we can send in multiple copies of 180771. The exploit lies in the fact that each instance of 180771 will be treated as a new update, and the counter corresponding to "180771" will keep on increasing, eventually overflowing.
Now since short has a maximum capacity of 32767,if we want our counter to reach 0, ie. cover a full circle, we have to increment it 65536 times. This is because, just after 32767, it wraps around to -32768. Thus, to expose the exploit, we write a bash script, in which we create a string having 65537 copies of "180771", each of which will be treated as individual updates. Thus we have one big string, which we send as the second parameter to the executable.
The main method of "code.cpp" receives these parameters as inputs from the script. When the entire code runs, we find out that we can update the submission more than 3 times. The exact value depends on the number of copies of the roll number attached to the string in the script.

- **Explain how your exploit works in the accompanying program:**

While running the executable, we pass some parameters. For example, one way to run it could be:

./code.out Soham 180771 Dummy 000000
This means that "Soham" with roll number 180771 has submitted once, and "Dummy" with roll number 000000 has submitted once.
./code.out Soham 180771 Dummy 000000 Soham 180771
This means "Soham" has submitted twice, while dummy has submitted once.

Expected Behaviour: A student can only submit at most 3 times. Thus,
./code.out Soham 180771 Soham 180771 Soham 180771 Dummy 000000 Soham 180771
Produces the output "Submission Updated for Roll 180771" only 3 times. The 4th input "Soham 180771" is ignored.

Actual Behaviour: The number of parameters after ./code.out could be infinite in number, which is what allows the exploit to be used. If we put in more than 65535 "Soham 180771" after ./code.out, the counter corresponding to 180771 will keep on incrementing, and since the counter type is signed short, it will overflow and wraparound.
Thus, a student can kind of use this hack to send a lot of fake update requests, and eventually causing overflow in the counter.

Let's say we have 65537 instances of "Soham 180771". After 32767 instances, mp["180771"]=32767. Then after the 32768th instance, mp["180771"]=-32768. Therefore after the 65536th instance, mp["180771"] = 0, and after the 65537th instance, mp["180771"]=1. The program sees no difference between mp["180771"]=1 when the counter was 1, and mp["180771"]=1 when the counter was 65537.
This is th eexploit.

- **Explain how to mitigate this weakness, ie. exactly pin-point what was wrong in the program and what is the way the program should have been written.**

There are quite a few ways to mitigate the weakness. I will enumerate some possibilities below:

1. Considering this to be a small scale submission portal, it might be implemented in some school, where the number of students is fixed. So, let's say we have 100 students. Each student will have at most 3 submissions. That's a total of 300 submissions,or 600 parameters.We also have 1 parameter--the program name. So, the maximum number of parameters main() has to accept can be 601. So if we put some kind of a check, we might prevent this exploit.Some possible checks are:
    - if(argc>601) return 0; //if number of parameters are greater than 601, return.
    - assert(argc<=601); //execute only if this is true.

    Here, the drawback is the fact that we need to know the maximum number of students beforehand.
    If the number of students exceed this maximum, the program will never run.

2. We might change the type of the counter variable. Thus instead of defining map as map<string,short>, we might define it as map<string,int>

Here, since int has a larger range, overflow might still occur, but it will occur at a very large value. The number of parameters to main() is upper bounded by the maximum value of a signed integer, so this implementation might work fine as well, mitigating the vulnerability.
The disadvantage of such an implementation is that a lot of memory is wasted, since the number of students is small, the maximum number of parameters will be small. So there is the vulnerability vs memory trade off here.

3. In Line 26 of "code.cpp", we have written mp[argv[pos]]++, without any checks. So, it increases the counter corresponding to argv[pos]++, everytime one of the parameters in main() is argv[pos]. What we might do to change it is:

```
if(mp[argv[pos]]<=3) mp[argv[pos]]++
```

What this does is that it ensures that the maximum value that mp[argv[pos]] could take is 4,
thus ensuring that there is no overflow.

- **Some other prospects of weakness:**
  Line 21 has a character array of size 12.
  If the "date" command can somehow be changed(code injection), buffer overflow might occur, and the attacker may be able to run any command on the system.

References:

1. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
2. https://samate.nist.gov/SRD/index.php
3. https://stackoverflow.com/
4. https://handouts.secappdev.org/handouts/2011/Frank%20Piessens/C-vulnerabilities-slides.pdf
5. http://www.jsums.edu/nmeghanathan/files/2015/05/CSC438-Sp2014-Module-6-Software-Security-Attacks.pdf?x61976