CWE-307 : Improper Restriction of Excessive Authentication Attempts. ;CWE-78 : OS Code Injection; CWE-259,798: Use of Hard-coded Password

Run " bash automated.sh 2> warning.txt" to run all test cases and all files together automatically!

Ever since the CS253 midsem, I have been hooked to authentication bypassing :p . And after Seeing this whole branch of improper access control of CWEs I was intrigued.

As far I can recall, the reason it was so easy to crack password in the midsem was that there was no authentication limit on the executable, so brute force attack exploited this little fact to its advantage and cracked that executable.

So, Developers while making websites, design the login bar in such a way that it is difficult for the attacker or some bad person to get the access and bypass it.

And while making this system, they sometimes choose a **improper path** And this is what **CWE-307** is all about, it is the weakness in which developer fails to include sufficient measure to stop the bad guys from bypassing authentication. There mistakes include:

1) Not having any check at all on the authentication process.(like in our midsem)
2) Having a improper method, such as timout, sleep for some time after every incorrect login.

What I have exploited in the midsem is the 1st point.

And what I will be exploiting in this assignment is the 2nd point that when they use **sleep function** to delay the next try. It becomes really difficult to attack by naive brute force. The reason being, that there are like 10^{no of digits} of passwords and if we wait for 2 seconds for every try. We wont be able to get access by brute force in entire lifetime :p

But, here is the exploit.

We can attack that system, **Parallelly**, from many different computers using the army of bots just like what happens in the DDOS attack. The same way we can make multiple simultaneous queries to the database.

So, we will entually gain access sooner or later depending upon the number of cores or threads we are attacking with.

In my case, I have used just 3 digit number password, which caps my number of passwords to 1000, I have chosen small number just for the illustration purpose and I have used parallel bash scripting in place of using multiple System to do my attack.

I Distributed all the probable passwords in 9 files, which 9 scripts are using each to do its own check in parallel with all other. This way, I increased the speed by 9x.

For example:

Before: doing the brute force naively, took 1000*2 seconds ~ 1 hrs.

But, using this method with just 9 scripts it takes less than 2 min.

Thats the vulnerability induced due to improper restriction of excessive authentication attempts.

This was the first vulnerability in my program. 2nd vulnerability is Classic Code injection. I have watched n number of videos on Sql injection, Code injection, etc. and Really wanted to implement in my program, So, I tried to just touch it.

In **CWE-78** Developer doesn't take into account that his code can be violated and modified through the use of command line arguments.

Like In my Case, The Program was intended just to greet the user, but due the fact that user has to input his name into the command line. I exploited this fact, by the use of piping or semicolon separated

argument, the attackers injects his own code into the argument to his own advantage, as in my example, I have outputted the sensitive information regarding the directory I am in, or I could also do something more harmful like destroying ~/.bashrc, or shutting off the console .

Shell feature	`USER_INPUT` value	Resulting shell command
Sequential execution	`; malicious_command`	`/bin/funnytext ; malicious_command`
Pipelines	`| malicious_command`	`/bin/funnytext | malicious_command`
Command substitution	`` `malicious_command` ``	`` /bin/funnytext `malicious_command` ``
Command substitution	`$(malicious_command)`	`/bin/funnytext $(malicious_command)`
AND list	`&& malicious_command`	`/bin/funnytext && malicious_command`
OR list	`|| malicious_command`	`/bin/funnytext || malicious_command`
Output redirection	`> ~/.bashrc`	`/bin/funnytext > ~/.bashrc`
Input redirection	`< ~/.bashrc`	`/bin/funnytext < ~/.bashrc`

This Picture from wikipedia summarises the shell code injection in short.

CWE-259: Another Vulnerability Which our program has is of Hard-Coded Passwords/credential.

Is is a extremely bad practice to use the hard coded passwords in the system. As the attacker can easily Dissamble the byte code or assembly code to make out the constant and finally see the password there itself.

For example : I have provided another program in C language. In which I have used a hardcoded password to get access to the ROOT.

But, If Someone has the access to even just the binary of that Code, he can get back the password of that file just by doing “strings a.out” or something similar which deassembles the code, and gives the high level feeling that what all would have been present in the source code.

CWE-307 is language and machine independent, cause it is the vulnerability in the design of the system. I have used multithreading/parallel computing to do Distributed brute force attack. Which I believe is pretty much avaible for every default configuration.

CWE-78 the example that I took works on bash, tscp, sql(ultra famous), etc. There aren't any anything extra required than these software being installed in our system.

In my example case: just normal bash will do.

CWE- 259: This is interesting in the sense, that It was hard for me to get the byte - code of python using pypy and then converting it into C and then doing gcc , then strings, it was a long process. I believe that this method is most vulnerable to simple languages like C, C++,java , etc.

Because in these languages dissembler is easily available and we can exploit hard-coded credentials just by converting or disassebling the byte code in .C example .

For 307, run the system parallely to expose the exploit. Running it with Fuzz tester can expose the exploit.

For 78, using piping or semicolon will expose the exploit or critical information.

For 259, run the "strings a.out" to expose the vulnerability and look at password without opening the source code.

For 307, can be mitigated by just adding the "static" variable which keeps track on number of attempts a single user is having. Thus logging off or blocking his id after reaching those many attempts.

The Approach to prevent excessive authentication attempt was improper, so we need to modify it to mitigate this error.

For 78, The problem was that we trusted the input without even validating it once, or having certain bounds on the type of input that we

can give, there many-many mitigations of code-injections given on mitre.org.

One of easiest ways to mitigate this error, is to make a list of allowable commands which user can give as a argument to the system, this way, attacker won't be able to abuse the system using the power of code injection, and must to do input validation before processing that input.

For 259, It is pretty obvious that the blind error which the developer made was using the hard-coded password ,which is always a terrible idea, People don't want to store people passwords in the database, because, it makes it prone to attackers, who will find some vulnerability to exploit them.

Clear Cut way is to not use the hard-coded password and use some random hash function, or store the password somewhere else and basically don't leave it open for anyone to dissemable and see it.

Thank you for giving me this opportunity, I really learned a lot. And Thanks for giving so much time in going through my report and submission.