Software Vulnerabilities

Written by Lewis Impey

**Missing Authentication for Critical Function**

This security vulnerability describes cases where the software does not perform any validation of the user’s identity before allowing access to privileged application functionality e.g. allow a user to add, modify or delete records without validating whether they are an admin or not.

The potential impact of this security vulnerability can depend on what functionality the software has, and can vary from trivial to severe. In the most severe of cases, its possible that private information could be disclosed or the entire application could be compromised.

To mitigated this software vulnerability, it is important that all functionality that uses privileged assets or information is identified. There should also be very clear roles defined for use within the application (such as admin and user) which have specific permissions given and withheld from them. It is also important to constantly check whether a user has access to the privileged information or assets that they are accessing.

**Weak Passwords**

This security vulnerability is rather straightforward, but a weak password is a password that is usually short, common, a system default (password is password) or something that could be very easily guessed with brute force attacks. Also passwords that could be guessed using a subnet that contains words from a dictionary, names, words based on the username, etc.

Since a password is a key to your computer (or another device/application) that could possibly contain very private information, it is important that a strong enough password is set so that it could not be easily guessed by someone trying to get into your account or device with malicious intent.

This is a very easy software vulnerability to fix, as it just entails changing a weak password to a stronger, more verbose password that is more difficult to guess. Below are some characteristics of a strong password that should be considered when creating a password:

* Contain both upper- and lowercase characters (e.g., a–z, A–Z).
* Have digits, punctuation characters, and letters (e.g., 0-9! (e.g., 0-9! @#$%^&\*()\_+|∼−=\′{}[]:“;'<>?,./).
* Are at least eight alphanumeric characters long.
* Not a word in any language, slang, dialect, jargon, etc.
* Are not based on personal information, names of family, etc.

Written by Shantanu Talukder

**File Upload Vulnerability**

This vulnerability comes in two forms-

1. Local file upload vulnerability- When a malicious script is being uploaded by the user directly and executed.

One way this can happen is when there are no restrictions on the users who can upload the files. This can easily be fixed by adding authentication checks via restricting who can upload (eg- log in/sign ups) and authorization checks via checking what files are being uploaded.

Another way this can happen is when a file name/contents are not sanitized and the non-sanitized file can contain an extension which the attacker can use to execute a code accessed via web. This can be avoided with application which has features that allows it to sanitize files before uploading.

1. Remote file upload vulnerability- is when the users provide a site URL which is used to save a file from that site locally on a disk. If the file is malicious is then used by the attacker to cause harm.

This can be prevented by

1. Checking what kind of file is being uploaded.
2. The allowance of only specific types of file extensions
3. Only specific users being able to access this form of file upload.
4. Storing files in a directory that is only accessible to authorized users.
5. Making stored files non executable.

**Missing Authorization Vulnerability**

This takes place when an application does not do an authorization check when a user tries to use its resources or use it to do something. This allows users who should not have the privilege to access certain data or actions to do so.

This can create a whole host of problems like leaking of sensitive data, modification data by the attacker, attackers to execute codes on a targeted device, denial of service to users and attacker could grant themselves privileges by modifying critical data.

This can be prevented by having the architecture and design of the software being reviewed, alongside other detection methods.

SQL Injection

Written by Jacob Beynon

**What is SQL Injection?**

SQL injection is a web-based vulnerability which allows the user to submit queries which interfaces with the data base. This grants access to information that is otherwise hidden to the user. This includes data which could belong to users or data which that application has access to.

If SQL injection is successful, then the user could remove or edit entries within the database itself. Most attacks recently have been due to SQL injection resulting in some major damage and even fines. If the SQL injection is not resolved and fixed, then it could be used as a backdoor into the server which the database is on and could cause major damage.

**How to SQL Inject**

Injecting SQL into a program is very easy. Say we have a program where we can search through the data base for a specific user, this is how we would do it:  
txtUsrId = getRequestString(“UsrId”)

searchSql = “SELECT \* FROM Users WHERE UserId = ” + txtUsrId;

This command allows us to enter in a user ID and it will search everything inside the Users table until it finds the given user. If instead of entering in the user ID of a user e.g., “Usr30005218”, we enter in a quire such as “5214 OR 1=1” This will then make the statement:

“SELECT \* FROM Users WHERE UserId = 5214 OR 1=1”

As 1=1 is true this results in every user being displayed within the program with all their private information. This is however the most basic way of SQL injection.

Another way SQL injection can occur is via the “=” operand. This occurs if you have a login screen where the user needs to enter in their username and password. If you enter in your details into the representative textboxes this is what the code would look like:  
UsrName = getRequestString(“Tb\_Username”);

UsrPassword = getRequestString(“Tb\_Password”);

“SELECT \* FROM Users WHERE Name= +”UsrName” + AND Pass=+ ”UsrPass”+”

Normally entering in your details here would either grant you access to your account or display an error message, but if you instead include the “=” operand then just like the example above all the account details will be displayed to the user.

Both problems we will be facing within our program and will need to create some code to prevent this from occurring.

**How to prevent SQL injection**

Prevent a SQL injection attack is rather simple to do. It can be as easy as writing a few lines of code to applying additional programs to work all together: Here are some ways that we will be trying to reduce SQL attacks on our program.

One way we can reduce SQL injections attack sis reducing the amount of access points. When constructing our program, we should only have the amount of SQL interactions needed for our program to run. If we have to many then this could be a sign of weakness and could be abused. We could also reduce the number of text boxes for user input so that the user has less limited area to attack.

Using a variety of hard coding and dynamic SQL commands reduces the about of possible SQL injection attacks. The main area for SQL injection is via the user input so if we reduce the need for user input and have pre-generated queries, this results in less ways off attacks. Of cause the program could not run properly if everything is hard coded so having a variety of both is a good procedure.

One of the main ways is having code which catches the SQL attack attempt. This can be done by many ways. As we are using SQLite it comes with a pre generated function called “string sqlite\_escape\_string()”. This function will escape any query being made if it detects any characters which are deemed special via SQLite.

**Buffer Overflow**

Written by Ryan Murunga

Data received can maliciously or unintentionally overrun a fixed capacity memory space and overwrite or corrupt data in adjacent spaces. This occurrence is termed as a buffer overflow.

Buffers are memory spaces that temporarily hold data while being transferred. They are vital in cases where the rate data is received is different from the rate data is processed. An example of a buffer overflow attack is when on a login interface page, the password input can be set to expect an 8-byte password, but instead when the user enters a 10-byte password, the excess data can overrun the adjacent memory spaces. A buffer overflow attack can cause system crashes, memory access errors or even make the program perform unauthorized tasks.

Among the various attacks of this nature, the commonly known ones are:

* Stack based buffer overflow, which occurs when a buffer located on the stack has more data written into it.
* Heap based buffer overflow, which occurs when the memory space is flooded having the application overwrite internal structures as a result.

Attackers use buffer overflows to alter the execution path of applications. When a carefully crafted malicious code is sent to the application and executed, the attacker can take over the application and cause harm.

C and C++ are programming languages that are highly susceptible to this kind of attack as they lack safeguarding techniques that guard against overwriting or data access in their memory. Programming languages that are relatively less susceptible to buffer overflow attacks are Java, Python and C# just to name a few.

**How to prevent/mitigate a buffer overflow attack**

All is not lost as you can protect your program against buffer overflow attacks and one way of doing so is address space randomization. This is randomly moving around the address space locations of data regions making it virtually impossible for the buffer overflow attack to happen for it depends on the locations of the executable code.

Data execution prevention, flagging certain memory areas as non-executable or executable is another data protection method. Furthermore, bounds checking can be applied at runtime, automatically checking that data written to a buffer is within acceptable boundaries.

A structured exception handler overwrite protection can be applied to assist in stopping harmful code from manipulating the Structured Exception Handling, a built in mechanism for managing software and hardware exceptions.

# References

*Buffer Overflow Attack*. (n.d.). Retrieved 01 28, 2021, from Imperva: https://www.imperva.com/learn/application-security/buffer-overflow/

Christian B. Lahti, R. P. (2005). *Sarbanes-Oxley IT Compliance Using COBIT and Open Source Tools*. Retrieved from ScienceDirect: https://doi.org/10.1016/B978-159749036-8/50006-9.

CyberSecurityTV. (2021, January 13). *File Upload Content-Type Validation Bypass | Web App Penetration E02*. Retrieved from Youtube: https://www.youtube.com/watch?v=v4gx9vTvmt4

*How to Prevent File Upload Vulnerabilities*. (2021, January 13). Retrieved from WordFence: https://www.wordfence.com/learn/how-to-prevent-file-upload-vulnerabilities/#:~:text=A%20local%20file%20upload%20vulnerability,Internet%20and%20store%20it%20locally.

Ltd., T. P. (2021, January 13). *Penetration Testing - File Upload Vulnerability*. Retrieved from Youtube: https://www.youtube.com/watch?v=jFRYPmCulh4

*Missing Authentication*. (n.d.). Retrieved January 31, 2021, from Common Weakness Enumeration: https://cwe.mitre.org/data/definitions/306.html

*Missing Authorzations*. (2021, January 13). Retrieved from Common Weakness Enumerations : https://cwe.mitre.org/data/definitions/862.html

*PortSwigger*. (n.d.). Retrieved 01 26, 2021, from SQL Injection: https://portswigger.net/web-security/sql-injection

Rubens, P. (2018). *How to Prevent SQL Injection Attacks*. Retrieved 01 17, 2021, from Esecurity Planet: https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/

*Software Security*. (n.d.). Retrieved January 31, 2021, from Micro Focus: https://vulncat.fortify.com/en/detail?id=desc.dynamic.xtended\_preview.access\_control\_missing\_authentication

*SQL Injection*. (n.d.). Retrieved 01 26, 2021, from w3schools: https://www.w3schools.com/sql/sql\_injection.asp

*SQLite – Injection*. (n.d.). Retrieved 01 27, 2021, from tutorialspoint: https://www.tutorialspoint.com/sqlite/sqlite\_injection.html

Sudhanshu Chauhan, N. K. (2015). *Weak Password*. Retrieved from ScienceDirect: https://www.sciencedirect.com/topics/computer-science/weak-password

Welekwe, A. (2020). *Buffer overflow vulnerabilities and attacks explained*. Retrieved 01 28, 2021, from Comparitech: https://www.comparitech.com/blog/information-security/buffer-overflow-attacks-vulnerabilities/

*What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks*. (n.d.). Retrieved 01 28, 2021, from Veracode: https://www.veracode.com/security/buffer-overflow