Table of Contents

[Vulnerabilities – An Introduction by Saidhbh O’Malley 2](#_Toc64627238)

[Integer Overflow by Chloe Andronicou 3](#_Toc64627239)

[What is an Integer Overflow? 3](#_Toc64627240)

[How an Integer Overflow Attack Works 3](#_Toc64627241)

[Prevention? 4](#_Toc64627242)

[Missing Data Encryptionby Jack Davies 6](#_Toc64627243)

[What is a Missing Data Encryption Vulnerability? 6](#_Toc64627244)

[How it Creates a Vulnerability 6](#_Toc64627245)

[Prevention? 6](#_Toc64627246)

[Missing Authentication and Authorization by Jenni Whewell 8](#_Toc64627247)

[What is Missing Authentication/Authorisation? 8](#_Toc64627248)

[How can this affect security? 8](#_Toc64627249)

[How Can it Affect Code? 9](#_Toc64627250)

[Prevention? 9](#_Toc64627251)

[Buffer Overflow Attack by Saidhbh O’Malley 10](#_Toc64627252)

[What is a Buffer? 10](#_Toc64627253)

[What is Buffer Overflow? 10](#_Toc64627254)

[How a Buffer Overflow Attack Works 11](#_Toc64627255)

[Prevention? 11](#_Toc64627256)

[OS Command Injection by Brendon Plummer 12](#_Toc64627257)

[What is an OS Command Injection? 12](#_Toc64627258)

[How OS Command Injection Works 12](#_Toc64627259)

[Prevention? 13](#_Toc64627260)

[SQL Injection by Jenni Whewell 14](#_Toc64627261)

[What is SQL Injection? 14](#_Toc64627262)

[How can it affect code? 15](#_Toc64627263)

[Prevention? 15](#_Toc64627264)

[Tainted Data by Chloe Andronicou 17](#_Toc64627265)

[What is Tainted Data? 17](#_Toc64627266)

[The Risk of Tainted Data 17](#_Toc64627267)

[Prevention? 17](#_Toc64627268)

[Unprotected Storage of Credentials by Jack Davies 18](#_Toc64627269)

[What is the Unprotected Storage of Credentials? 18](#_Toc64627270)

[How does it affect the code? 18](#_Toc64627271)

[Prevention? 19](#_Toc64627272)

[References 20](#_Toc64627273)

# Vulnerabilities – An Introduction

As engineers create increasingly complex and elaborate software systems, the need to place emphasis on securing their code-base from unwanted intrusion has also increased proportionately. Yet with the movement towards ‘cloud computing’ as standard and increased layers of code abstraction, unsurprisingly dangerous practices and unsecure products do persist (McNab, 2017, pp. 21).

The internet has evolved dramatically since the early days of the World Wide Web. The websites of 20 to 30 years ago were little more than online information repositories with the browsers being invented later as a means of retrieving and displaying static documents (Stuttard and Pinto, 2011, pp2). Internet-based security threats were largely due to server software vulnerabilities and compromised web servers only provided access to the already-public data stored on that particular repository.

In recent years with the exponential increase of ‘Internet of Things’ development, most devices will now have either a web or mobile component to allow users access across the internet. Identifying vulnerabilities and providing security solutions has become a high priority for security research professionals across the industry (Gupta, 2019). As any device or network with an IoT component can potentially lead to entire systems becoming vulnerable, shutting down these new ‘attack vectors’ has become an industry-wide priority.

According to Hazim et al. (2021) it would seem that a ‘prevention is better than a cure’ approach serves as the best solution in the long term. Rigorous testing and static/hybrid code analysis, along with breakthroughs in areas such as ‘machine learning’ are providing developers and security professionals alike many of the essential tools needed to fight against potential cybercrime.

Outlined in this report are what we consider to be some of the primary categories of software vulnerability. While clearly not an exhaustive list, we feel that our research covers the vast majority of over-arching areas, and describes many of the prominent methodologies and countermeasures in a developer’s ‘arsenal’ in their fight against software security breaches.

# Integer Overflow

(Chloe Andronicou)

## What is an Integer Overflow?

Integer Overflows have become one of the many causes of software vulnerabilities and have played a significant part in threatening the security of the software as well as corrupting memory.

Dietz et al. (2015) mention that integer errors can be exploitable, expensive, and insidious, often leading to fatal errors. They can prove quite difficult to track down since not all occurring overflows have proven to be bugs and for the fact that they reside deep in the program. Therefore, due to their difficulty of being detected, Wang, Song and Lee (2015) mention that there needs to be a way to differentiate malign integer overflows from less harmful overflows. They suggest that benign integer overflows usually appear in software that make use of hashing and random number generators. This makes the detection of harmful overflows even more challenging as these errors usually appear throughout the execution of the program.

## How an Integer Overflow Attack Works

Usually, there is an indirect exploitation of integer overflows that happen across the stack or heap that could potentially result in buffer overflows. The vulnerabilities associated with integer overflows could transpire to several exploits. Some of these include arbitrary execution, logic error, bypass of an upper bound sanitization check, Denial of Service (DoS) and array index attacks that are a result of vulnerable integer value. (Muntean, Grossklags, & Eckert, 2017)

Furthermore, Muntean et al. (2019) allude to more errors that derive from integer overflows:

|  |  |
| --- | --- |
| Vulnerability | Cause |
| Integer coercion error | During bad type casting and extension or truncation of primitive data types. |
| Off-by-one error | During product calculation/usage; an incorrect maximum/minimum value is used which is one more, or one less, than the correct value. |
| Unexpected sign extension | Appears when an operation executed on a number can cause it to be sign-extended when it is converted into a larger data type. |
| Signed to unsigned conversion error | When a signed primitive that is used inside a cast to an unsigned primitive can produce an unexpected result if the value of the signed primitive cannot be represented using an unsigned primitive. |
| Unsigned to signed conversion error | When an unsigned is used inside a cast to a signed primitive, which can produce an unexpected value if the result of the unsigned primitive cannot be represented using a signed primitive. |
| Numeric truncation error | When a primitive is cast to a primitive of a smaller size and data is lost in the conversion. |
| Integer overflow to buffer overflow | When an integer overflow occurs that causes less memory to be allocated than expected, which can lead to a buffer overflow. |

## Prevention?

To possibly resolve some of the vulnerabilities caused, Muntean, Grossklags and Eckert (2017), bring to light some repair solutions and discuss their advantages as well as their disadvantages. Firstly, Manual- Based Input Validation is used to check and repair integer overflows however it does come with its drawbacks. It is easily prone to errors and takes a large amount of time to be inserted into large pieces of code. Additionally, it cannot be applied across numerous integer precisions and it cannot assure that the program’s behaviour is maintained along with that it cannot be certain that the integer overflow error was efficaciously removed.

Another solution mentioned is Compiler- Based Input Validation which can be very easily and quickly inserted and is low-priced, but its job can be improved on by some compilers as they are implementation specific and specify that integer overflows are undefined behaviour. It also shares the same disadvantages with Manual-Based Input Validation by not guaranteeing that the integer overflow bug was removed and by being easily subjected to errors. Nevertheless, being compiler based it has access to more detailed data compared to static tools, which could prove highly advantageous with bug prevention.

Symbolic Execution-Based Input Validation is another prevention method that is more sufficient in repairing errors in comparison to the other techniques stated above. However, it needs to be applied appropriately as it is based on computationally rigorous analysis strategies that might not work well with largely built programs.

Overall, manual source code repairs should only be used sparingly to address errors and should if can, be avoided. Compilers should be evaded when wanting to repair integer overflows as it has a low guarantee rate and specialized tools should be used instead as their guarantee for repair is much higher.

# Missing Data Encryption

(Jack Davies)

## What is a Missing Data Encryption Vulnerability?

The missing data encryption weakness exists due to the lack of correct encryption on important or sensitive data before storage or transmission. If the data is not correctly encrypted, it can’t provide the appropriate confidentiality, integrity and accountability that needs to be properly implemented in the encryption and data. According to Howard, LeBlanc and Viega (2010, pp. 253 – 266) it doesn’t matter what data you’re storing or how it is stored, as long as it is encrypted correctly using a standard and well-tested encryption algorithm. The missing data encryption weakness is introduced during the Architecture & Design, Implication Stages. This vulnerability also allows users to maliciously attack and modify application data (CWE, no date).

In order to encrypt data, the process will take some information like a piece of text or email, and will then encrypt it to an unreadable form, such as cipher text. According to Norton.com (no date), this will protect the confidentiality and the importance of the data as it is transmitted through the network. To decrypt the data, both the sender and receiver will use a secret decryption key that will decipher the text back to its original form.

## How it Creates a Vulnerability

Without data encryption, data that is being transmitted between devices, can be easily accessed by hackers as they attack and modify data. Cybercrime is a global business, and they will try and steal data mainly for financial gain. Which is why encryption is needed when transmitting data through the network.

## Prevention?

#### Symmetric Encryption:

Symmetric Key encryption is when the sender and recipient use only one single key to encrypt and decrypt data. Therefore, the key is shared between the sender and recipient to establish the identity of the clients and servers (Howard, LeBlanc and Viega (2010), pp. 371).

##### Asymmetric Encryption:

Asymmetric Key encryption is when you need two keys to work. A public and private key is required. Firstly, the public key is used to encrypt a piece of data. The public key belongs to the recipient. If the recipient then wishes to decrypt the sender’s message, they will use their own private key to decrypt that data (Kembora, 2020).

Here are some types of encryption that are being used globally (Norton, no date):

* Data Encryption Standard
* Triple DES
* RSA
* Advanced Encryption Standard
* TwoFish

# Missing Authentication and Authorization

(Jenni Whewell)

## What is Missing Authentication/Authorisation?

An authentication attack is a serious type of hacking used by an attacker to compromise an entire infrastructure and gain access for financial gain. The process of authentication is when a program or application requires authorisation to confirm a user’s identity, quite frequently using a password. Hence this can be a serious vulnerability if the password lacks complicity or the proper encryption algorithms aren’t in place (Peikari and Chuvakin, 2004, pp. 301).

An authorisation attack against a computer system aims to access the computer infrastructure without the correct credentials. The difference between authentication and authorization is authentication is how a program or application determines who you are whereas authorization is the limitation of access a person is granted to what they can see or do.

## How can this affect security?

There are a vast variety of different types of attack, but these are the main categories:

Brute Force

Brute force attacks are done by guessing a person’s credentials where the password verifiers are improperly created (Howard, LeBlanc and Viega, 2010). Whether that be the username and password or any other form of personal details, they are acquired through the means of trial and error.

Insufficient Authentication

Having insufficient authentication measures in place means an attacker would be able to access a web site or network containing sensitive data without proper authentication.

Weak password recovery authorisation

Having weak authorisation recovery means that during an attack, an attacker can obtain authentication information illicitly. This can then be changed to lock the victim out of their own accounts. Similarly with forgotten passwords where improper multi-step authentication isn’t present (Howard, LeBlanc and Viega, 2010).

## How Can it Affect Code?

Without authentication and authorisation code can be affected in many ways, one being that it gives the attacker the freedom and power to maliciously alter code. Once inside a network it gives them the ability to change source code whenever they want, technically making it an open-source code. The attack may expose the victim’s personal details through the means of HTTP authentication.

## Prevention?

By having multi-factor authentication means that it checks your credentials against another source. A commonly used example would be a stored phone number to which a code can be sent via text message. The user then has to enter this verification code to establish a connection.

When coding, you can implement authorisation code which could consist of an alphanumeric password. The most common use of authorisation code is used through financial services, they use this when validating a person’s identity in terms of a transaction. It can also be used when the customer enquires about their credit. Assuring you have the authorising code means that an attacker will have to establish multiple connections to the server to gain access which is a lot harder to do if you don’t have it.

# Buffer Overflow Attack

(Saidhbh O’ Malley)

## What is a Buffer?

The Buffer is an area of computer memory that contains data which is stored for a short period of time, usually directly before it is used (TechTerms.com). A buffer helps with computer performance, allowing more efficient access to data on the disk. As interconnected devices are rarely completely in sync the buffer allows sent data to be temporarily stored somewhere while the recipient device prepares to receive it. For a real-world example, if you are streaming a video from the internet the buffer stores an initial percentage of the file so you can begin watching directly from the buffer itself, rather than directly from the internet. This lessens the chance of the file skipping or stalling due to a bad network etc.

## What is Buffer Overflow?

Buffer Overflow occurs when the volume of data in the buffer exceeds the storage capacity of the memory buffer itself, causing the data to overflow into adjacent storage. According to Padmanabhuni and Tan (2011), while writing data to a buffer the program overruns the buffer’s boundary and can write to (or even overwrite) adjacent memory locations (Veracode.com). For example, if the log-in buffer for user credentials can accept username/password inputs up to 8 bytes and an input of 10 bytes occurs, the program can ‘overflow’ and may write the excess 2 bytes beyond the buffer boundary and into a random and unknown location in memory (Imperva.com).

This is quite often caused by improper allocation of space in the buffer or simply malformed inputs, and is usually the result of poorly designed and constructed software (Deckard, Osipov and Bhalla, 2005, pp. 3). This writing outside of the allocated buffer memory block can cause the program to behave quite unpredictably. It can cause all manner of errors from generating incorrect results/returns and memory access errors, to corruption of data and program crashes, or (as outlined below) it can lead to the insertion and execution of malicious code and leave your application wide open to exploitation.

## How a Buffer Overflow Attack Works

An attacker exploits this vulnerability to overwrite application memory, thus triggering a response that damages files and can potentially expose private/confidential data stored outside the buffer. The more common kind of attack is a stack-based vulnerability overflow where an attacker essentially overflows the buffer “on the stack beyond its allocated memory” (Padmanabhuni and Tan, 2011). This can be used to modify return addresses to change the control to a location of their choice, and makes use of an overflow that can only happen during the execution time of a function or program. A heap-based attack, while more difficult to carry out, can involve flooding the memory space allocated for a program beyond runtime operations (Imperva.com).

## Prevention?

Poor coding standards and code errors at the development stage are typically the cause of a Buffer Overflow. The most common of these errors are failing to allocate a large enough buffer, not correctly handling these errors as they occur and not actually checking for overflow problems during the software testing process. According to McNab (2017, pp.27) languages such as C/C++ are particularly vulnerable to these attacks as they do not have any in-built protection against buffer overflows. They lack in-built bounds checking operations and allow overwriting of any part of memory without any checks as to whether the data to be written might exceed its allocation of memory (Padmanabhuni and Tan, 2011) Hence applications written in these languages are quite often the target of an overflow attack. There are numerous methods to detect and prevent the buffer overflowing. Checks can be added to verify every access is within bounds and often there are library tools available for inbuilt bounds checking (Black and Bojanova, 2016). Obviously, prevention is the best solution with early detection being the key and static analysis on the source code or binaries can easily be used in conjunction with a solid testing technique (Black and Bojanova, 2016). Quite often, simply writing your code in a language that incorporates bounds checking (such as Java, Python, Visual Basic etc) which enforces runtime checking, can detect if an overflow has occurred (Kuperman *et al.*,2005).

# OS Command Injection

(Brendon Plummer)

## What is an OS Command Injection?

OS Command Injection is an attack where the goal is allowing arbitrary commands to be executed on the operating system of the server/application. It is made possible when executing user-controlled input, be it through cookies, parameters, HTTP headers, forms etc. Also known as a “Shell Command Injection”, these commands are executed via the system shell of aforementioned server/application, this being a major problem as through the shell the commands are run with the root privileges of the application (Stasinopoulos, Ntantogian and Xenakis, 2015).

## How OS Command Injection Works

In the example below by Zhong (OWASP.org, no date) I believe he highlights the necessity of input validation with any user-controlled input.

Here we see a wrapper around the UNIX command *cat*, where this command simply prints the contents of a given file to standard output:

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*\*argv) {

char cat[] = "cat ";

char \*command;

size\_t commandLength;

commandLength = strlen(cat) + strlen(argv[1]) + 1;

command = (char \*) malloc(commandLength);

strncpy(command, cat, commandLength);

strncat(command, argv[1], (commandLength - strlen(cat)) );

system(command);

return (0);

}

As standard, the user could then execute this command as shown below;

$ ./catWrapper Story.txt

This however due to its lack of input-validation, or even sanitization (not recommended in this case, due to a high possibility of error) has the potential for injection attacks to occur, if the user were to append the command with a *‘;’* and subsequently a command of their liking, in this case we will assume the *‘ls’* command is used. This would then execute both commands consecutively, first outputting the *cat* return value and then a list of all directories within the system.

Although this being just a very simple and obvious example of what an OS Command Injection may look like, the implications this can have on any system should it be allowed, could be catastrophic for the company and must be avoided at all points. Command injections tend to occur in applications that allow user provided input, and problems tend to occur when the OS executes this received input (Stasinopoulos, Ntantogian and Xenakis, 2019).

## Prevention?

The most important programming techniques to avoid command injections from occurring are to either completely remove the system shell from the process or employ extensive input validation and sanitization methods. Removing the system shell can be achieved through implementation of other safer platform API’s, this completely diminishes the possibility of an injection attack to occur. If the OS Shell is unavoidable in any calls however, as prior mentioned extensive input-validation must be carried out(Stasinopoulos, Ntantogian and Xenakis, 2015).

Things to consider removing when validating or sanitising any input include:

* Whitelisting permitted values.
* Confirming the input has only alphanumeric characters, no whitespace or other syntax.
* Ensuring the input is an integer value.
* Escaping input data (i.e. Rendering dangerous characters as simple text strings)

Although input sanitising is also possible it is highly ill-advised, it is both highly error prone and also has the potential to be bypassed by a particularly skilled attacker.

# SQL Injection

(Jenni Whewell)

## What is SQL Injection?

SQL injection in its entirety is the common use of attack using SQL code for database manipulation to access personal data which can then be used for financial gain. Used to destroy databases, SQL injection is one of the most common hacking techniques as SQL code is easy enough to manipulate. According to Stuttard and Pinto (2011) SQL vulnerabilities usually arise when “hard-coded strings are concatenated with user controllable data”. This forms an SQL query which can then be executed.

According to Acunetix’s article on SQLi (no date) there are 7 subclasses of SQL injection attack:

Classic (In-Band) SQLi:

This type of attack is the most common use as the attacker uses the same communication channel to attack and receive results.

Error-based SQLi:

This type of attack relies on error-based messages to gather information based on the structure of the database. Through just error messages alone, an attacked can gather the entirety of the database.

Union-based SQLi:

This type of SQL injection is the use of the “UNION” statement to combine results of 2 different “SELECT” statements from different tables to find flaws in the system.

Inferential SQLi:

This is the longer approach to attack a system but still as dangerous as the other forms. Also commonly known as a “blind injection attack”, the attacker cannot see the result of the attack. The idea of an inferential SQL injection is to reconstruct the structure of the database through the means of sending payloads.

Boolean-based SQLi:

This attack relies on the idea of query that only returns TRUE or FALSE. This is a slow attack as it must be done character by character and the attacker can use the result of the statement to infer the database structure.

Time-based SQLi:

This SQL injection attack is all about the time a database takes before responding. Based on this time, the attacker can infer whet\her the result is TRUE or FALSE. With this information, the attack can then go through the database structure character by character to gain access.

Out-of-band SQLi:

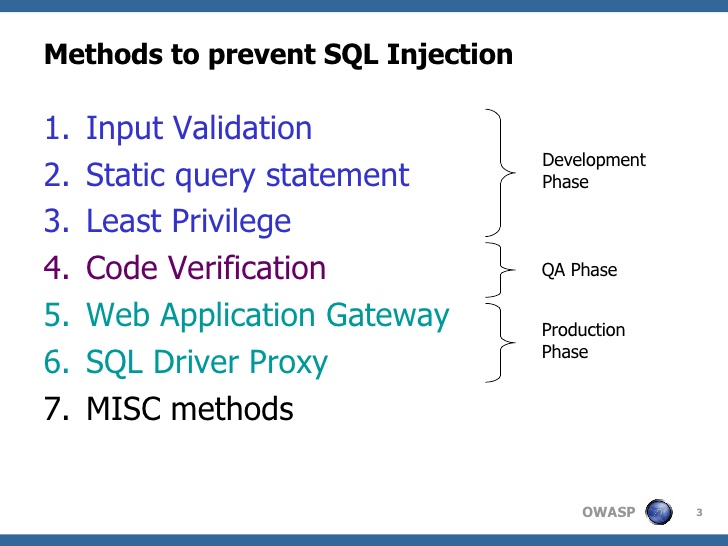
The most uncommon form of SQL injection as the attacker needs to already have access to the database server. It is used when the attacker cannot use the same channel to gather and receive result. It sends DNS or HTTP requests to receive data.

## How can it affect code?

SQL injection can affect code in many ways. In the event of a successful SQL attack, unauthorized access can be obtained to data which can potentially be used against a company, often for financial gain (Peikari and Chuvakin, 2004). Also, an attacker can use this method to insert malicious code into SQL statements (W3schools, no date).

## Prevention?

The best way to prevent an SQL attack is to utilize safe programming functions which make SQL injections impossible. Below is one methodology which, if followed correctly, could prevent SQL< injection ever becoming an issue later.



There are many defenses out there to combat SQL injection attacks including ‘Security by Obscurity’ to obfuscate the controls, the mythic ‘Magic Firewall’ by implementing barrier security tools between the database and the web and simply ‘doing things right’ using sound coding techniques and a database configured with a ‘least-privilege’ principle (Peikari and Chuvakin, 2004).

There are a few key steps to reduce the impact of an SQL injection attack:

* Trust No one
* Do not use dynamic SQL

# Tainted Data

(Chloe Andronicou)

## What is Tainted Data?

Computers these days are under a continuous risk of exploitation. Data that is unchecked may be potentially dangerous and might impose a serious threat to software. That is why all data that is passed through software must be treated as if it is malicious. This is known as tainted data.

## The Risk of Tainted Data

There are areas in the program that are exposed to attackers and these define the program’s attack surface. Taint sources, which are locations in the program where data is read from a potentially risky source, correspond closely to the attack surface of the program. These sources include network services, environment variables, data, and files. Attackers can trigger security vulnerabilities through these uncertain channels, and these might result in the crash of the program. Many of these vulnerabilities are triggered by tainted data.(Stewart, n.d.)

## Prevention?

Tainted Analysis in CodeSonar as mentioned by Stewart (n.d.), is a technique used to demonstrate how potentially malicious data can flow from one part of a program to another. This proves to be a useful technique in reducing the risk of attackers as in CodeSonar, tainted data flow can be visualized. The elements of the program involved in the flow can be overlay the regular code view. This then assists in pointing out the risks present in the code which then enables the affected code to be changed, resulting in the vulnerability being resolved. Any exploitable attack surfaces are then abolished consequently.

According to Bekrar et al. (2011), there is also a security testing method called fuzzing that includes entering invalid and random input into software to catch tainted data. By doing this, any errors and potential threats are picked up along with any unpredicted behaviour and can be resolved before they become an issue. Fuzzing is improved upon by a technique called data tainting and this is what tracks the spread of tainted data and checks if the data is used in any harmful ways such as overwriting return addresses.

# Unprotected Storage of Credentials

(Jack Davies)

## What is the Unprotected Storage of Credentials?

A system can be compromised when storing an application password in plaintext. This practice is considered to be poor password-management as storing confidential authorisation details in application properties or configuration files can be a security vulnerability (CWE, no date). If you store a password as plain text in a configuration file, it means that anyone who has access to that file can read the file access to the password-protected resources. Password authentication, is by its very nature, is inherently weak as they are low-entropy and often quite short (so as to be easily memorised) (Yang and Bao, 2010)

To quote Tsipenyuk, Chess and McGraw (2005), “software security is not security software”. They argue that that the majority of security features regarding user credentials relate less to the code itself and more to ‘principles of good practice’. This can relate to areas such as authentication, access and privilege management.

They lay out 9 broad areas through which software can be vulnerable to attack as a result of careless security practices:

1. Insecure randomness of cryptographic number generation.
2. Least privilege violation where elevated user access is not removed after use.
3. Not performing access control checks across all execution paths.
4. Saving password in unsecure plaintext.
5. Using an empty string as a password in a config-file.
6. Storing the password itself in an unsecure config-file.
7. Hard-coded credentials that use external components.
8. Using a weak cryptographing algorithm to obfuscate rather than encrypt.
9. Insufficiently protected credentials and mishandling of private information.

## How does it affect the code?

Where there is unprotected storage of credentials there can be a huge vulnerability risk. If any of the data you’re storing or sending is readable and unsecure, anyone can have (un)authorised access to your network. Which means if a hacker takes advantage of a credential vulnerability, and breaks into your device or network, if the data your storing is not encrypted securely, your confidential data will now belong to the attacker. When sending passwords in plaintext via email, the ‘man in the middle’ can harvest this information while the data is being transmitted between the sender and receiver’s devices. Which means it’s an easy win for the hacker as he now has your password. According to PassCamp.com (2020) 65% of users reuse their passwords across multiple platforms. This provides any potential hacker with the freedom to compromise more accounts and systems and steal or modify data.

## Prevention?

To prevent such vulnerabilities and attacks, developers should avoid risky coding practices. They should store password in easy to access locations and write it in plaintext in the software, using hard-coded credentials. Encryption at this point is absolutely critical, especially for outbound authentication. Those who have permission to view these files should only have access if it is completely necessary (i.e. least privilege access). Key function access permissions should be limited to authorised personnel only and strong security should be implanted on files and passwords using encryption keys (Veracode, no date).

It recommended to use strong one-way hashes for inbound authentication passwords. As a one-way function it theoretically results in creating undecipherable passwords as the information itself is lost in the hashing process. Although the ‘hashing and salting’ is truly only as strong as the original password on which it is encrypted (Peikari and Chuvakin, 2004, pp. 260). The hashes should be strongly protected using databases or configurations files. For every hash that is generated, it is advisable to use randomly assigned salts. The reason for this is that a hacker would find it more difficult to hack the credentials and are required to use more computing resources which will reduce the risk. Another way to prevent a hack is that the automatically generated passwords for front-end to back-end connections should be entered within a certain time frame, limiting the chance for hacker to hack the credentials.

# References

Acunetix (no date), *Types of SQL Injection (SQLi).* Available at: https://www.acunetix.com/websitesecurity/sql-injection2/ (Accessed: 24th January)

Bekrar, S., Bekrar, C., Groz, R. and Mounier, L. (2011) ‘Finding Software Vulnerabilities by Smart Fuzzing’, *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 427-430. Available at: doi:10.1109/ICST.2011.48. (Accessed: 5 February 2021)

Black, P. E., Bojanova, I. (2016) ‘Defeating Buffer Overflow: A Trivial but Dangerous Bug’, *IT Professional*, 18(6), pp. 58 – 61.

CWE (no date) *CWE-311: Missing Encryption of Sensitive Data.* Available at: https://cwe.mitre.org/data/definitions/311.html (Accessed: 20 January 2021)

CWE. (no date).CWE-256: *Unprotected Storage of Credentials.* Available: https://cwe.mitre.org/data/definitions/256.html. (Accessed 26 February 2021)

Deckard, J., Osipov, V. and Bhalla, N. (2005) *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Massachusetts: Syngress

Dietz, W., Li, P., Regehr, J., & Adve, V. (2015). Understanding Integer Overflow in C/C++. *ACM Journals, ACM Transactions on Software Engineering and Methology, Vol. 25, No. 1*.

Gupta, A. (2019) *The IoT Hacker’s Handbook: A Practical Guide to Hacking the Internet of Things*. California: Adpress.

Hanif, H., Md-Nazir, M. H., Ab-Razak, Firdaus, A. and Anuar, N. B. (2021) ‘The Rise of Software Vulnerability: Taxonomy of Software Vulnerabilities Detection and Machine Learning Solutions’, *Journal of Network and Computer Applications*, 179

Howard, M., LeBlanc, D. and Viega, J. (2010) *24 Deadly Sins of Software Security*, McGraw-Hill.

*Imperva* (no date) *Buffer Overflow Attack.* Available at: https://www.imperva.com/learn/application-security/buffer-overflow/ (Accessed: 24 January 2021)

Kembora, O., (2020) *Symmetric vs Asymmetric Encryption: What’s the Difference?*. Available at: https://blog.mailfence.com/symmetric-vs-asymmetric-encryption/ (Accessed: 21 January 2021)

Kuperman, B. A., Brodley, C. A., Ozdoganoglu, H., Vijaykumar, T. N. and Jalote, A. ‘Detection and Prevention of Stack Buffer Overflow Attacks: How to mitigate remote attacks that exploit buffer overflow vulnerabilities on the stack and enable attackers to take control of the program’, *Communications of the ACM*, 48(11), pp. 51 – 56.

McNab C. (2017) *Network Security Assessment.* 3rd edn.California: O’Reilly.

Muntean, P., Grossklags, J., & Eckert, C. (2017). Practical Integer Overflow Prevention, arXiv:1710.03720v9 [cs.CR]. *Cryptography and Security*.

Muntean, P., Monperrus, M., Sun, H., Grossklags, J., & Eckert, C. (2019). IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering ( Early Access )*.

Norton (no date) *What is Encryption and How Does it Protect Your Data?* Available at: https://us.norton.com/internetsecurity-privacy-what-is-encryption.html (Accessed: 20 January 2021)

Padmanabhuni, B. and Tan, H.B.K. (2011) ‘Defending Against Buffer-Overflow Vulnerabilities’, *Computer*, 44(11), pp. 53 – 60.

PassCamp (2020). *Dangers of Storing and Sharing Passwords in Plaintext.* Available: https://www.passcamp.com/blog/dangers-of-storing-and-sharing-passwords-in-plaintext/. (Accessed: 26th February 2021).

Peikari, C. and Chuvakin, A. (2004), *Security Warrior*, California: O Reilly.

*Portswigger* (no date) *OS Command Injection*, Available at: https://portswigger.net/web-security/os-command-injection (Accessed: 10 February 2021)

Stasinopoulos, A., Ntantogian, C. and Xenakis, C. (2015) ‘Commix: Detecting and Exploiting Command Injection Flaws’, *Dept. Digit. Syst., Univ. Piraeus, Piraeus, Greece, White Paper*. Available at: https://www.blackhat.com/docs/eu-15/materials/eu-15-Stasinopoulos-Commix-Detecting-And-Exploiting-Command-Injection-Flaws-wp.pdf (Accessed: 10 February 2021)

Stasinopoulos, A., Ntantogian, C. and Xenakis, C. (2019) ‘Commix: Automating Evaluation and Exploitation of Command Injection Vulnerabilities in Web Applications’, *International Journal of Information Security*, 18(1), pp. 49 – 72. Available at: doi:10.1007/s10207-018-0399-z (Accessed: 12 February 2021)

Stewart, L. (no date) *What is Taint Checking*. Available at: https://blogs.grammatech.com/what-is-taint-checking (Accessed: 05 February 2021)

Stuttard, D. and Pinto, M. (2011), *The Web Application Hacker’s Handbook: Finding and Exploiting Security Flaws*, 2nd edn. Indianapolis: Wiley Publishing.

*TechTerms* (no date) *Buffer.* Available at: https://techterms.com/definition/buffer (Accessed: 24 January 2021)

Tsipenyuk, K., Chess, B. and McGraw, G. (2005) ’Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors’, *IEEE Security & Privacy*, 3, pp. 81 – 84. Available at: doi:10.1109/MSP.2005.159.

Veracode (no date) *Credentials Management Flaws Information, Tutorial and Cheat Sheet.* Available at: https://www.veracode.com/security/credentials-management. (Accessed: 26th January 2021)

Veracode (no date) *What is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks.* Available at: https://www.veracode.com/security/buffer-overflow (Accessed: 25 January 2021)

W3schools (no date), *SQL Injection*. Available at: https://www.w3schools.com/sql/sql\_injection.asp (Accessed: 25 January 2021)

Wang, T., Song, C., & Lee, W. (2014). Diagnosis and Emergency Patch Generation for Integer Overflow Exploits. *Detection of Intrusions and Malware, and Vulnerability Assessment* , 255-275.

Yang, Y. and Bao, F. (2010) ‘Password Protected Credentials’, *2010 International Conference on Multimedia Information Networking and Security’*, pp. 541 - 545

Zhong, W. (no date) *Command Injection,*  Available at: https://owasp.org/www-community/attacks/Command\_Injection (Accessed: 19 January 2021)