INTERNATIONAL JOURNAL FOR TRENDS IN ENGINEERING & TECHNOLOGY
VOLUME 28 ISSUE 1 – APRIL 2018 - ISSN: 2349 – 9303

Secure Coding: As a basic Security Paradigm

Tabassum Gull Jan
Central University of Punjab,
Computer Science and Technology,
tabassumgull2012@gmail.com

Sheikh Irfan Akbar
Central University of Punjab,
Computer Science and Technology,
sheikh.irfan4630@gemail.com

Abstract— in today's era of technology, issues related to data security and privacy have become a major concern and a challenging problem. Every program or data is a potential target of an attacker. So, to better secure the data is an important aspect of the study because attacker will not be able to find the security vulnerabilities and gain access to the data. Secure coding is a practice or process of writing the secure programs that are immune to attack by a malicious user. In this paper, we have highlighted how different secure coding practices can be employed to avoid buffer overflow, buffer underflow and string manipulation vulnerabilities. Apart from this, we have also given a brief explanation of how to avoid SQL injection attacks using secure programming.

Index Terms— Security, vulnerabilities, attack, SQL injection, threat, Buffer overflow, Buffer underflow.

1 INTRODUCTION

We live in an era of unprecedented technological growth, fueled by data and information security. In this age of information, softwares are the primary means of taming the information. Without adequate and efficient security mechanisms, we cannot realize the full potential of the digital age. The sharp increase in the Internet and rapid increase in the production of web-based applications and information systems, and recent trends such as cloud computing and outsourced data management, have increased the exposure of data and made security more difficult. We use softwares to automate factories, streamline business, e-governance, and e-commerce and put information into the hands of people who use it. Hence there are maximum chances of changing the security of software by exploiting the vulnerabilities. Changing the software security requires changing the way software is built or changing the ways how the program solicits input whether it be from a user, from a file or any other means. Security of data, softwares cannot be 100 percent achieved as there are countless number of security mistakes that programmers could make while saving data, while coding and development of a software. Software developers and programmers need to know the essence and importance of secure coding, which can help in protecting the information that is accessed or provided by different web and mobile application users. The best security practices, if included, in the early development phase of applications/softwares will ensure confidentiality, integrity, and availability of the information. There are some coding mistakes that are generally and inadvertently introduced in the common coding practices that increase the risks and vulnerability areas in the digital application Vulnerabilities continue to grow at an alarming rate. So, grows the security risks of the software and hence less secure become the data and information concerned with software. Hence there is a need, that secure coding practices must be incorporated in each development stage of the web application to provide protection against cyber-attack, cybercrime, and cyber espionage. When secure coding best practices are applied during initial developmental phase of an application, it guarantees us to provide security against SQL Injection**,** Command line injections, exposing sensitive data, path manipulation, missing function level excess control, metacharacter vulnerabilities,, Broken authentication and session management, string manipulation errors, Cross-site scripting attack, Insecure direct object references, security misconfiguration, Cross-site request forgery, using components with known vulnerabilities, invalidated redirects and forwards [1].

2 SECURE CODING

Secure coding is the practice of writing a source code or code base that is immune to attacks by a malicious attacker and is compatible with the best security principles for a given system and interface. Producing secure programs require secure designs. However, sometimes best designs lead to insecure programs because of unawareness about many security put falls that are inherent in the system or the programming language they are using [2]. To write good code is an art of a programmer, but then how a bad code is written which later on becomes a security vulnerability. Bad code is written by programmers who are not aware of the secure coding practices or carelessness of the programmer when defining variables, strings, functions, structures, passing values to the functions, passing references as pointers from one module to the other module.

2.1 Overview of Secure Coding Practices or Code Security

The most important defensive measure software developers can take is to keenly observe and validate the input their software receives. Input validation and representation is the basic and most important secure coding practice because an unchecked or improperly validated input to the software is the source of worst vulnerabilities around, including string manipulation errors, buffer overflow, buffer underflow, SQL Injection and many more. Input validation plays a very critical role in the program from a

security point of view. The program first reads the data from outside sources until it uses it later in many security-related contexts. Hence the two aspects of input validation are the kinds of input that require validation and second one is the kinds of operations that depend on validated input [5], [6]. The secure coding practices related to handling input are:

- **Validate all input that is in the context of your application:** validate all input in such that there is no chance for the programmer to say that I forgot to do input validation.
- **Validate input from all sources:** Validate input from all sources, whether it is an input coming from command line arguments, configuration files, data retrieved from a database query, environmental variables, network services, system properties, registry values, temporary files, and another outside source for the program input. Make sure syntax and semantic checks are performed on every piece of input [13]
- **Establish trust boundaries between the data:** Store trusted and untrusted data separately to ensure that input is properly validated.
- Prefer whitelisting or indirect selection as strong input validation techniques.
- **Avoid blacklisting.**
- **Don't confuse usability with security:** Any check that a program performs for usability purposes is not meant to ensure security as well.
- **Discard bad data:** Reject data that fail input validation checks. Do not repair it or sanitize it for its reuse.
- **Using the security API's make good input validation the default.**
- **Always apply length checks against a minimum and maximum expected length**.
- **Bound numeric input against maximum and minimum values to ensure no integer overflow occurs.**

2.2 Buffer overflow and underflow vulnerability

A buffer overflow occurs when a program writes data outside the bounds of allocated memory. Buffer overflow vulnerabilities are usually exploited to overwrite values in memory to the advantage of the attacker. A major source of security vulnerabilities in C, Objective-C, and C++ code are buffer overflow vulnerabilities both on the stack and on the heap. Basically, what happens in case of buffer overflow the program writes the attacker's code into the buffer and continues beyond the buffer bounds until function's return address is overwritten with the starting address of the malicious attacker's code. When the function returns it jumps to the value stored in the return address. Normally function will be returned to the context of calling function, but because the return address has been overwritten control jumps to the buffer instead and starts executing the malicious code.

```
void BufferOverflowAttack () {
        int a= 32;
        char line [128];
        gets (line);
}
```

Let us suppose 0xNN be the starting address of BufferOverflowAttack () function and 0x<return> points to the function that called BufferOverflowAttack () function. The attacker uses NOP (No Operation) simply to bypass the memory locations for which user's code is running.

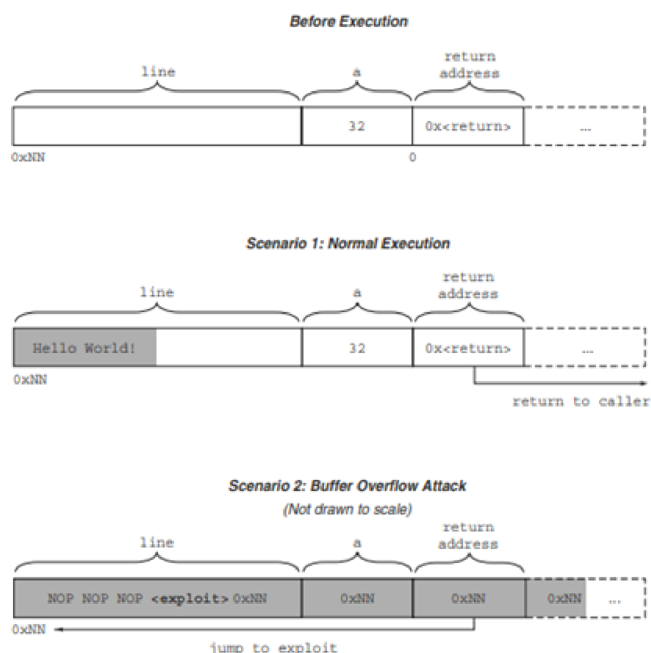

Figure 1

Figure 1 shows how buffer overflow exploits happen and the stack frame of BufferOverflowAttack () before execution, normal execution and buffer overflow Attack [6].

Buffer overflows, a common software security vulnerability and it happens when the user inputs data that has a length longer than the reserved space. If we don't truncate it, the memory locations beyond the reserved ones will be overwritten. This will result in a security vulnerability (stack smashing) or program termination (segmentation fault) [14]. If the overwritten memory contains data essential to the program operation, hence resulting in a bug.

A simple example of a C program prone to a buffer overflow is:

```
int overvulnerable_function (char * large_user_input) {
        char dst [SMALL];
        strcpy (dst, large_user_input);
}
```

Buffer overflow will occur if the user input is larger than the destination buffer. To fix this unsafe program, use strncpy () to prevent a possible buffer overflow. In strncpy () we specify the maximum no of characters to be copied.

```
int little_secure_function (char * user_input) {
        char dst [BUF_SIZE];
```

```
    //copy a maximum of BUF_SIZE bytes
        strncpy (dst, user_input, BUF_SIZE);
}
```

Alternative secure way is to dynamically allocate memory on the heap using malloc () function.

```
char * more_secure_copy (char * src) {
        int len_string = strnlen (src);
        char * dst = (char *) malloc (len_string + 1);
        if (dst! = NULL) {
                strncpy (dst, src, len_string);
                //append null terminator
            dst [len_string] = '\0';
        }
        return dst;
}
```

The code snippet above attempts to copy the contents of ***src*** (source) into ***dst*** (destination), while also checking the return value of malloc to ensure that enough memory was able to be allocated for the destination buffer.

2.3 Buffer Underflow

Basically, buffer underflow occurs when the input data is shorter or appears to be shorter than the reserved or allocated space. Buffer underflow is due to erroneous assumptions, incorrect length values, or copying raw data as a string used in the program. The main cause of problems from incorrect behaviour to leaking data that is currently on the stack or heap is the buffer underflow. Although input checks are performed for buffer underflow, a buffer overflow in many languages, but C, C++ does not [4].

2.3.1 Avoiding buffer overflows

When two parts of code disagree about the buffer size or the buffer content, we say buffer underflow has occurred. For example, a fixed-length string variable might have a vacancy of 56 bytes, but contains a string that is only 22 bytes long [2], [8]. When the correct operation depends upon both parts of the code treating the data in the same way, buffer underflow conditions become more dangerous. This occurs most often when you read the buffer to copy it to another block of memory, to send it across a network connection.

There are two main classes of buffer underflow vulnerability: short reads and short writes.

A short-read vulnerability happens when a read from a buffer fails to read the complete buffer contents. When the program makes decisions based on that short read, a number of erroneous behaviours can result. For example, take a C string, which is made of a series of bytes that end with a null terminator. But this string cannot contain null bytes or '\0' prior to the end of the string. Hence if a user entered '\0' or null terminator in the text, it will be treated as an end of the string and rest operations like strncpy (), strlen () and so on can be done leading to incorrect results.

A *short write vulnerability* occurs when a short write to a buffer fails to fill the buffer completely. When this happens, some of the data that was already in the buffer is still present after the write. If the application later performs an operation on the entire buffer (writing it to disk or sending it over the network, for example), that existing data comes along for the ride. The data can be garbage values, but if the data happens to be interesting, you have an information leak.

If the values in the locations affect program flow, the underflow can cause an incorrect behaviour, allowing the user to skip or bypass an authentication or authorization step.

Rules that one should obey to avoid most buffer underflow attacks:

- Zero-fill all buffers before use so that they don't reveal any sensitive information.
- Always check return values.
- If a call to an allocation or initialization a function fails, do not evaluate the resulting data, as it could be unsafe.
- Use the value returned from read system calls and other similar calls to determine how much data was actually read.
- Display an error if a write call returns without writing all of the data
- Perform length checks for data structures to verify that the data is the size you expected.
- Always check for null bytes in the source data.
- Avoid mixing buffer operations and string operations.

3 BUFFER OVERFLOWS AS A STRING MANIPULATION ERROR

Strings are the most common form of user input. Since many string-handling functions have no built-in checks for string length that is why strings are frequently the source of buffer overflows. Figure 2 illustrates the different ways three string copy functions handle the same string over-length string [4].

Char destination [5];
Char *source=" LARGER";

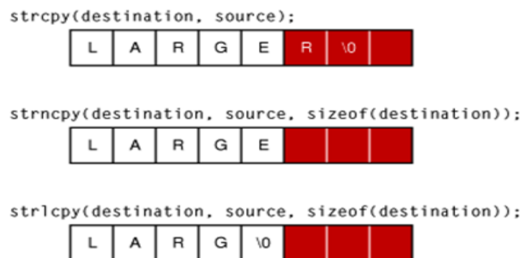

Figure 2

The strcpy () merely writes the whole string overwriting the locations that came after the reserved location for storing the string. The strncpy function truncates the string to the correct length, but without the terminating null character or '\0'. When this string is read, then, all of the bytes in memory following it, up to the next null character, are read as part of the string [8].
The strlcpy function is more secure, truncating the string to buffer size minus one and adding the '\0'. Secure coding does and does not for string functions are shown in table 1.

Don't use these functions	Use these instead
Strcat	Strlcat
Strcpy	Strlcpy
Strncat	Strlcat
Strncpy	Strlcpy
Sprintf	snprintf
Gets	Fgets

Table1.

4 SQL INJECTION USING COMMAND LINE ARGUMENTS

There are already vulnerabilities in the languages that exist because of poor design, for example, the Java version of Hibernates Schema Export tool accepts a command-line parameter which it uses to separate SQL commands in the scripts, it generated. These options exist, so that a user can specify the separator that should appear between SQL statements. The typical values user might enter can be a semicolon or a carriage return and a line feed. But the program does not place any restrictions on the argument's value. So, from the command-line parameter, the user can write any string you want into the generated SQL script, including additional SQL commands. SQL injection attack uses malicious SQL code for backend database manipulation to access sensitive information that is otherwise not intended to be disclosed [14].SQL attack can be used to attack any SQL database where websites are the most frequent targets.
A user-provided input
http://www.snapdeal.com/items/items.asp?itemid=999 can then generates the following SQL query.
SELECT ItemID, ItemDetail FROM Item WHERE ItemNO=999
An attacker wishing to exploit non-validated input vulnerabilities in a database executes SQL injection manipulates a standard SQL query to, For example, the above Query input, which pulls information for a specific product, can be altered to read http://www.snapdeal.com/items/items.asp?itemid=999 or 1=1
The corresponding SQL query looks like this:
SELECT ItemID, ItemDetail FROM Item WHERE ItemNO=999 OR 1=1
Since the statement 1 = 1 is always true, the query returns all of the product ID's and details in the database, even those that you may not be eligible to access.

Attackers are also able to take advantage of incorrectly filtered characters to alter SQL commands, including using a semicolon to separate two fields.
For example, this input:
http://www.snapdeal.com/items/iteams.asp?itemid=999; DROP TABLE Users would generate the following SQL query:
SELECT ItemID, ItemDetail FROM Item WHERE ItemNO=999; DROP TABLE Users
As a result whole Users table will be deleted, another way SQL queries can be manipulated is with a UNION SELECT statement.
A good answer for many metacharacter problems is the use of parameterized commands.

4.1 SQL injection prevention strategies

Sanitization can be basic precaution which usually involves executing any submitted data through a function to ensure that any vulnerable or dangerous characters like " ' " are not passed to a SQL query in data Validation is carried out in two different ways: by blacklisting dangerous or unwanted choices of a user input and by whitelisting [2]. Here are some preventive mitigation strategies for SQL injection attacks:

- **Trust no-one**
- **Use prepared statements, parameterized queries or stored procedures instead of SQL query whenever possible**.
- **Firewall**
- **Reduce your attack effort**
- **Use appropriate administrative privileges.**
- **Keep your secrets always secret.**
- **Don't reveal more information than you need to.**
- **Don't forget the basics of security.**

Keeping data and control information separate metacharacter vulnerabilities can be eliminated. Some interfaces have a built-in facility, try using them and some interfaces allow the program to supply a parameterized request and a set of data values to be filled in as parameter values. The enabling factor behind SQL injection is the ability of an attacker to change the context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as part of a command instead [7]. When a SQL query is constructed, the programmer knows the difference between the data and the command. When used correctly, parameterized SQL statements enforce this distinction by disallowing data-directed context changes and preventing many SQL injection attacks. Parameterized SQL statements are made using regular SQL by binding user-supplied input with placeholders for the data. Forming an SQL query by concatenating string leaves and the code becomes vulnerable to SQL injection attack [9], [10], [12].

```
String UserName=ctx.getAuthenticatedUsername ();
String itemname=request.getParameter ("itemname");
```

```
String query=" SELECT * FROM items WHERE owner='"+UserName+"' AND itemname='"+itemname+"'
";
Statement stmt=conn.createStatement ();
ResultSet rs=stmt.executeQuery (query);
```

Using parametrized query helps to prevent SQL injection as

```
UserName=ctx.getAuthenticatedUsername ();
String itemname=request.getParameter ("itemname");
String query =" SELECT * FROM items WHERE owner =?" +" AND itemname=?"
PreparedStatement stmt=conn.prepareStatement ();
stmt.setString (1, UserName);
stmt.setString (2, itemname);
ResultSet rs=stmt.executeQuery (query);
```

Preventing SQL injection can also be viewed as a problem of input validation. A programmer might accept characters only from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. As strict input validation rules are enforced in whitelisting, but parameterized SQL statements require less maintenance and guarantees more security with respect to other.

5 Conclusion

Secure coding practices is a mandatory concept, that helps in developing secure and robust web and mobile applications that practically reduce the security threats, risk areas and vulnerabilities. To avoid buffer overflows, underflows, string manipulation errors and SQL injection attacks different secure coding principles and practices can be employed. Buffer underflows cause incorrect behaviour data/information leakage and are the main source of bugs in a program. Similarly, buffer overflows due to string manipulation cause malicious code injection and arc injections. SQL injection attacks are caused because of improper input validation. Command line arguments need to be sanitized before using them. We can use parametrized SQL instead where a programmer might accept values from a whitelist and reject values in the blacklist. However, preventing SQL injection using Parametrized SQL is a bit vulnerable to attacks in case there exist loopholes in the blacklist user for input validation.

6 Acknowledgments

We would like to thank Almighty Allah for giving us this opportunity. We also extend our word of thanks to the faculty Department of Computer Science and Technology, Central University of Punjab, all the authors and researches whose work and research was really helpful.

Author Profile:

- *Tabassum gull Jan is currently pursuing master's degree program in Computer Science and Technology (Cyber Security) in Central University of Punjab, India, PH-7889856176 E-mail: tabassumgull2012@gmail.com*
- *Sheikh Irfan Akbar is currently pursuing master's degree program in Computer Science and Technology in Central University of Punjab, India, PH-7889388168. E-mail: sheikh.irfan4630@gmail.com*

7 References

[1] N. G. Q. A. M. T. Maleerat Sodanil, "A knowlwdge Transfer framework for secure coding practices," in *international joint conference on computer science and software engineering*, songkhla thailand, 2015.

[2] C. H. T. S. A. T. Keqin Li, "Tool support for secure programming by security testing," in *2015 IEEE Eighth International conference on software testing, Verification and validation workshops(ICSTW)* , Graz, Austria, 2015.

[3] S. S. Kanchana Natarajana, "Generation of SQL-injection free secure algorithm to detect and prevent SQL-injection attacks," *sciverce science direct,* 2012.

[4] https://en.wikipedia.org/wiki/Secure_coding.

[5] http://skillcube.in/importance-of-secure-coding-in-making-digital-india-successful/.

[6] J. W. Brain chess, "Secure programming with static analysis," in *Handling and Buffer Overflows*, USA, Gray McGraw, 2010.

[7] http://www.upenn.edu/computing/security/swat/SWAT_Top_Ten_A5.php.

[8] https://developer.apple.com/library/content/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html.

[9] http://www.upenn.edu/computing/security/swat/SWAT_Top_Ten_A6.php.

[10] https://www.synopsys.com/blogs/software-security/prevent-sql-injection-attacks/.

[11] http://www.enterprisenetworkingplanet.com/netsecur/article.php/3866756/10-Ways-to-Prevent-or-Mitigate-SQL-Injection-Attacks.htm.

[12] https://www.incapsula.com/web-application-security/sql-injection.html.

[13] https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices.

[14] https://www.incapsula.com/web-application-security/sql-injection.html.