Investigating the Input Validation Vulnerabilities in C Programs

Input Validation in C

Shouki A. Ebad
Department of Computer Science
Faculty of Science, Northern Border University
Arar, Saudi Arabia

***Abstract*—Input validation is a fairly universal programming practice that helps reduce the chances of producing protection-related vulnerabilities in software. In this paper, an experiment is conducted to specifically determine the input validation issues found in programs and the problematic functions that lead to such issues. The experiment evaluated 12 arbitrarily selected open source C projects written by different programmers. The top two most common input validation problems are buffer overflow/XSS and potential memory mismanagement. In addition, the functions that caused the first problem are: (a) strings/text functions (e.g., strcpy and strcmp), and (b) functions that read from standard input, STDIN (e.g., scanf and gets). In contrast, the functions that caused the second problem are (a) memory allocation/deallocation functions (e.g., memmove and malloc), and (b) file manipulation functions (e.g., fopen and fseek). Furthermore, the goto construct—to a small extent—plays a role. The recommendations are that (a) developers are encouraged to use memory-safe programming languages, otherwise, they should perform different types of checks for the validity of inputs as soon as they are entered, and (b) they should have the required knowledge of secure source code and use tools/suites to manage malformed strings.**

***Keywords*—Input validation; buffer overflow; memory mismanagement; safe C functions**

I. Introduction

Input validation is accepted as good programming practice when writing reliable software. This practice is fairly universal and helps reduce the chances of introducing protection-related vulnerabilities in delivered software [1][2]. This practice can be applied regardless of the programming language (PL) used in development, although the way it is used depends on the specific PL and notations that are used for software development. Every software takes input from its environment and processes it. The specification of such an input makes assumptions about this input that reflects its real-world use. For instance, it may be supposed that an employee ID is always a 10-digit positive integer. However, the software specification does not determine the steps that should be taken in case of wrong input. The user often makes mistakes and sometimes enters the data incorrectly. Regardless of the type of source of unexpected inputs (human, IoT devices, sensors, the system itself[1], or other systems), the software can behave in an unanticipated way and provide incorrect values. Such inputs may lead to many security issues. One of these is memory error exploitation, which still ranks among the top three most dangerous software vulnerabilities [1][3].

Buffer overflow and SQL injection are two examples of memory and string-based attacks. The former may be executed using long input strings, and the latter may be performed when a user enters a piece of SQL that is interpreted by a server [4]. The decision that one makes if any validation check fails relies on the software type being developed. For example, while reporting the issue and requesting to re-input the value may be enough in a business application, the input value might have to be estimated according to previous data in a real-time system that should be operated continuously. In contrast, if the source of the input value is a sensor, the most recent valid value could be enough to use. The paper is structured as follows. Section II reviews the existing literature. The research approach is stated in Section III. In Section IV, the results are analyzed and discussed. Section V discusses threats to the study's validity. Finally, Section VI summarizes the article and presents plans for future work.

II. Related Work

Scholte et al. [4] proposed a technique to prevent the exploitation of cross-site scripting (XSS) and SQL injection vulnerabilities based on the automated data type detection of input parameters. They implemented the technique for PHP and validated it on five web applications with known XSS and SQL injection vulnerabilities. Their technique prevented 83% of SQL injection vulnerabilities and 65% of XSS vulnerabilities while incurring no developer burden. Veen et al. [3] presented memory errors and considered attacks, defenses, and statistics. During a short period in the 70s, they discussed buffer overflows and established CERTs, Bugtraq, and various important methods and countermeasures. A set of related research areas is also explored. Tirronen [6] proposed a technique to eliminate SQL injection attacks by enabling web applications to work with abstract syntax trees while ensuring uniform interpretation of the result. The method involved moving away from processing data as strings to implement a

[1] For configuration, a mobile app can directly read the inputs from itself [5]

non-trivial XSS-protected application with the method in a limited resource. Braz et al. [2] designed an online survey involving 146 participants to understand the extent to which programmers can(not) discover improper input validation vulnerabilities. The participants were assigned to changes with one of two vulnerabilities: (a) SQL injection, and (b) improper validation of specified quantity input. Only 45% of the participants found the vulnerability. There was a lack of knowledge and practices to detect vulnerabilities when an attack scenario is not visible.

Rodr´ıguez et al. [7] analyzed over 50 documents to gather information on the techniques and tools that were used to discover XSS attacks. Their results showed that the trend was increasing in the analysis of content and patterns and decreasing in the use of artificial intelligence to reduce such attacks. Zhao et al. [5] demonstrated an important application of input validation, exposing input-triggered secrets such as backdoors and blacklists of unwanted items. They proposed a tool to find both the execution context of user input validation and the content involved in the validation to expose hidden functionality. The tool was tested with many mobile apps and it was found that they contained more than 12,000 and 4,000 backdoor secrets and blacklist secrets, respectively. Pereira et al. [8] studied a number of buffer overflow vulnerabilities in the Linux kernel, Mozilla, and Xen open-source C/C++ projects to analyze possible methods of improving their detection. The results showed that most of the vulnerable source code units were with defects in checking and dealing with input data types. Static analysis tools lack rules to detect missing or incorrect checking logic vulnerabilities. Moreover, there is no causality between buffer overflow existence and the value of software metrics. Khalaf et al. [9] explored detecting/removing bugs from client-side and server-side code using an input validation mechanism. They supported tests using easy-to-use and accurate models. A program statement that was vulnerable in SQL injection was checked according to static attributes. They also presented a script whitelisting built into the browser's JavaScript engine, where the SQL injection was detected and the XSS attack resolved using that mechanism.

In conclusion, in contrast to the existing literature, this study tries to identify the most common security issues related to input validation and the functions that are the source of such issues. This will help programmers and developers to give more attention to such functions.

III. Experimental Approach

An experiment was performed to understand how well C programs follow input validation practices. The details of the experiment are discussed in the following subsections.

A. Research Questions (RQs)

The following RQs are investigated:

RQ1: What are the most common input validation vulnerabilities at source code level?

RQ2: Which functions are the source of the majority of the above input validation vulnerabilities?

B. Subjects and Variables

As a representative of software source code, C programs are used; they are collected from the GitHub repository[2]. C is still one of the most common PLs in the market [10]. A controlled experiment is conducted where the subjects of the experiment are 12 open source software (OSS) programs written in C; the subjects are arbitrarily selected. The software developer of the subjects is a factor that could have an impact on the results because individual developers could use the same input validation practices for all of their programs. Therefore, we control this variable by selecting software programs that were designed and developed by different people.

C. Data Source and Tools

In order to evaluate the RQs, the data source in our analysis is restricted to the C PL, since (1) it is widely used, and (2) it provides input validation functions or constructs. Additionally, the experiment can be performed without any confounding factors introduced by different PLs. By restricting to just one PL, the results can be placed in context, and we can have more confidence in the conclusions. Table I describes the data source used for this experiment. There is a diversity in domains: education, business, management, tourism, and health. The differences among experimental subjects (i.e., C programs) were not substantial. The well-known metric of non-commented lines of code (LOC) was used to measure the program size. The mean size is 728.2 LOC, indicating the programs are small and medium-sized.

Visual Code Grepper (VCG) was used to collect the input validation violations of C programs. VCG is a source code security tool (available on the web[3]) to handle different programs, including those written in C. The statistical data were analyzed using Microsoft Excel 2016.

IV. Results and Discussion

Table II shows the memory-related input validation issues that were discovered by VCG. According to Table II, the top two most common input validation vulnerabilities are potential memory mismanagement and buffer overflow. Before going into the main results in depth, we should first note that the reader is supposed to be familiar with memory and string functions in C. Interested readers can refer to any C textbook for more information.

[2] https://github.com/

[3] https://sourceforge.net/projects/visualcodegrepp/

TABLE I. DETAILS OF THE EXPERIMENTAL SUBJECTS (C PROGRAMS)

Subject	Project name	Domain	Program name	Program LOC
CP1	Stellarium	Science	`Indiserver.c`	1528
CP2	Bank-Management-Program	Management	`bank management system.c`	507
CP3	Departmental-store-management-system	Business	`finalProject.c`	453
CP4	Library-Management-System	Education	`FINAL PROJECT.c`	1668
CP5	Calendar	Date and time	`Calender&age.c`	145
CP6	Contact-Management-System	Management	`code.c`	204
CP7	Pharmacy management system	Health	`Phamacy Managment System.c`	913
CP8	Student-record-system	Education	`main.c`	943
CP9	Phonebook Application	Management	`Phonebook Application.c`	288
CP10	Personal Diary Management system	Personal	`Personal Diary Managment System.c`	618
CP11	Hotel-menu-and-billing-main	Business	`main.c`	49
CP12	Tux paint	Entertainment	`onscreen_keyboard.c`	1422
Average				728.2

TABLE II. MAIN ERRORS REPORTED BY VCG

Vulnerability / Subject no.	Buffer overflow	Potential Memory Mismanagement	Accepting anonymous internet connection or unverified input data	Can expose residual memory contents	Facilitate format strings bugs	improper control flow	Sum
CP1	48	3	2	5	2	0	60
CP2	42	0	0	0	0	0	42
CP3	28	0	0	0	0	2	30
CP4	25	0	0	0	0	4	29
CP5	4	0	0	0	0	0	4
CP6	18	0	3	0	0	0	21
CP7	29	0	0	0	0	1	30
CP8	107	1	0	0	0	0	108
CP9	2	0	0	0	0	0	2
CP10	32	0	0	0	0	1	33
CP11	6	0	0	0	0	0	6
CP12	23	10	4	2	3	0	42
Sum	364	14	9	7	5	8	407
Mean	30.3	1.2	0.8	0.6	0.4	0.7	33.9

A. Potential Memory Mismanagement

This problem includes a variety of memory-related errors such as memory leaks, using memory inefficiently, invalid deallocation, double frees, heap corruption [4], memory overhead, and file-access violation [3]. The reader is assumed to be familiar with such type of errors; interested readers can consult [3] for more information. Table III shows a sample of lines of code and functions that caused the memory mismanagement problems in C programs. From Table III, the functions and constructs that caused this problem can be divided into three groups:

- Memory allocation/deallocation functions such as `memmove`, `malloc`, `free`, `new`, and

[4] *Memory leaks* result from memory that is allocated but never freed. Using *memory inefficiently* happens when a program allocates memory and fails to use it for a long time. It doesn't constitute a memory leak, but can waste a significant amount of memory.

`delete`.

- File manipulation functions such as `fopen`, `fprint`, and `fseek`.
- The `Goto` construct.

As a feature in C, the above C programs offered control on memory usage by allowing the optimization of memory allocation for their resources. However, this makes the developers responsible for tracking any memory that their programs dynamically allocate/deallocate. Otherwise, memory-related input validation problems will be introduced as in most cases in Table IV. Some of the above experiment subjects (e.g., CP12) accessed memory via pointers, which produced a memory access error. Use of `goto` should be minimized as much as possible; programmers have been urged to abandon the `goto` statement for more than 50 years on the advice of Dijkstra [11]. Despite this, it is still very much used in C projects [12]. Fig. 1 shows a sample of CP2's source code that uses `goto` in such a case. Use of `goto` came from the fact that C is a non-memory-safe PL in the input validation principle. In particular, it does not have explicit error handling and cleanup constructs like try/catch (for exceptions and errors handling) and finally (for cleanup activity) in Java; the programmers must therefore resort to using `goto` statements. Most of the current C projects that used `goto` are for these two purposes [12].

In general, tracking memory is not simple—even programs written by skilled programmers contain such problems. The issue originates when an unallocated area is corrupted, and a fatal error often happens in the coming allocation request. Besides the use of `goto` in not handling exceptions and input errors [12], there are three main causes of potential memory mismanagement problems [8].

- Passing a wrong parameter to an allocation function such as `malloc()` and `realloc()`.
- Passing invalid data to a deallocation function such as `free()` and `delete()`.
- Writing before/after the start/end of the allocated space, causing an underrun/overrun error.

B. Buffer Overflow/XSS

Buffer overflow is a form of memory mismanagement problem. It happens when code goes beyond the portion of data assigned to a buffer. In particular, code with a limited-size buffer accepts unlimited length input. In such a case, the program can crash or malicious code can be executed. In recent years, this issue has grown rapidly with web applications; it is known as XSS attacks, which allow an attacker to insert client-side scripts into web pages that the victim can access; it is also known as internet buffer overflow [9]. As mentioned earlier, writing data to memory beyond a buffer occurs with non-memory-safe PLs like C and C++ that have no bounds checking. Table V shows a sample of lines of code and functions that cause the buffer overflow problem in C programs.

From Table IV, several C functions are known to be unsafe and the source of the vast majority of buffer overflow attacks. They can be divided into two groups:

- Functions to read from STDIN (standard input) such as `scanf()`, `fscanf()`, `sscanf()` and `gets`() where inputs are taken from the keyboard or file.
- Functions to manipulate strings/texts such as `strcpy()`, `strcmp()`, `strlen()`, and `strtok()`.

The advice herein is that C programmers should never use these functions. Fortunately, there are safer alternatives to such unsafe functions. The safer alternatives to `strcpy()` and `strcmp()`, for example, are `strncpy()` and `strncmp()`, respectively. However, the safer functions are not completely safe because `strncpy()` was a cause for buffer overflow in CP1 in Table V. This finding has been confirmed by previous researchers [8]. In particular, the unsafe `strcpy()` takes two arguments—destination and source—and the function copies the source, including the `NULL` character, to the destination. Contrary to this, the safe `strncpy()` function takes the same two arguments as well as n, an unsigned integral type; the function copies the first n characters of source to destination i.e., at most, n bytes of the source are copied. If there is no `NULL` character in the first n characters of source, the string placed in the destination will not be `NULL`-terminated. Therefore, there is no guarantee that the destination will be `NULL`-terminated i.e., a non-terminated string in C is waiting to destroy the program. The question is why such functions were then built in C? The answer comes from C's history; those functions were particularly designed to address specific problems in manipulating strings stored in the manner of original UNIX directory entries, which use a short limited-size array of 14 bytes, and a `NULL`-terminator was only used when the filename was less than the array.

C. Timely Recommendations

Although this problem has been known for decades, it is still found in C/C++ software, as has been seen in this study. Any application must not be vulnerable to input validation-based attacks. Therefore there are three timely recommendations herein:

- Use memory-safe PLs. Developers should try not to use non-memory-safe PLs that fail to validate inputs; such a failing can not only lead to buffer overflow attacks (due to long input) but also DoS attacks (due to low memory). In contrast, safe PLs can address these challenges because they check, at runtime, that any access to the memory is within the declared bounds; they remove most buffer overflows at source.
- Perform input validation checks. The first recommendation would not always be a good choice due to the trade-off between performance and security. With memory-safe PLs, there is a necessary performance penalty for this validation, and, for that reason, much code will continue to be written in C. In such a case, the validity of the input should be checked

as soon as it is read. This check includes the specification of the format and structure of the expected inputs, especially considering there are different sources for input, as mentioned in Section 1. Input validation then relies on different checks; used in input checks when the software is implemented, four types of checks [1][5] are shown in Table V.

- Undergo training in writing secure code: The findings have indicated a lack of knowledge and practices to find vulnerabilities. This finding has also been confirmed by a recent study [8] that showed that most buffer overflow vulnerabilities are associated with missing checks (e.g., missing `if` construct around a statement) or incorrect checking (e.g., the wrong logical expression used as a branch condition). Regardless of the PLs used in coding, developers should have the required knowledge, training, and practices of secure source code.
- Use appropriate security tools: they should also use static and dynamic analysis security tools that include the use of suites of prebuilt attacks and malformed strings that can quickly discover and eliminate different software vulnerabilities [13]. Examples of such tools that can help developers in this regard include Clang-Tidy, FlawFinder, and Loggly by SolarWinds, which focus on insufficient input validation, XNU memory, and log file analysis and SQL injection, respectively [14].

TABLE III. MAIN ERRORS OF POTENTIAL MEMORY MISMANAGEMENT REPORTED BY VCG TOOL

Subject no.	Problematic statement	Problematic function	Error description
CP1	`memmove(ptr, ptr + 1, --len);`	`memmove` `malloc`	Unrestricted memory copy function. Can facilitate buffer overflow conditions and other memory mismanagement situations.
CP2	`goto account_no;`	`goto`	Use of the `goto` construct. The `goto` construct can result in unstructured code that is difficult to maintain and can result in failures to initialize or de-allocate memory.
CP3	`file2 = fopen("tempfile.txt", "rb");`	`fopen`	Unsafe temporary file allocation.The application appears to build a temporary file with a static, hard-coded name. This causes security issues in the form of a classic race condition (an attacker creates a file with the same name shared between the application's creation and attempted usage) or a symbolic link attack where an attacker creates a symbolic link at the temporary file location.
CP4	`rewind(fp);`	`rewind` `fopen`	The `rewind` function is considered unsafe and obsolete. Using `rewind` makes it impossible to determine if the file position indicator was set back to the beginning of the file, potentially resulting in improper control flow. `fseek` is considered a safer alternative.
CP5, 6, & 11	Buffer overflow (See the next section)		
CP7	`fmeds=fopen("Medicines.txt","r");`	`fopen`	Function used to open a file. Carry out a manual check to ensure that the user cannot modify filename for malicious purposes and that the file is not 'opened' more than once simultaneously.
CP8	`new_node = (struct node *)malloc(sizeof(struct node));`	`malloc` `fopen`	Potential memory mismanagement. Variable name: `new_node` malloc without free.
CP9	`ft=fopen("temp","wb+");`	`fopen`	Unsafe temporary file allocation. The application appears to build a temporary file with a static, hard-coded name. This causes security issues in the form of a classic race condition (an attacker creates a file with the same name shared between the application's creation and attempted usage) or a symbolic link attack where an attacker creates a symbolic link at the temporary file location.
CP10	`rewind(fp);`	`rRewind` `fopen`	The `rewind` function is considered unsafe and obsolete. Using `rewind` makes it impossible to determine if the file position indicator was set back to the beginning of the file, potentially resulting in improper control flow. `fseek` is considered a safer alternative.
CP12	`layout->keysymdefs = realloc(layout->keysymdefs, sizeof(keysymdefs) * (i + 1));`	`realloc` `malloc`	Potential memory leak. On failure, the `realloc` function returns a NULL pointer but leaves memory allocated. The code should be modified to free the memory if NULL is returned. Dangerous use of `realloc`: the source and destination buffers are the same. A failure to resize the buffer will set the pointer to NULL, possibly causing unpredictable behavior.

```
add_invalid:
	printf("\n\n\n\t\tEnter 1 to go to the main menu and 0 to exit:");
	scanf("%d",&main_exit);
	system("cls");
	if (main_exit==1)
		menu();
	else if(main_exit==0)
		close();
	else
	{
		printf("\nInvalid!\a");
		goto add_invalid;
	}
```

Fig. 1. Sample of `goto` procedure in CP2

TABLE IV. MAIN ERRORS OF BUFFER OVERFLOW REPORTED BY VCG TOOL

Subject no.	Problematic LOC	Problematic function	Error description

CP1	`strncpy(dp->host, "localhost", MAXSBUF);`	`strcpy`
		`sscanf`
		`memmove`
		`gets`
		`strncpy`
			The function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions. While "safer", the current "n" functions include non-null termination of overflowed buffers and no error returns on overflow.

CP2	`scanf("%d/%d/%d",&add.deposit.month,&add.deposit.day,&add.deposit.year);`	`scanf` `fscanf`	The function directs user defined input to a buffer and so can facilitate buffer overflows.
CP3	`strcpy(item.product_code, code);`	`scanf` `strcpy`	As CP1
CP4	`scanf("%d",&i);`	`scanf` `gets`	As CP2
CP5	`scanf("%d",&c);`	`Scanf`	
CP6	`scanf("%d",&choice);`	`scanf` `strcpy`	
CP7	`gets(c1.name);`	`gets` `scanf`	
CP8	`scanf("%f", &new_node-> university_current_result);`	`gets` `scanf`	
CP9	`scanf("%ld",&p.mble_no);`	`Scanf`	
CP10	`gets(e.duration);`	`gets` `scanf`	
CP11	`scanf("%d",&a);`	`scanf`	

CP12	`#define strtok_r(line, delim, pointer) strtok(line, delim`	`strtok`
		`sscanf`
		`strcpy`	

TABLE V. CHECK TYPES FOR INPUT VALIDATION

#	Check type	Description
1	Range checks	Inputs may be within a particular range. For example, any ratio should be between 0.0 and 1.0; the grade of student should be within the range 0 to 100, the date should be legal (e.g., not February 31st), and so on.
2	Size checks	Inputs are expected to be a given number of characters or upper limit. For instance, an employee ID should be represented with 10 integers, no name with more than 40 characters including family name, no address with more than 100 characters, and so on.
3	Format checks	Inputs may be of specific types; if a number is expected, no alphabetic characters should be allowed. For example, email address should include @ sign, the person's name must be alphabetic with no numbers or punctuation (apart from a hyphen) allowed, and so on.
4	Semantic checks	This check concentrates on the meaning of inputs. As an example, the reading of a household electricity meter should not be so far from that in the corresponding duration in the past year because it is known that the amount of electricity used is expected to be approximately the same.

V. THREATS TO VALIDITY

Here, we present two threats to the validity of the study's results.

- Internal validity: Individual developers would probably carry out the same (insufficiently robust) practices in all programs. This variable was controlled by selecting programs that were written by different developers. In addition, the selection of subjects was arbitrary. The expected threat to internal validity, if there is one, may come from errors in the VCG tool.
- External validity: The 12 OSS projects are chosen from different domains to minimize the effect of domain-specific issues. Two factors may affect the interpretation and reduce the generality of the results; studying the input validation practices of developers from 12 OSS projects may not be sufficient, and all projects considered herein are OSS, i.e., not representative of all industrial domains.

VI. CONCLUSION AND FUTURE WORK

An indicator of secure source code quality is input validation. It is believed that good practices improve program protection, which directly affects recoverability and reliability. In particular, it helps reduce the chances of producing security vulnerabilities in software. This paper conducts an experiment to identify the input validation vulnerabilities in programs and the problematic functions that lead to such issues. The experiment assessed 12 OSS projects written in C, a widely-used PL that provides input validation functions and constructs. The projects have different authors. The results show that buffer overflow (or XSS) and potential memory mismanagement are the top two most common input validation problems. The two types of functions that caused the buffer overflow problem are (a) strings/text functions such as `strcpy` and `strcmp`, and (b) functions that read from standard input, `STDIN`, such as `scanf` and `gets`. In contrast, the functions that caused the memory mismanagement are threefold: (a) memory allocation/deallocation functions such as `memmove` and `malloc`, (b) file manipulation functions such as `fopen` and `fseek`, and (c) the `goto` construct used in handling input errors or exceptions. Two main recommendations are discussed: (a) programmers are encouraged to use memory-safe PLs. Otherwise, they should perform different types of checks for the validity of inputs as soon as they are entered (four checks are presented in this paper), and (b) in addition they should have the required knowledge of secure source code and should be able to use tools/suites for malformed strings. The results may not be very surprising for skilled C developers, but it is important that there is experimental evidence about the use of a set of C functions and constructs.

There is an open point for further research to examine the problems of using different mechanisms for (a) more than 12 software projects, and (b) real-world systems (not only OSS). However, in the second mechanism, there might be a "data scarcity" research problem due to a lack of sufficient data [14].

REFERENCES

[1] I. Sommerville, Software Engineering, Pearson, UK, 2015

[2] L. Braz, E., Fregnan, G. Çalikli, A. Bacchelli, "Why don't developers detect improper input validation? drop table papers", The IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, 22-30 May 2021, DOI: 10.1109/ICSE43902.2021.00054

[3] V. D. Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future', in: Balzarotti, D., Stolfo, S.J., Cova, M. (eds) 'Research in Attacks, Intrusions, and Defenses RAID'. Lecture Notes in Computer Science, Volume 7462. Springer, Berlin, Heidelberg, 2012.DOI: https://doi.org/10.1007/978-3-642-33338-5_5

[4] T. Scholte, R. William, B. Davide, and K. Engin, "Preventing input validation vulnerabilities in web applications through automated type analysis", The Proceeding of the 36th International Conference on Computer Software and Applications, Izmir, Turkey, pp. 233-243, 16-20 July 2012, DOI: 10.1109/COMPSAC.2012.34

[5] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps", The IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18-21 May 2020, DOI: 10.1109/SP40000.2020.00072

[6] V. Tirronen, "Stopping injection attacks with code and structured data', in M. Lehto, & P. Neittaanmäki (Eds.), Cyber security: power and technology, 2018, pp. 219-231. Springer. Intelligent Systems, Control and Automation: Science and Engineering, 93. DOI: https://doi.org/10.1007/978-3- 319-75307-2_13

[7] G. E. Rodr´ıguez, J. Torres, P. Flores, D. E. Benavides, "Cross-site scripting (XSS) attacks and mitigation: a survey', Computer Networks, Volume 166, 15 January 2020, 106960. DOI: https://doi.org/10.1016/j.comnet.2019.106960

[8] J. D. Pereira, N. Ivaki, and M. Viera, "Characterizing buffer overflow vulnerabilities in large C/C++ projects', IEEE Access, vol. 9. 2021, DOI: 10.1109/ACCESS.2021.3120349

[9] O. I. Khalaf, M. Sokiyna, Y. Alotaibi, A. Alsufyani, and S. Alghamdi, "Web attack detection using the input validation method: DPDA theory", Computers, Materials & Continua, vol. 68, no. 3, 2012, DOI:10.32604/cmc.2021.016099

[10] S. A. Ebad, A. A. Darem, and J. H. Abawajy, "Measuring software obfuscation quality–a systematic literature review", IEEE Access, vol. 9, 2021, 99024–99038.

[11] E. W. Dijkstra, "Goto statement considered harmful', Communications of ACM: Letters to the editor, 1968, vol. 11, no. 3, pp. 147–148.

[12] M. Nagappan, R. Robbes, Y. Kamei, É, Tanter, S. McIntosh, A. Mockus, A.E. Hassan, "An empirical study of goto in C code from GitHub repositories', The Proceedings of the ESEC/FSE'15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Bergamo, Italy, Aug. 30- Sep. 4, 2015, DOI: https://doi.org/10.1145/2786805.2786834

[13] A. Rashid, H. Chivers, G. Danezis, E. Lupu, A. Martin, "The Cyber Security Body of Knowledge", CyBOK Version 1.0. 2019. [Online]. Available: https://www.cybok.org/media/downloads/CyBOK-version-1.0.pdf, accessed 20 Jan 2023.

[14] M. Taeb, and H. Chi, "A personalized learning framework for software vulnerability detection and education", International Symposium on Computer Science and Intelligent Controls (ISCSIC), 12-14 November 2021, Rome, Italy. DOI: https://doi.org/10.1109/ISCSIC54682.2021.0003

[15] S. A. Ebad, "An exploratory study of ICT projects failure in emerging markets", Journal of Global Information Technology Management, vol. 21, no. 2, 2018, pp. 139-160. DOI: https://doi.org/10.1080/1097198X.2018.1462071