Open Access Library Journal
2024, Volume 11, e11491
ISSN Online: 2333-9721
ISSN Print: 2333-9705

Defence against Command Injection Attacks in a Distributed Network Environment

Oluwatobi Akinmerese[1], Samuel Fasanya[2], Daniel Aderotoye[3], Ngozichukwuka Adingupu[1], Evelyn Ezeoke[1], Rukayat Muritala[4], Oyindamola Lawal[4], Blessing Akingbade[4], Chiamaka Ifekandu[5]

[1]Department of Cybersecurity, Augustine University, Lagos, Nigeria
[2]Department of Information Security, Osun State University, Osun, Nigeria
[3]Centre for Space Research and Applications, Federal University of Technology, Akure, Nigeria
[4]Department of Library and Information Science, Tai Solarin University of Education, Ogun, Nigeria
[5]Department of Computer Science, Augustine University, Lagos, Nigeria
Email: oluwatobi.akinmerese@augustineuniversity.edu.ng, fasanyasam@yahoo.com, aderotoyeda@futa.edu.ng, adingupungozi@gmail.com, oge.eviey@outlook.com, rhukkissmart09@gmail.com, rhukkissmart09@gmail.com, bleakingbade@gmail.com, thebossphoebe@gmail.com

How to cite this paper: Akinmerese, O., Fasanya, S., Aderotoye, D., Adingupu, N., Ezeoke, E., Muritala, R., Lawal, O., Akingbade, B. and Ifekandu, C. (2024) Defence against Command Injection Attacks in a Distributed Network Environment. *Open Access Library Journal*, **11**: e11491.
https://doi.org/10.4236/oalib.1111491

Received: March 26, 2024
Accepted: May 28, 2024
Published: May 31, 2024

Abstract

Regardless of the programming language used to create the application or the operating system on which it runs, command injection is common in all applications. Command injection attacks can result in a variety of consequences, such as compromised data confidentiality and integrity or unapproved remote access to the system hosting the susceptible application. The recently found Shellshock flaw is a perfect example of a real, notorious command injection vulnerability that demonstrates the dangers of this kind of code injection. The research community has not paid much attention to the type of code injection, despite the fact that command injection assaults are common and have a significant impact. To the best of our knowledge, no specific software program exists that can automatically identify and take advantage of command injection attacks, unlike those caused by SQL injection or cross-site scripting [1]. This study aims to close this gap by presenting COMMIX, an open-source tool that automates the process of finding and taking advantage of web application command injection vulnerabilities (COMMand Injection eXploitation). To address scenarios of serial exploitation, this tool offers a wide range of functions. Additionally, commix has a high success rate in determining whether a web application is susceptible to command injection attacks. Ultimately, we have identified multiple 0-day vulnerabilities in applications during the tool review process. The work's overall contributions include offering a thorough analysis and classification of command injection attacks; describing and evaluating our open-source tool that automates the process of identifying; and taking advantage of command injection vulnerabilities that are found on a variety of web-based applications, ranging from web servers to home services (embedded devices).

Subject Areas

Cloud Computing

Keywords

SQL Injection, Shell Command on Unix-Based Systems, Operating System, Input Validation, Web Vulnerability, COMMIX

1. Introduction

1.1. Context of the Research

The term "command injection" refers to a broad category of attack methods wherein specific code is injected and the vulnerable (web) application then executes it [2]. According to the 2013 OWASP Top Ten Web Security Risks, this kind of attack has been identified as a major security issue and is ranked first [3]. An attacker can introduce arbitrary code or commands into an application through a code injection vulnerability, which takes advantage of careless handling of untrusted data and causes unexpected execution behaviour. Code injection attacks come in a variety of forms, such as SQL injections, LDAP injections, XPATH injections, and cross-site scripting assaults [1] [2] [4]. We will just address command injection attacks in this paper [2] [4]. The objective of a command injection attack, according to OWASP, is to use a vulnerable application to execute arbitrary commands on the host operating system. The main cause of command injection attacks is inadequate input validation [2] [4].

Because these attacks take place when an application invokes the operating system shell (command prompt shell on Windows, shell command on Unix-based systems), they are also referred to in the literature as "shell command injection" or "OS (OPERATING SYSTEM) command injection" [2] [4]. We will simply refer to these assaults as "command injection" in this third section [2].

1.2. Problem Definition

A web application is a type of software that operates on a web server. The majority of organizations, including small businesses and schools, employ web applications because they provide as a conduit for potential customers to contact the business. If these online applications are not adequately secured, information may be stolen, the web server may be infiltrated, system files may be corrupted, and an attacker may replace legitimate data with fake data. If online security is not correctly handled, command injection can be used to accomplish all of them [2] [3]. Below are some of the common problems that this research poses to help solve:

Information Theft: Web applications can be vulnerable to attacks that steal user data or other sensitive information.

Web Server Infiltration: Hackers can gain unauthorized access to the web server that runs the web application.

System File Corruption: Malicious actors can corrupt system files, causing the web application or underlying system to malfunction.

Data Manipulation: Attackers can tamper with data stored within the web application.

1.3. The Study's Aims and Objectives

The purpose of this project is to utilize an appropriate defense for your web application to stop attackers from injecting commands.

These are the goals that:

i) To determine which online applications are susceptible to command injection attacks.

ii) To develop a secure web application that requests user input and verifies the accuracy of the data supplied.

iii) Adding permitted input to a white list to prevent command injection.

1.4. Techniques

This project's methodology, which consists of 5 steps, is presented here. constructing a website that is vulnerable or utilizing a website that is vulnerable and accepts user input, such as ping; configuring a preferred user and tester interface; or utilizing a server-side scripting language (such as Python, PHP, etc.) that acts as a bridge to the terminal. Any tool can be used to test for vulnerabilities in a web application; however, the Commix tool is suggested for this project. Comix tools are useful because they are dependable, adaptable, and a command injection dictation tool for all time.

1.5. Methods

This document outlines the five stages of the methodology used to carry out this project: creating or utilizing a susceptible website that takes user input, such as ping; configuring a preferred user and tester interface; or utilizing a server-side scripting language (such as PHP, Python, etc.) that acts as a bridge to the terminal Tools are used to test web applications for vulnerabilities; any tool can be used, however, the Commix tool is suggested for this project. Because Comix Tools are reliable, adaptable, and an ever-present command injection dictation tool, they are an important tool for work.

1.6. Procedure

The project was conducted in five steps, which are shown here as the approach used: establishing a website that is vulnerable or utilizing one that is insecure and accepts human input, such as ping; configuring a desired user interface for testers; or employing a server-side scripting language (such as PHP, Python, and so on) that acts as a bridge to the terminal. A web application's vulnerabilities can be tested using a variety of tools; however, the Commix tool is suggested for this project. Due to its stability, flexibility, and constant command injection dictation capabilities, Comix Tools are an indispensable tool for work.

1.7. Approaches

This project's methodology, which consists of five steps, is presented here: creating a website that is vulnerable or utilizing a website that is vulnerable and accepts user input, such as ping; configuring a preferred user and tester interface; or utilizing a server-side scripting language (such as Python, PHP, etc.) that acts as a bridge to the terminal. Any tool can be used to test vulnerabilities in a web application; however, the Commix tool is suggested for this project. Comix Tools is a useful tool since it is a command injection dictation tool that is always reliable, adaptable, and current.

1.8. Importance of the Research

The primary goal of the research is to use appropriate tools to identify command injection vulnerabilities in the Web program. Commix has been used for the project's tools, which include exploitation and command injection tools. Many functionalities are supported by this tool to cover a variety of exploitation scenarios. Additionally, Commix has a high success rate in determining whether a web application is susceptible to command injection assaults. The Python-written program operates on both Windows and Unix-based operating systems. Commix can be downloaded for free from the GitHub site.

1.9. The Study's Scope and Limitations

This study focuses on web applications that are created and overseen by Augustine University's IT administrator and distributed to faculty and staff.

There are limitations. Financial constraint: The implementation costs would be relatively significant if it were to be applied to every employee at Augustine University. Not enough time to put into practice.

2. Description of the Core Project Topic Concepts

The literature review explains what it takes to protect against command injection attacks in a distributed network environment [2] [4]. It also clarifies testing vulnerabilities and provides solutions for the issue. It functions essentially as a basis for a deeper understanding of the project.

Procedure for testing:

Step 1: Recognize potential attacks

Step 2: Examine the reasons and preventative actions

Step 3: Begin experimenting and examining

Step 4: Special cases

2.1. Recognize Attack Scenarios

Understanding and grasping the attack method is the first step in testing for command injection vulnerabilities [2]. There are two typical kinds:

a) Direct command injection: Giving more commands to the susceptible application directly is the most popular type of command injection [2]. By directly supplying user-supplied data as an argument to the command, the attacker first determined whether the program is vulnerable. If so, the malicious command is sent by the attacker as the expected argument. The program runs the first command, then the harmful one.

b) Indirect command injection: in this scenario, the additional command is supplied to the susceptible application indirectly, either via a file or an environment variable [2].

2.1.1. Consider Causes and Improve Method

Input validation is the only source of command injection problems [2]. The development community must search for all instances in which the application invokes a shell-like system function, such as exec or system, and refrain from executing them until the parameter has been properly validated and sanitized. Applications that construct command strings using non-sanitized data are susceptible to this type of bug. Using a black list or a white list are the two methods available for validating three parameters.

2.1.2. Get Testing and Researching Started

To begin with, you must locate every location where your application calls upon a system command in order to function. Then begin investigating how the program handles the fundamental characters required for command insertion on each of these locations. Experts have created a number of test tools and platforms that can assist developers in investigating and testing vulnerabilities. Two typical examples are the commix and burp suite [5].

An essential tool for conducting web application security testing is the Burp suite. Its many tools operate in unison to facilitate every step of the testing process, from the first mapping and analysis of the application attack surface to the identification and exploitation of security flaws [3].

2.1.3. Start Your Research and Testing

To start, in order for your application to run, you need to find every place where it uses a system command. Then start looking into how the software handles the basic characters needed to insert commands on each of these spots [6]. Many test platforms and tools have been developed by experts to help developers find and test vulnerabilities. The commix and burp suite are two common examples [7].

The Burp suite is a vital tool for performing security testing on online applications [3]. Together, its numerous tools make testing easier at every stage, from the initial mapping and analysis of the application attack surface to the detection and exploitation of security vulnerabilities [3]. It was created by Portswigger Web Security and is Java-based [3].

2.1.4. Perfect Cases

You must cover every potential entry point and scenario in order to fully test your application against command injection issues [2]. Keep investigating the various application entrance points; the test case data format will change based on the entry point [8]. For instance, depending on the URL encoding, the string file.txt "|dir c: can appear like one of the two below if you are testing through the URL:

*File.txt"|dir%20c: * File.txt "|dir+c: Source

For additional command injection entry points like input fields, URL parameters, POST data, web service methods, environment variables, database contents, registry contents, file contents, third-party APIs, and network packets, it is crucial that you take different encoding and data formats into consideration [2].

2.2. An Explanation of the Core Project Topic Concept's Existing Defense Method

No longer is it a concern for site developers to have an effective and trustworthy defensive mechanism. Because OWASP, an international non-profit organization devoted to web application security, has uncovered new modules, platforms, and tools over the years—some of which were stated in section 2.1.3—in the course of their research [3] [9].

Owasp Primary Defence Methods

Steer clear of directly calling OS commands:

Avoiding directly calling OS commands is the main result. Because built-in library functions can only be used for the purposes for which they were designed, they are a great substitute for OS commands. Use mkdir (), for instance, rather than system ("mkdir_name"). This is the recommended approach if the language you choose has libraries or APIs accessible [10].

a) Using parameterization and input validation together

In the event that it is not possible to avoid calling a system command that uses user-supplied data, software should employ the next two layers of security to fend off attacks:

First Layer: Parameterization

Use structure mechanisms, if they are available, as they can assist in automatically enforcing the division between data and commands [8].

Layer 2: Validation of input

The relevance argument and the command values should both be verified; the command and its argument have varying levels of validation.

- When it comes to command use4, they have to be verified against an authorized command white list.

- The following parameters should be used to validate the arguments given for this command.
- Positive or white list input validation: specifies exactly where an argument is allowed.
- White list regular expression: this defines the maximum string length as well as a white list of acceptable characters.

PLUS DEFENSE

We advise implementing all of these extra defenses in addition to the fundamental defenses of parameterization and input validation. These are:

The application should operate with the least amount of privilege necessary to do the task at hand.

If at all possible, establish a single-task isolated environment with restricted privileges.

2.3. A Comparison of the Core Project Topic's Existing Types Ideas

An attacker can submit various forms of input to a program using a variety of attack techniques known as injection attacks [2] [4]. The CPU then interprets the input, treating it as a query or command, and executes it, producing inaccurate results [11]. The attacker could then cause your website to fail or steal your private information.

The following categories of injection attacks:

- Injection attacks using code: Application code written in the application language is injected by the attacker [2] [4]. This code could be used to run operating system commands with the user's privileges when the web application is executing. In advance, the attacker might take advantage of further privilege escalation flaws, which could result in a complete web server (See **Table 1**).

Table 1. A tabular format review of all related project work.

Year and Author	Title And Source	Assistance	Issue Resolved
[12]	Identifying and averting attacks using code injection. Scholar Commons at University of South Florida	A technique that prevents CIAOs through optimization (for applications where the program language output has an LR (K) grammar where each closed value corresponds to a syntactic category)	Attacks using code injection
[12]	Thorough analysis of the research on SQL injection attacks, 2016(26–35), International Journal of Soft Computing 11(1), ISSN: 1816–9503.	A systematic literature review (SLR) on SQL injection attacks was conducted using the kitchen hams technique.	SQLI identification and mitigation
[13]	Injection of commands. University of Ottawa's School of Information Technology and Engineering, Ontario, Canada, kin 6N5	An overview of typical injection attacks	a list of frequent injection attacks

Continued

[14]	Evaluation of statistical tools for identifying security holes in the source code of Java and C/C++ programs at the University of Ontario Institute of Technology's Department of Electrical, Computer, and Software Engineering, Oshawa, ON, Canada.	The source codes for C/C++ and Java are examined for errors and vulnerabilities.	Assessing the instruments used for code analysis.

The method or framework used to carry out this investigation is presented in this Chapter. This chapter focuses on the methodical examination of the current or proposed system to ascertain the information needs and procedures of the system, as well as the definition of hardware and software architecture components and interfaces to meet the necessary specifications [6].

3. Existing System Analysis

There are currently automated solutions for web application security testing that audit your website by looking for vulnerabilities [3]. Generally speaking, the system searches any webpage or web application that supports the HTTP/HTTPS protocol and can be accessed by a web browser. With a powerful crawler that can locate practically any file, the system provides a robust and distinctive solution for assessing custom and off-the-shelf web applications, including those that use Java Script, AJAX, and web 2.0 apps.

3.1. Work of the System

i) The system analyzes every link on the website, including those created dynamically with JavaScript and those included in site marks. xml and robot. txt files (if accessible).

ii) It features sensor technology, which will obtain a list of every file present in the web application directories and send any files the crawler could not find to the crawler output. Typically, the crawler misses these files because they are either not accessed from the web server or do not link through the website.

iii) The scanner automatically starts a series of vulnerability checks on every page it finds after the crawling process. Additionally, the algorithm scans every page for locations where it can enter data and generate reports, developer reports, and different compliance reports like ISO 270001 or PCIDSS

3.2. The Goal of the Current System

The automated Scan Stage is this:

i) The vulnerability identification is displayed by the scan outcome. Every vulnerability warning includes details regarding the vulnerability, including POST, data utilized, impacted item, server HTTP response, and more ii. Details like source code, line number, stack trace, or impacted SQL query that caused the vulnerability are listed if sensor technology is being used [1].

Attacks include file inclusion, directory traversal, code execution, command injection, CRLF injection, file inclusion, input validation, and authentication [2] [4].

- The following are examples of attacks: input validation, authentication, file inclusion, directory traversal, code execution, command injection, and CRLF injection [2] [4].

Input validation, authentication, file inclusion, directory traversal, code execution, command injection, and CRLF injection are a few instances of threats [2].

3.3. Suggested Workflow for System Design

i) The attack module: As its name suggests, the attack vector generator module creates a collection of command injection attacker vectors [2].

ii) The vulnerability detection module: this module generates a set of distinct attack vectors for each type of attack, which are then passed to the vulnerability detection module. The command injection separator list and the types of command injection that will be executed (classic, dynamic code evaluation, time-based, and file-based) are the sources of this information [2].

iii) The exploitation module: COMIX instructs the exploitation module to attempt automatic exploitation if the vulnerability module detection finds that the application is susceptible (See **Figure 1**).

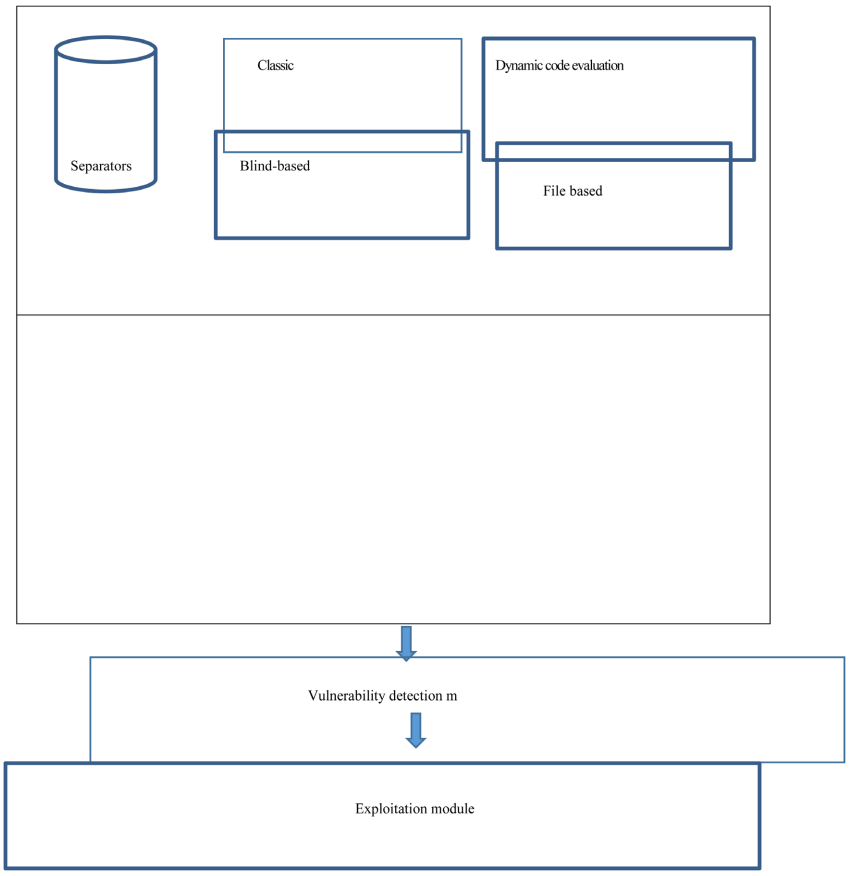

Figure 1. Commix steps [3].

4. System Implementation and Evaluation

Executing a design is the process of implementation. In terms of the project, implementation entails coordinating Commix to be used to properly test and exploit as well as a guide on how to validate input going into the computer. It also involves installing a physical system and the processes involved in putting the design to work efficiently.

4.1. The Tool Option

PHP

This language for server scripting is an effective tool for creating dynamic and interactive webpages. PHP is the scripting language used to create the web application that will be used as an example to eat.

Python

Python is different from HTML, CSS, and Javascript in that it is a general-purpose programming language. In addition to web development, it may be utilized for other kinds of programming and software development [6]. This covers, among other things, scripting scripts, data science, software development, and back-end development.

A free and open-source cross-platform web server solution stack package is called XAMPP.

It was created by friends of Apache and consists mostly of MariaDB, interpreters for PHP and PERN scripts, and Apache HTTP servers.

Code Visual Studio

Microsoft created this free source code editor for Windows, Linux, and Mac OS X. Debugging, syntax highlighting, code rewriting, snippets, and embedded Git are all supported. It is capable of editing different codes written in multiple languages.

Mixing Tool

With the help of this automated, all-in-one OS command injection and exploitation tool, it is very easy to identify and take advantage of command injection vulnerabilities in specific HTTP headers or vulnerable parameters [2].

Commix-Test-bed

This is a set of test web pages for commix vulnerability detection and exploitation that are susceptible to command-injection vulnerabilities [2]. Clone the official Git repository by bit cloning
"https://github.com/commix-project/commix-test-bed git" to get it (commix-test bed).

4.2. System Demands

These are the necessary specifications for the system. To efficiently use a given system, a device must have. These prerequisites consist of:

- Hardware specifications
- Requirements for software

4.2.1. Devices to the Hardware

The physical parts that the device needs in order for the system to function properly are known as hardware requirements. The following hardware is necessary for this system to operate effectively:

- CPU: Pentium 133MHz or higher;
- RAM: at least 1GB;
- 180MB of free hard disk space.

4.2.2. Devices to the Software

Software requirements are the applications that must be installed on the device in order for the system to function.

Operating System: Commix supports Linux and macOS. It can also run on Windows with Cygwin or the Windows Subsystem for Linux (WSL).

Python: Commix is written in Python and requires Python 2.6.x or 2.7.x to run. However, it's recommended to use Python 2.7.x as it's the most compatible version.

Dependencies: Commix relies on various Python libraries, such as requests, argparse, and lxml. These dependencies can typically be installed via the Python Package Index (PyPI) using pip.

Internet Connection: Commix may require an internet connection for certain features, such as fetching payloads or accessing remote resources during exploitation.

There are two phases to the system's implementation.

i) The testing phase; ii) The web application validation phase.

4.2.3. Phase of Testing

We chose PHP as the server-side language for this testing phase for the following reasons:

i) PHP is a well-liked server-side web application programming language that is utilized by many content management systems (e.g. Word Press, Drupal, etc.).

ii) Setting up a development environment is simple and cost-free.

iii) We can evaluate the security of multiple open-source PHP projects that are accessible in public repositories.

A lite-web app called "normal PHP" was created and a test-bed called "commix-test bed" was utilized for the actual testing.

4.2.4. Application Validation Stage (Web)

Specifically, the process of filtrating—that is, removing unwanted characters from the input data is referred to as input validation.

At this point, web pages and security are given more importance. It is advised to use appropriate input validation and sanitation to control what types of commands are entered into the computer and run through "echo."

The following crucial text hygiene and validation measures are likely to stop command injections:

- No pipe, no colon.

- No dollar sign, no ampersand.
- System compromise and data leaks via the internet not using alphanumeric characters, nested quotes, or special characters.

Additionally, developers should be aware of any situations in which a program calls for the execution of an OS command, such as "exec()" or "system()," and refrain from doing so unless authorized beforehand.

The parameter has either been successfully escaped or validated. Input validation should be carried out correctly by using two distinct methods:

(a) Whitelisting; (b) Blacklisting.

Additionally, developers should use the programming language's APIs to escape input data:

i) Blacklisting methodology.

ii) The whitelisting method.

iii) Data escape.

5. Summary

Web-based applications are essential in helping people in the twenty-first century complete tasks that can occasionally be highly difficult and time-consuming.

It is for this reason that web application security is crucial. An insecure online application may cause data leaks or illegal computer access.

5.1. Recommendation

Web developers now have a platform to learn about command injections and how to defend against them, thanks to this initiative. To guarantee seamless deployment and robust security for the school web application that is uploaded on a server, it is advised that the IT department engage in the practices outlined in this project and also involve the staff and students in his practices.

5.2. Conclusion

Because web applications are now the primary means of communication between clients and service providers, skilled hackers target their targets for financial or personal advantage.

5.3. Contribution to Knowledge

By enabling web developers to test and exploit these command injection problems, this work will improve web security because web developers should be able to recognize future vulnerabilities that could endanger data and OS.

Acknowledgements

I am greatly indebted to God Almighty for His grace and mercy that saw me through to the crucial attainment of achieving the aim of writing this article alongside all other authors. I cannot but praise His Holy Name for the unusual favour, spirit of understanding, and inspiration we received from Him, even when we did not believe we could pursue this article title to attain this level of success. Blessed be His Holy Name forever and ever.

I am particularly grateful to my article editors, Professor O. Awodele who would stop at nothing to insist on the best of the best. I immensely appreciate Professor C. Ogbonna for his fatherly advice, mentorship and contributions to my work which was remarkably a compass to the successful direction of the study. I am indebted to all my lecturers in the Department of Cybersecurity and other departments of the University from whose wealth of knowledge I benefited immensely. I am also grateful to the founding fathers of Augustine University, Ilara-Epe, Lagos, Nigeria, for this great institution that has sharpened my academic prowess.

I cannot but also be grateful to my academic mentors, senior colleagues, and friends who were in no small measure instrumental to my success during the course of writing this paper. These associates include Dr. C. Okunnbor, Dr. J. Akinsola, Dr. O. Kalesanwo, Dr. C. Ajaegbu, Prof. A. Simpson, Dr. D. Aleburu, Mrs. A. Mamza, Dr. C. Ogu, Prof. M. Eze, Mr. A. Oyebode, Mr. O. Alowosile, Mr. O. Blaise, Dr. O. Abiodun, Dr. O. Ebiesuwa and Dr. A. Omotunde

My undiminished gratitude goes to my parents, Pastor F. Akinmerese and Mrs. M. Akinmerese for bringing me forth and training me to be bold and courageous, and to go for the best education I can get. My gratitude goes to my brothers’ in-law, Mr. O. Ayodele and Mr. O. Olagbemiro and all my siblings Mrs. O. Ayodele and Mrs. F. Olagbemiro for their support all the time. My heartfelt appreciation goes to my wife—Mrs. E. Akinmerese who has proved herself a worthy pillar of support since I met her. Her steadfast perseverance, care, understanding and support kept me moving. Words cannot express my gratitude to my children E. Akinmerese, T. Akinmerese and O. Akinmerese, may God bless them all abundantly.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] Ali, S., Rauf, A. and Javed, H. (2019) SQLIPAI: An Authentication Mechanism against SQL Injection. *Journal of Scientific Research in Europe*, **38**, 604-611. https://www.academia.edu/9892425/SQLIPA_An_Authentication_Mechanism_Against_SQL_Injection

[2] Meijer, E. and Schulte, W. (2017) Unifying Tables, Objects and Documents. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cdcd6b67a0b7ea867bb43e5fbb36679ad936b8cc#page=149

[3] Prokhorenko, V., Choo, K.-K.R. and Ashman H. (2016) Web Application Protection Techniques: A Taxonomy. *Journal of Network and Computer Applications*, **60**, 95-112. https://www.sciencedirect.com/science/article/pii/S1084804515002908 https://doi.org/10.1016/j.jnca.2015.11.017

[4] Scott, D. and Sharp, R. (2003) Establishing and Putting into Practice Application-Level Web Security Policies. *IEEE Transactions on Knowledge and Data Engi-*

neering, **15**, 772-783. https://doi.org/10.1109/TKDE.2003.1208998

[5] Aho, A.V. (1986) Compilers, Principles, Techniques, and Tools. https://archive.org/details/compilersprincip0000ahoa

[6] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J. and Evans, D. Automatically Hardening Web Applications Using Precise Tainting. https://link.springer.com/content/pdf/10.1007/0-387-25660-1_20.pdf

[7] Ray, D. (2013) Identifying and Averting Attacks Using Code Injection. Master's Thesis, University of South Florida, Tampa. https://digitalcommons.usf.edu/cgi/viewcontent.cgi?article=5763&context=etd

[8] Lawal, M.A., Sultan, A.B.M. and Shakiru, A.O. (2006) Systematic Literature Review on SQL Injection Attack. *International Journal of Soft Computing*, **11**, 26-35. https://www.researchgate.net/publication/282377809_Systematic_literature_review_on_SQL_injection_attack

[9] Guy-Vincent J. Injection of Commands. University of Ottawa's School of Information Technology and Engineering, Ontario, Canada. https://site.uottawa.ca/~gvj/Courses/CSI4539-OLD/lectures/CommandInjections.pdf

[10] Qusay, H.M. and Rahman, M. Evaluation of Statistical Tools for Identifying Security Holes in the Source Code of Java and C/C++ Programs. The University of Ontario Institute of Technology's Department of Electrical, Computer, and Software Engineering, Oshawa, ON, Canada. https://arxiv.org/pdf/1805.09040

[11] Jourdan, G. (2009) Securing Large Applications against Command Injections. *IEEE Aerospace and Electronic Systems Magazine*, **24**, 15-24. https://doi.org/10.1109/MAES.2009.5161718

[12] AlBreiki, H.H. and Mahmoud, Q.H. (2014) Evaluation of Static Analysis Tools for Software Security. 2014 10*th International Conference on Innovations in Information Technology* (*IIT*), Al Ain, United Arab Emirates, 09-11 November 2014. https://doi.org/10.1109/INNOVATIONS.2014.6987569

[13] Su, Z. and Wasserman, G. (2006) The Essence of Command Injection Attacks in Web Applications. *ACM SIGPLAN NOTICES*, **41**, 372-382. https://doi.org/10.1145/1111320.1111070

[14] William, G., Orso, A. and Manolios, P. (2006) Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. *SIGSOFT* '06/*FSE*-14: *Proceedings of the* 14*th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland Oregon, USA, 5-11 November 2006, 175-185. https://doi.org/10.1145/1181775.1181797