Spring 2023

OWASP ZAP vs Snort for SQLi Vulnerability Scanning

Christopher Kalaani

OWASP ZAP VS SNORT FOR SQLI VULNERABILITY SCANNING

by

CHRISTOPHER KALAANI

(Under the Direction of Lei Chen)

ABSTRACT

Web applications are important to protect from threats that will compromise sensitive information. Web vulnerability scanners are a prominent tool for this purpose, as they can be utilized to find vulnerabilities in a web application to be rectified. Two popular open-source tools were compared head-to-head, OWASP ZAP and Snort. The performance metrics evaluated were SQLi attacks detected, false positives, false negatives, processing time, and memory usage. OWASP ZAP yielded fewer false positives and had less processing time. Snort used significantly fewer memory resources. The internal workings of ZAP's Active Scan feature and Snort's implementation of the Boyer-Moore and Aho-Corasick algorithms were identified as the main processes responsible for the results. Based on the research, a set of future working recommendations were proposed to improve web vulnerability scanning methods.

INDEX WORDS: Snort, OWASP ZAP, SQL injection, Web vulnerability scanning

OWASP ZAP VS SNORT FOR SQLI VULNERABILITY SCANNING

by

CHRISTOPHER KALAANI

B.S., Georgia Southern University, 2020

M.S., Georgia Southern University, 2023

A Thesis Submitted to the Graduate Faculty of Georgia Southern University
in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

OWASP ZAP VS SNORT FOR SQLI VULNERABILITY SCANNING

by

CHRISTOPHER KALAANI

Major Professor:	Lei Chen
Committee:	Chris Kadlec
	Yiming Ji

Electronic Version Approved:

May 2023

ACKNOWLEDGMENTS

Thank you very much to the committee members – Dr. Lei Chen, Dr. Chris Kadlec, and Dr. Yiming Ji – for approving this thesis. Special thanks to and appreciation for Lei Chen and Chris Kadlec for all their assistance and guidance.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1
INTRODUCTION

1.1 Background

Web applications are becoming increasingly prominent in today's technological society. There is no shortage of sensitive information that users expect web applications to securely manage – this includes credit card numbers in e-commerce transactions, patient medical records in hospital databases, student records in school/university databases, personal photos and videos on mobile devices, and so on. As such, it is crucially important that web developers are up to date with the latest software and best practices to ensure that applications are protected from the most common cybersecurity threats that could compromise sensitive data. Open Web Application Security Project, known commonly as OWASP, is a non-profit organization that provides open-source documentation of the most prominent web application vulnerabilities. As of 2021, their top ten vulnerability list is as follows: 1) Broken Access Control, 2) Cryptographic Failures, 3) Injection, 4) Insecure Design, 5) Security Misconfiguration, 6) Vulnerable and Outdated Components, 7) Identification and Authentication Failures, 8) Software and Data Integrity Failures, 9) Security Logging and Monitoring Failures, 10) Server-Side Request Forgery (Top 10 Web Application Security Risks, 2021).

An effective and widely used means of detecting and protecting applications from these vulnerabilities is the deployment of a web vulnerability scanner (WVS). These tools are primarily used for Blackbox or penetration testing purposes, which essentially is the systematic analysis and probing of a given application's security foundations, typically undergone by white hat hackers to improve its defensive integrity (Hawamleh, 2020). For legal purposes, it is of course always advised that web vulnerability scanners are used on applications that a user either owns or has permission to scan, such as in the Blackbox testing case. How web vulnerability scanners function will be discussed in detail later, but most operate using three main modules – a crawling module, an attacking module, and an analyzer module. In the crawling module, all associated web pages and links are explored from the primary target URL. In the attacking module, the information gathered from the crawling module is used to generate various attacks against the target web application. In the analyzer module, the success rate of those attacks is processed to raise alerts about potential vulnerabilities that exist on the target web application (Amankwah, 2020). Examples of commercial and open-source web vulnerability scanners include OWASP ZAP, Skipfish, Arachni, Black Widow, VulScan, Nessus, and Acunetix, among many others.

1.2 Motivations and Aim

As explained above, it’s very important that web applications are kept as secure as possible due to the prevalence of sensitive data that exists and depends on web applications to manage. Web vulnerability scanners are useful and widely used tools for this purpose. The goal of this thesis is to conduct a comparative analysis on the performance of two existing vulnerability scanning tools that, based on current knowledge to date, have not been compared before. The contribution this work seeks to first provide is an understanding of the performance of the tools based on identifying specific methods behind how they function, and secondly, the research proposes ideas for future work that might help improve the performance of web vulnerability scanners and inspire innovation in different methods used for web application vulnerability scanning.

The two tools this paper evaluates are OWASP ZAP and Snort. ZAP was specifically chosen because it is one of the most popularly used and regularly updated open-source web vulnerability scanners free to download on the internet. The ZAP developers host open forums where the community can ask questions and learn more about how the tool works, which assisted in the research process for this thesis. ZAP also has a graphical user interface that makes the tool easy and convenient to use. Snort is also a popular and regularly updated open-source tool, but unlike ZAP, it does not function as a web application vulnerability scanner per se – rather, it is used for network intrusion detection purposes. Still, this tool was primarily chosen because there is a great deal of overlap between web vulnerability scanning and network intrusion detection, and its inclusion would make a novel comparison. There is a large existing body of literature that evaluates ZAP compared to a variety of different popularly used web vulnerability scanners, so this specific comparison with Snort helps this thesis stand out as a unique contribution to the field.

SQL injection (SQLi) attacks were focused on exclusively as a vulnerability type to assess as a metric, for three main reasons – first, SQL injection is one of the most prominent web application vulnerabilities, so further research contributions assessing detection methods for it are important. Second, there are two key algorithms in Snort that primarily serve as pattern matching functions. This makes SQLi an appropriate vulnerability type to assess using this tool, as SQLi attacks are often characterized by various kinds of string inputs. Third, it is for the sake of simplicity and specificity. While the evaluation of other injection-based vulnerabilities such as Cross Site Scripting (XSS) would make this work more comprehensive, it is outside of the scope of this thesis and is a topic for future research.

1.3 Research Questions

The three core research questions this thesis seeks to answer are the following:

RQ1) Which tool performs better or worse?

RQ2) What specific aspects of each tool can be identified to explain their performance?

RQ3) How can this information generate new research ideas for future work innovations in different methods that may improve the detection performance of web vulnerability scanners?

CHAPTER 2
LITERATURE REVIEW

This section summarizes how both ZAP and Snort performed respectively in detecting SQLi attacks, as well as other vulnerabilities, according to previous literature investigations.

2.1 ZAP

MBurano and Si conducted a performance comparison between ZAP and Arachni for two common performance metric benchmarks – OWASP and WAVSEP. They created a penetration testing environment that consisted of a LAN connecting two computers, with one computer functioning as a target and the other functioning as an attacker. Five different categories of vulnerabilities were tested – Command Injection, LDAP Injection, SQL Injection, XSS, and Path Traversal. The OWASP benchmark metrics evaluated were true positives and negatives, false positives and negatives, true positive rate percentage, false positive rate percentage, and score percentage. The results for ZAP were as follows for SQL Injection: TP = 158, FN = 114, TN = 224, FP = 8, TPR% = 58.09, FPR% = 3.45, Score% = 55. The results for Arachni were as follows for SQL Injection: TP = 136, FN = 136, TN = 227, FP = 5, TPR% = 20, FPR% = 2.16, Score% = 48. For the OWASP SQL Injection benchmark, ZAP scored 48% and Arachni scored 55%. These results were then compared with the WAVSEP score benchmark, which was 58% and 50% for ZAP and Arachni respectively with regards to OWASP, and 96% and 100% for ZAP and Arachni respectively with regards to WAVSEP (Mburano, 2018).

Makino and Klyuev compared ZAP and Skipfish for vulnerability detection. Both scanners were tested on two different web applications, which were DVWA and WAVSEP. The metrics evaluated were the total number of vulnerabilities categorized by severity, detection precision, and false positive rate percentage. The results for ZAP were as follows: On DVWA, there were 13 high vulnerabilities, 10 medium vulnerabilities, 83 low vulnerabilities, and 1 informational vulnerability for a total of 107 vulnerabilities found. On WAVSEP, there were 0 high vulnerabilities, 34 medium vulnerabilities, 2535 low vulnerabilities, and 625 informational vulnerabilities for a total of 3194 vulnerabilities found. The results for Skipfish were as follows: On DVWA, there were 7 high vulnerabilities, 2 medium vulnerabilities, 3 low vulnerabilities, and 1 informational vulnerability for a total of 13 vulnerabilities found. On WAVSEP, there were 0 high vulnerabilities, 11 medium vulnerabilities, 14 low vulnerabilities, and 353 informational vulnerabilities for a total of 378 vulnerabilities found. According to WAVSEP, the detection precision for ZAP was as follows: 100% for RXXS, 100% for SQLi, 43.2% for LFI, and 9% for RFI. For Skipfish, the detection precision is as follows: 8.2% for RXXS, 9.5% for SQLi, 1% for LFI, and 0% for RFI. The false positive rate for ZAP was 0% for all vulnerabilities, and for Skipfish it was 100% for all vulnerabilities. Based on this study, ZAP was shown to be significantly superior compared to

Skipfish for both detection precision and false positive rate percentage for all vulnerabilities (Makino, 2015).

Another paper comparing ZAP and Skipfish by Zukran and Siraj looked at SQL Injection and XSS. A penetration testing environment was set up on Kali Linux and both scanners were run on two different vulnerable web applications, DVWA and WAVSEP. Data was collected about the total vulnerabilities found categorized by severity, detection accuracy, and false positive rate percentage. The results for ZAP were as follows: on average, 3 high, 3 medium, 5 low, and 0 informational vulnerabilities were found. For Skipfish, the results were as follows: on average, 1 high, 6 medium, 4 low, and 12 informational vulnerabilities were found. For ZAP, the detection precision was 100% for both SQLi and XSS. For Skipfish, it was 8.2% for SQLi and 9.5% for XSS. ZAP reported a false positive rate of 0% for both SQL Injection and XSS, whereas Skipfish reported a false positive rate of 100% for both vulnerabilities. The vulnerability detection results for SQLi and XSS were also compared to the results of the prior Klyuev paper (5), which was as follows: ZAP found 1 SQLi and XSS vulnerability each, whereas Skipfish found 1 SQLi and 2 XSS vulnerabilities according to the results in this paper. In Klyuev et. al, ZAP found 1 SQLi and 2 XSS vulnerabilities, whereas Skipfish found 1 SQLi and XSS vulnerability each (Zukran, 2021)

Sagar and Brahma compared ZAP, Skipfish, and w3af for vulnerability scanning detection on DVWA. The testing environment consisted of Windows 10, Linux, and OWASP VMs. Data was collected regarding the total number of true positives detected and the runtime of each tool. The results for ZAP showed the following: 1 vulnerability each was found for RXXS, SSXSS, SQLi, CSRF, LFI, and CMD Execution. No vulnerabilities were found for BSQLi. The total run time was 2 minutes and 50 seconds. For Skipfish, 1 vulnerability each was found for RXXS, SSXSS, SQLi, and CSRF. None were found for BSQLi, LFI, or CMD Execution. The total runtime was 1 minute and 48 seconds. W3af did not detect any of the vulnerability metrics. The total runtime was a whopping 5 hours and 20 minutes. In conclusion, the results show ZAP as being superior for vulnerability detection. Skipfish saw the fastest runtime, and w3af performed the worst for both vulnerability detection and runtime speed (Sagar, 2018).

2.2 Snort

Alnabulsi et al. proposed a customized rule set for SQL injection detection with Snort. The signature rules contained various iterations of certain keywords and phrases designed to detect SQLi activity in the captured traffic. This includes ((OR%20), (OR+), (%0A), (OR/*), (LIKE), (CONCAT), (VERSION), (HOSTNAME), (DATADIR), and (UUID)). In this paper, the experimental setup consisted of a Linux operating system where Snort and a DVWA web server were installed. The dataset consisted of 46 different SQLi attacks launched against a large sample of both normal and intentionally vulnerable websites. The detection results for each rule are as follows: Rule 1 – 15 false positives, 0 false negatives, 1 precision rate, and 0.7540 recall rate. Rule 2 – 15 false positives, 12 false negatives, 0.7391 precision rate, and 0.6938 recall rate. Rule 3 – 17 false positives, 0 false negatives, 1 precision rate, and 0.7301 recall rate. Rule 4 – 4 false positives, 0 false negatives, 1 precision rate, and 0.9387 recall rate. Rule 5 – 0 false positives, 0 false negatives, 1 precision rate, and 1 recall rate. Based on the results, rule 5 performed the best in all metrics, and all the rules performed well in precision rate except for rule 2 (Alnabulsi, 2014).

Mookhey and Burghate proposed various methods for detecting SQL injection and XSS attacks using Snort rules. Their proposed methods consisted of an in-depth list of regular expressions with explanations for the logic of the code structure. For SQLi, the rules include regex detection of meta-characters /(\%27)|(\')|(\-\-)|(\%23)|(#)/ix, modified regex detection for meta-characters /((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/i, regex for typical SQLi attacks /\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix, regex detection for union based SQLi attacks / ((\ % 2 7) | (\ ')) union/ ix (\ % 2 7) | (\ '), and regex detection MS SQL Server based SQLi attacks /exec(\s|\+)+(s|x)p\w+/ix. For XSS, the rules include regex detection for standard XSS attacks /((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix, regex detection for an html img src XSS attack /((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[^\n]+((\%3E)|>)/I, and ‘paranoid ’regex detection for XSS attacks /((\%3C)|<)[^\n]+((\%3E)|>)/I (Mookhey, 2004).

Caesarano and Riadi tested Snort for SQL injection detection in a network environment using the NIST method. NIST has nine phases for risk assessment, which are as follows: system characterization, threat identification, vulnerability identification, control analysis, likelihood determination, impact analysis, risk determination, control recommendations, and results documentation. A web server environment was set up and SQLMAP was used to conduct the SQLi detection simulation with Snort. The detection process for Snort was described in a series of steps as follows: A) The SQLi attacks are launched on the target system. B) Snort’s detection engine finds and processes the attacks. C) The captured attack packets are cross checked with the configuration rules in Snort to detect SQLi. D) If the attack matches the rule, it will be marked as an attack. E) The marked attacks will be raised as alerts in log files. Based on the result analysis using the User Acceptance Test, system users who chose Disagree before mitigation yield 35% and mitigation yield 0%. System users who chose Less Agree before mitigation yield 48% and after mitigation yield 10%. System users who chose Agree before mitigation yield 17% and after mitigation yield 44%. System users who chose Strongly Agree before mitigation yield 0% and after mitigation yield 46% (Caesarano, 2018).

Silva et al. tested Snort’s efficacy for detecting SQL injection attacks. In this paper, the testing environment was conducted using two virtual machines on a shared network – Kali Linux which functioned as an attack agent to launch SQLi attacks via Sqlmap, and Windows 7 which hosted the target database server and Snort. Based on the results, Snort did not raise any alerts for any of the SQLi attacks the target web application was vulnerable to, which were logically incorrect queries, timing attacks, and generic union queries. However, through Nmap Snort was able to generate three “Reset outside window” alerts due to a particular rule for detecting port scan traffic. Due to Snort not being configured to detect specific SQLi attacks, this paper determines it is a poor tool for SQLi detection in a network setting. However, it should be noted that the strength of Snort’s detection capability is largely dependent on the strength of its rule configuration. As such, the implementation of robust signature rules designed specifically for SQLi detection could make Snort an efficacious tool for this purpose (Silva, 2016).

CHAPTER 3
THEORY

3.1 Web Vulnerability Scanners

The structure of web vulnerability scanners usually consists of three main components – the crawler, fuzzer, and analyzer (Amankwah, 2020). The basic logic flow begins with the crawler searching through web links on a target application. The crawler starts with a seed web link, which is usually the home page of the web application, to traverse to other entry points (Kals, 2006). From there it will search for HTTP requests and URLs through the HTML DOM tree (Koswara, 2019). There are two lists that the found URLs will subsequently be distributed – the first list contains unsearched URLs, which will go through the HTML DOM tree until they have all been searched, and the list is empty. The second list contains parsed URLs. Next, the fuzzer uses the information gathered from the crawler to initiate attack patterns against the identified web links (Idrissi, 2017). Finally, the analyzer processes the outputs from the fuzzer to determine the success rate of the generated attacks by parsing the response feedback and looking for certain parameters that would indicate the existence of a vulnerability (Kals, 2006).

3.2 The Crawler

The general workflow of a web crawler starts with checking for web links to be downloaded. This includes checking if they are allowed to be downloaded by reading instructions found in a web application's robots.txt file. When a web link is downloaded, all associated information on the page is extracted and saved in a queue. The web links investigated by the crawler are called seeds, and each seed that is found is added to a list known as the crawl frontier. How the crawler visits links in the frontier is determined by certain rules that may differ depending on the crawler. The saved web links are downloaded from the parser and generator modules and stored in a database. This crawling process is how most search engines work to find the information requested by the user (Singh, 2014). The process is illustrated in Figure 1.

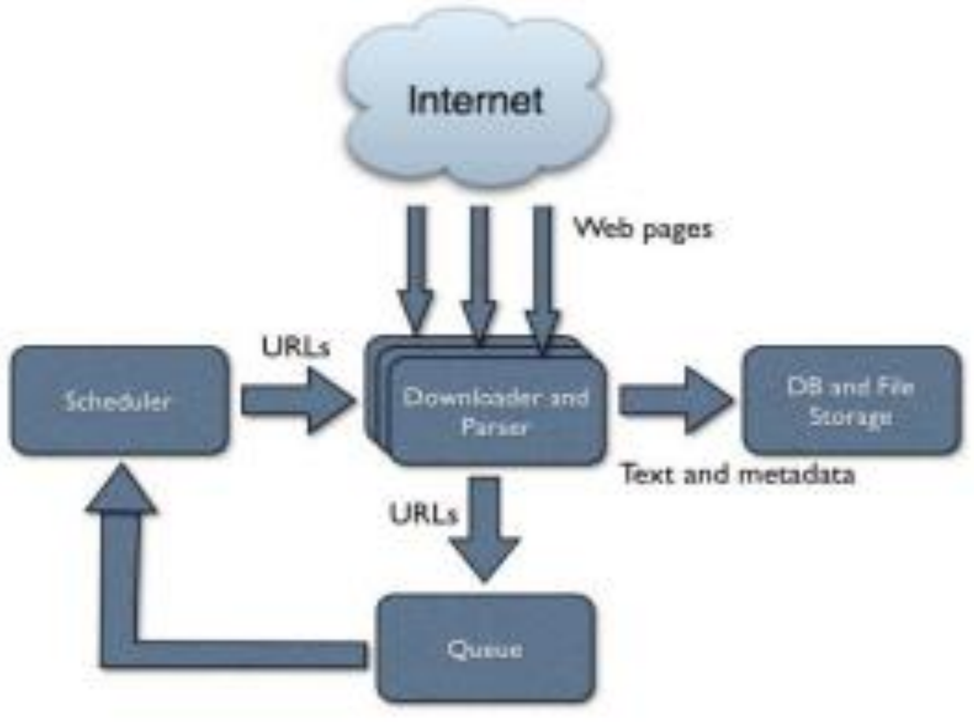


Figure 1 Crawler Workflow (Singh, 2014)

There are several different algorithms that can be used for web crawling functions. One of which is the Breadth First Search (BFS) algorithm, which is one of the most simple and traditional crawling methods. It works by using the frontier as a FIFO queue, which finds web links in the order they are first encountered. This algorithm is often used to solve problems related to graph traversal. The overall main advantage of BFS is its ability to provide domain specificity (Physu, n.d). Another crawling algorithm is Depth First Search (DFS). It starts its path by analyzing the contents of a web link first and moving to the next path based on the child links. The advantage of this method is its ability to traverse deep into a web application, but this is a double-edged sword that introduces the possibility of the crawler getting lost and entering an infinite loop if the targeted application's network of links is too complex or obscure (Yu, 2020).

3.3 The Fuzzer

Fuzzing is the automated process of finding vulnerabilities by inserting invalid data into an application – this includes files, network packets, and code. The workflow of fuzzing consists of four primary modules: test case generation, program execution, runtime state monitoring, and crash analysis (Figure 2). The test case generation module is where input is generated for fuzzing, which starts with the creation of seed files. The seed files undergo mutation which creates a variety of different test cases.

These test cases are then filtered depending on how well they can find new paths and vulnerabilities in the application (Wang, 2020).

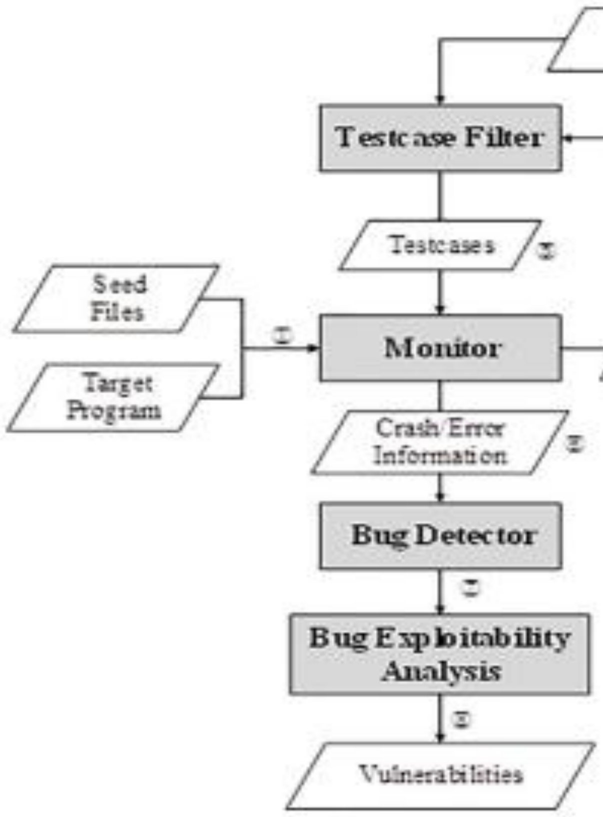


Figure 2 Fuzzer Workflow (Wang, 2020)

The two main types of fuzzing algorithms that are widely implemented are mutation and generation-based fuzzing. In mutation-based fuzzing, valid data is first collected and then modified in either random or heuristic ways. Heuristic modifications include replacing a small string with a long string or changing a length value to be either small or large. In generation-based fuzzing, test cases are generated from specified file formats and network protocols. The advantage that the former holds over the latter is that it does not require specific knowledge about the target system to conduct an attack, making the fuzzing process faster and more convenient. However, the advantage of generation-based fuzzing is the potential to produce more comprehensive and accurate fuzzing, because it is manually conducted based on the specifications of the target system. Miller and Peterson showed that generation-based fuzzing can perform up to 76% better than mutation in the context of PNG formats (Miller, 2007).

3.4 The Analyzer

The analyzer is typically the final component of the web vulnerability scanner workflow, and its main job is to determine the existence of a vulnerability in the application. As pointed out in the paper by Doupe, the Analyzer determines this based on the output from the fuzzing module. As an example, the return of a database error message related to SQL injection is likely to be interpreted as an SQL vulnerability existing by the analyzer (Doupe, 2010). The SQL example is illustrated in Figure 3.

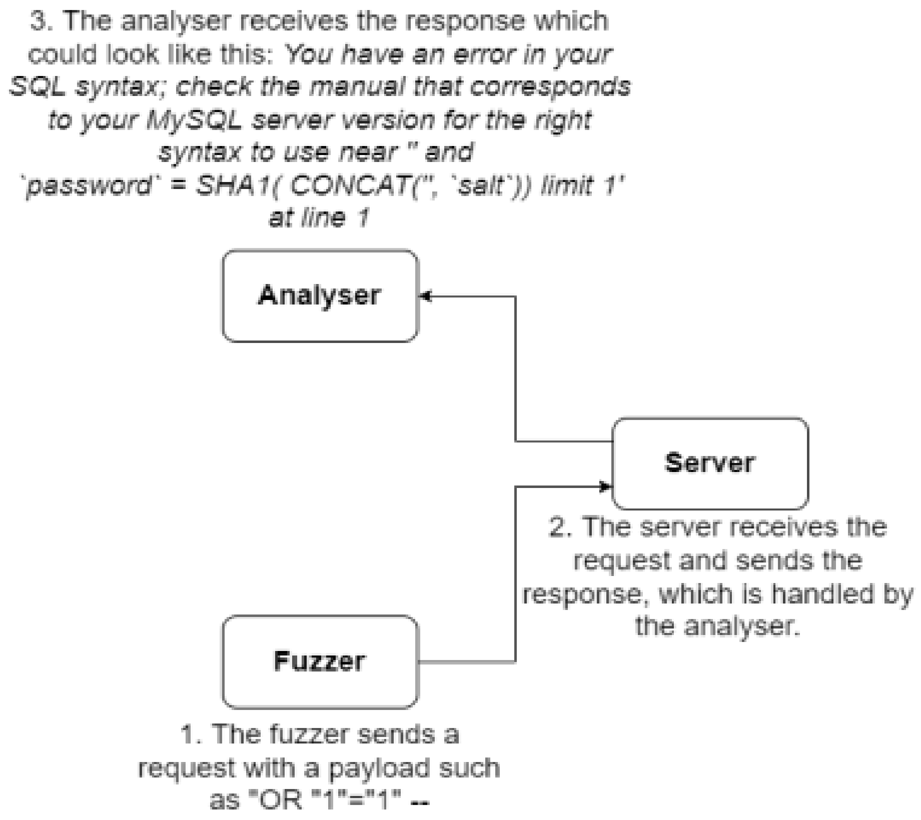

3. The analyser receives the response which could look like this: *You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' and `password` = SHA1(CONCAT('', `salt`)) limit 1' at line 1*
Analyser
Server
2. The server receives the request and sends the response, which is handled by the analyser.
Fuzzer
1. The fuzzer sends a request with a payload such as "OR "1"="1" --

Figure 3 Analyzer SQLi Example (Doupe, 2010)

The main limitation of the analyzer relates to how accurately it can find real vulnerabilities in an application. There are two specific problematic cases here, false positives and false negatives. The former case describes a scenario where the analyzer classifies a response as a vulnerability, when it is not. The latter case describes a scenario where the analyzer classifies a response as not being a vulnerability, when it is. It should be intuitive as to why false negatives are so potentially dangerous – they result in leaving vulnerabilities undetected that can be exploited in an application. False positives are not as threatening but are still problematic in their own respect as a large quantity of them can warrant a time-consuming effort to manually check which vulnerabilities are real and which ones aren't.

3.5 Limitations

The limitations of web vulnerability scanners are very much a product of the limitations that exist for the three core components discussed above. If the crawler fails to find all the relevant URLs in an application, any vulnerabilities that exist on those URLs will be entirely overlooked. Another problem pertains to user authentication, as web vulnerability scanners may not have the means to accommodate all types of authentications for different web applications. The scanner automatically logging out from accessing a web application’s logout page is an additional error that can occur, as well as failure to verify the validity of an authenticated session. Many web vulnerability scanners do provide means of handling these downsides, which include configuration options that allow the user to add URLs to be crawled for entry points, manually scan the application, and specify an authentication ID for accessing protected areas if it lacks the compatibility with specific authentication methods used by the web application (Esposito, 2018).

The fuzzing process can become cumbersome and inefficient when trying to reach deep parts of a program, and increasingly so the larger and more complex the program inputs are. The reliance on searching for program crashes is another limitation to keep in mind with fuzzing algorithms. The problem is that not all bugs become apparent in the event of a crash. This is the case with embedded devices, as crashes do not often occur even in the event of memory function errors. As such, more ingenuity is needed for optimal vulnerability detection (22). The main problem relevant to both fuzzing and the analyzer component in web vulnerability scanning is the occurrence of false positives and false negatives, which was discussed above. A common cause of this is the overconsumption of computing resources, which can result in the application not functioning properly. Some methods for mitigating this are creating an environment where fuzzers can operate in a distributed manner and improving fuzzing speed by minimizing performance impacts from events such as path explosion (Beaman, 2022).

3.6 SQL Injection

SQL injection (SQLi) is a type of code injection-based attack where SQL code is inserted into an input field on an application. The structure of the code is often designed to be interpreted as a database query by the application’s server-side functions, which can result in sensitive data stored in the database being exposed, manipulated, or deleted. Proper validation of user input is the main means of defending against and preventing SQLi attacks. Different types of SQLi attacks include tautologies, invalid queries, union queries, piggy-backed queries, stored procedures, and inference (Halfond, 2006). Figure 4 describes examples of each SQLi attack.

Tautologies are usually meant to bypass authentication and extract data from an application. The logic behind how they work is to exploit conditional statements in the internal application code to always evaluate as true. Successful attacks depend on the attacker identifying vulnerable parameters and figuring out how the internal code is structured to evaluate input for those parameters. The indication for an attack being successful is typically the return of at least one record from the database, but well-structured tautologies can often return all records (Halfond, 2006).

Invalid queries are invalid input that is meant to trigger an error response from the database. The intended purpose of this is to find information generated from the error response that may reveal details about the structure of an application's back-end database that can be used to find and exploit vulnerabilities. Examples include input that trigger syntax, type conversion, or logical error responses from the database. Syntax error responses can reveal injectable parameters, type conversion error responses can reveal information about column data types, and logical error responses can reveal associated table and column names (Halfond, 2006).

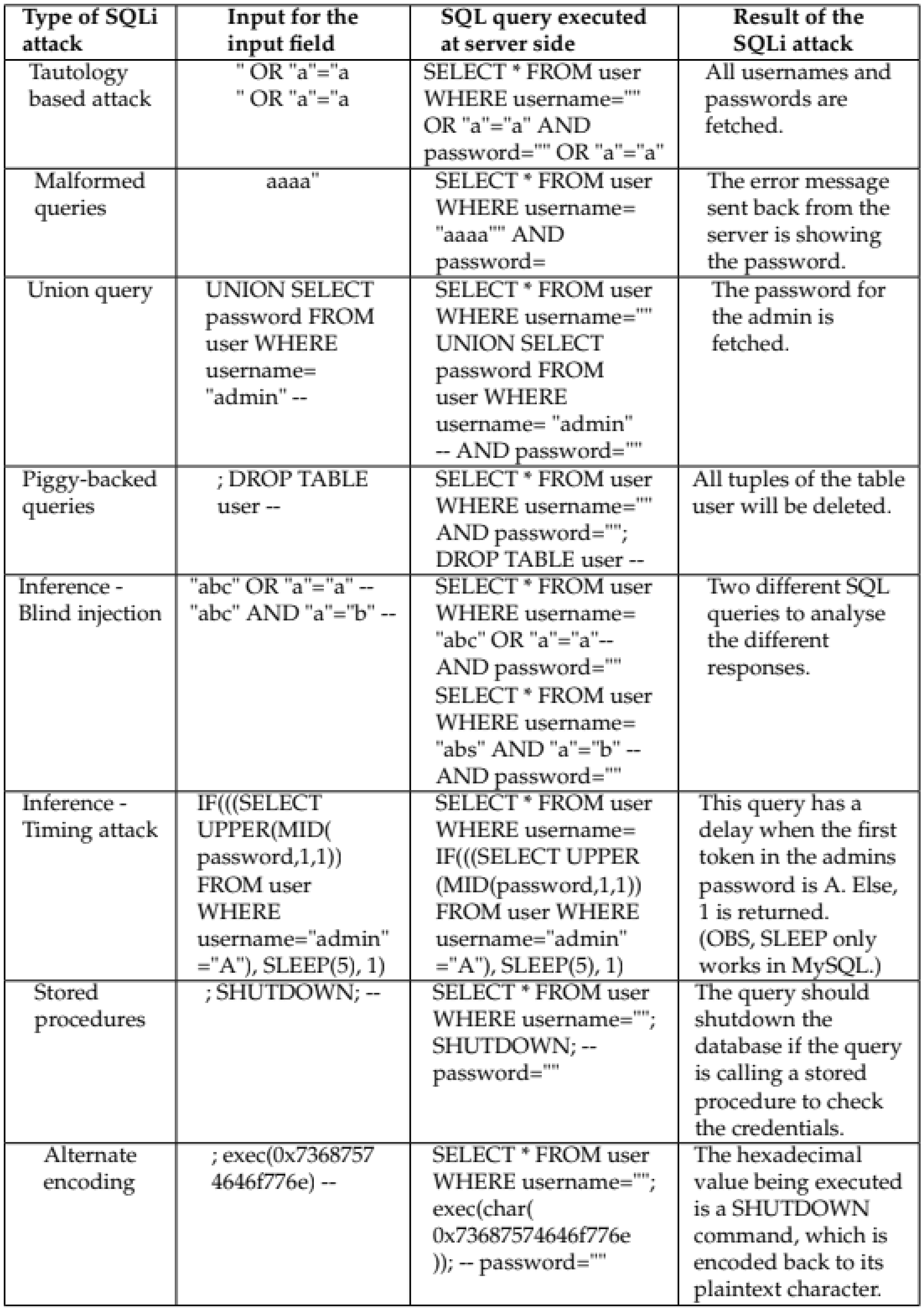
Union queries are aimed to extract data from different tables than what is intended by the application. This is accomplished by injecting code using the 'UNION SELECT' statement. A successful union query attack will return data that is the union of the original query intended to be executed by the application and the second query which is the malicious code intended to extract specific data by the attacker (Halfond, 2006).

Piggy-backed queries are malicious input designed to be executed in addition to the original query that the application is intended to execute. Piggy-backed queries are particularly dangerous because essentially any kind of attack can be carried out on the application from the additional queries if successful. An application is usually susceptible to this when multiple statements can be contained in one string within the internal code (Halfond, 2006).

Stored procedure SQLi attacks attempt to exploit existing stored procedure statements in the application's internal code. Because stored procedures are typically incorporated into databases for allowing additional functionality to the operating system, successful stored procedure attacks pose a threat to the operating system of the application as well. It is interesting and important to note that stored procedures are often designed to protect against SQLi attacks, yet it is clearly the case that they too are susceptible to being compromised. Therefore, developers should not rely on stored procedures exclusively and make sure to implement other defensive measures in place (Halfond, 2006).

Inference is a type of SQLi attack that attempts to change a query to function as an executable action based on true or false questions related to the data values in the database. This attack is often deployed when other SQLi attacks aimed to trigger error responses from the database fail to yield any useful information. Instead, inference seeks to find vulnerabilities in the application by injecting input and observing how the application reacts to it. Blind injection and timing attacks are two prominent methods of inference attacks. Blind injection works by inserting true/false statements and observing how the web pages on the application respond – if the statement is evaluated as true, they will function normally. If evaluated as false, the web pages will function differently than those that are true. Timing attacks work by inserting if/then statements and observing time delays in database responses. The if/then statements branch into areas of the database that at first are unknown. One of the branches will trigger an SQL statement that already takes a prespecified amount of time to execute – the WAITFOR keyword, for example. From this, the attacker can measure the delay in the response time and determine the specific if/then branch that the database interacted with (Halfond, 2006).

Alternate encodings are different from the other SQLi techniques discussed above – they do not provide any direct means of attacking an application. Rather, their purpose is to evade SQLi prevention and defensive measures that exist on an application. As such, they are used in conjunction with other SQLi attacks to facilitate exploiting vulnerabilities. For example, one means of defending against SQLi is validating input by searching for ‘bad characters’ that would commonly be used in an attack. This can be bypassed by alternating encodings of strings such as hexadecimal, ASCII, and Unicode, as basic bad character scanning practices may not cover all special combinations of these strings (Halfond, 2006).



Type of SQLi attack	Input for the input field	SQL query executed at server side	Result of the SQLi attack
Tautology based attack	" OR "a"="a " OR "a"="a	SELECT * FROM user WHERE username="" OR "a"="a" AND password="" OR "a"="a"	All usernames and passwords are fetched.
Malformed queries	aaaa"	SELECT * FROM user WHERE username= "aaaa"" AND password=	The error message sent back from the server is showing the password.
Union query	UNION SELECT password FROM user WHERE username= "admin" --	SELECT * FROM user WHERE username="" UNION SELECT password FROM user WHERE username= "admin" -- AND password=""	The password for the admin is fetched.
Piggy-backed queries	; DROP TABLE user --	SELECT * FROM user WHERE username="" AND password=""; DROP TABLE user --	All tuples of the table user will be deleted.

Inference - Blind injection	"abc" OR "a"="a" --
	"abc" AND "a"="b" --
		SELECT * FROM user WHERE username= "abc" OR "a"="a"-- AND password=""
		SELECT * FROM user WHERE username= "abs" AND "a"="b" -- AND password=""
			Two different SQL queries to analyse the different responses.

Inference - Timing attack	IF(((SELECT UPPER(MID(password,1,1)) FROM user WHERE username="admin" ="A"), SLEEP(5), 1)	SELECT * FROM user WHERE username= IF(((SELECT UPPER (MID(password,1,1)) FROM user WHERE username="admin" ="A"), SLEEP(5), 1)	This query has a delay when the first token in the admins password is A. Else, 1 is returned. (OBS, SLEEP only works in MySQL.)
Stored procedures	; SHUTDOWN; --	SELECT * FROM user WHERE username=""; SHUTDOWN; -- password=""	The query should shutdown the database if the query is calling a stored procedure to check the credentials.
Alternate encoding	; exec(0x7368757 4646f776e) --	SELECT * FROM user WHERE username=""; exec(char(0x73687574646f776e)); -- password=""	The hexadecimal value being executed is a SHUTDOWN command, which is encoded back to its plaintext character.

Figure 4 SQLi Examples (Jakobsson, 2022)

3.7 ZAP Internals

ZAP's crawler offers two crawling methods, Depth First Search and Breadth First Search, which were described above. It also has two main crawling capabilities – traditional crawling and AJAX crawling, where the latter is useful for applications with AJAX functionality. The fuzzer will save the requests from the crawler to be later fuzzed. The analyzer checks the DOM for stored vulnerabilities and error messages similar to other scanners but does not specify different database management types such as MySQL and SQLite (Jakobsson, 2022).

ZAP has two primary automated scanning modalities – passive and active scanning. Passive scanning does not change HTTP requests and responses gathered from the crawler, and thus is a safe means of exploring an application without compromising its integrity. Active scanning, on the other hand, conducts attacks on an application to find vulnerabilities. An interesting aspect about ZAP's active scanning method that makes it different from other conventional web vulnerability scanners is it does not utilize much of the fuzzer. Instead, ZAP finds vulnerabilities based on a comprehensive list of rules it applies to finding specific vulnerabilities. ZAP's fuzzing capabilities instead are more purposed for specific, manual tests. To that point, the automated nature of ZAP's passive and active scanning methods precludes their ability to find certain vulnerabilities, such as logic-based vulnerabilities (ex. broken access control), so manual testing is recommended in addition to automated testing to cover all bases in vulnerability scanning (Features, n.d).

3.8 Snort Internals

Snort functions use five main modules – packet capture, parsing, preprocessing, detection, and alarm generation (Figure 5). In the packet capture module, also referred to as Snort DAQ, packets are captured from the network card using Libpcap. In the parsing module, the captured packets are decoded based on the network protocol stack. In the link-layer protocol, packets are processed into data structures that prepare them for future detection functions. In the preprocessing module, the packet structures are processed further to prepare them for the detection module. In the detection module, the packets are analyzed through a set of rules. Every rule contains features that are designed to represent some form of network intrusion attack. The packets are compared to these features, and if matched raises an alert that gets sent to the alarm generation module. The algorithms behind this process are Boyer-Moore, Aho-Corasick, and PCRE regular expressions. In the alarm generation module, the matched rules are processed and documented in Snort's logs (Shuai, 2021).

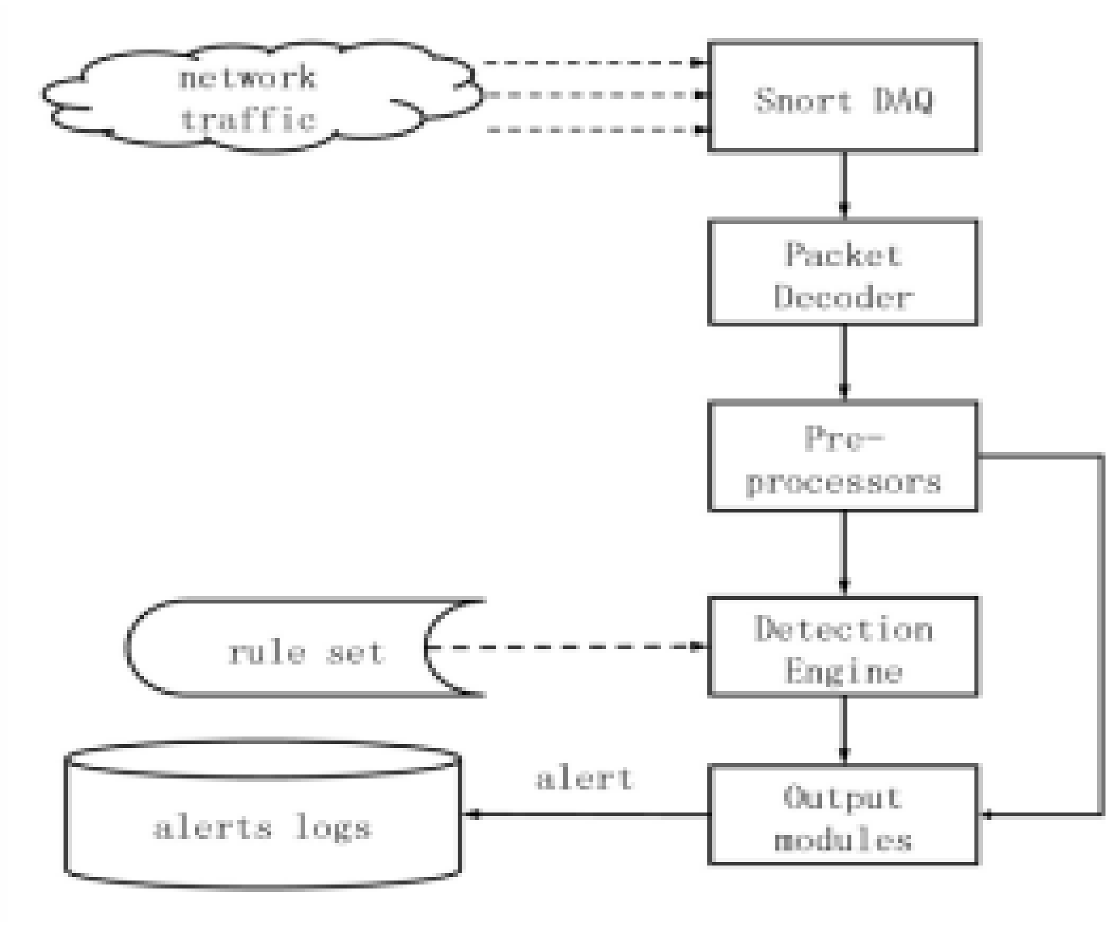


Figure 5 Snort Workflow (Shuai, 2021)

3.9 Boyer-Moore Algorithm

The Boyer-Moore algorithm is one of the many prominent string-matching algorithms used in pattern searching applications. The logic behind the algorithm is as follows – consider a body of text (T), and a pattern to be found in that text (P). P is first aligned under T from the left side. From there, the algorithm evaluates each character from the right side and skips alignments until P is found in T. The two rules that the full Boyer-Moore algorithm uses to do this are the bad-character rule and good-suffix rule, but a simplified version that is often used only implements the bad-character rule (Tsai, 2006). The bad-character rule works by skipping alignments upon a mismatched character between T and P until the characters match or P moves past the mismatched character entirely (Lin, n.d). An example is explained below:

1) T = ABCDEADEBC

P = ADE

Right away, there is a mismatch between C and E.

2) T = ABCDEADEBC

P =   ADE

Another occurrence of C is searched for in P. Because there is no matching C in the rest of P, P moves past C entirely (skipping two alignments). Now, there is a mismatch between A and E.

3) T = ABCDEADEBC

P =   ADE

Another occurrence of A is searched for in P. It exists, and so P shifts over two alignments (skipping one alignment) to match the A in P with the A in T. In this case after doing so, P is found in T.

The good-suffix rule functions similarly – the main difference is instead of only evaluating one character at a time between T and P, it evaluates a substring of T (t) to match P. The rule skips alignments until there is no mismatch between P and t or P moves past t entirely. Regarding the evaluation of P and t, there are two cases to consider – A) t matches P, or B) a prefix of P matches a suffix of t (28). An example of each case is explained below:

A)

1) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = DEFCADEF

DEF is matched between T and P, so t = DEF. A mismatch is then found between E and A.

2) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = DEFCADEF

Another iteration of DEF is searched for in P, which in this case exists.

3) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P =   DEFCADEF

P is then shifted over five alignments (skipping four alignments) to match DEF in T and P.

B)

1) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = EFACADEF

DEF is matched between T and P, so t = DEF. A mismatch is then found between E and A.

2) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = EFACADEF

Another iteration of DEF is searched for in P. In this case, it does not exist. However, a prefix of P that can match a suffix of t does exist (EF).

3) T = ABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = EFACADEF

P is then shifted over six alignments (skipping five alignments) to match the prefix EF in P with the suffix EF in t.

When using the good-suffix rule, it is also important to note the character directly preceding an occurrence of t found in P upon a mismatch between T and P. If the character preceding the occurrence of t is the same as the current mismatched character in P, then it is best to skip this and instead look for another occurrence of t before the character preceding the first found occurrence, as doing so will avoid a given subsequent mismatch (29). An example is explained below:

1) T = GAAAAAACABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = ACDEFCGBDEFACBDEF

DEF is matched between T and P, so t = DEF. A mismatch is then found between E and B.

2) T = GAAAAAACABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = ACDEFCGBDEFACBDEF

Another occurrence of DEF is searched for in P. It is found, but the character directly preceding it is the same as the current mismatched character in P (B). If this occurrence of DEF is shifted to match DEF in T, it will automatically result in another mismatch. As such, this occurrence should be skipped, and another occurrence of DEF should be searched for.

3) T = GAAAAAACABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = ACDEFCGBDEFACBDEF

Another occurrence of DEF is found, and the preceding character C is not the same as the current mismatched character B.

4) T = GAAAAAACABCDEDEFDCADGFGDECADEFCGABCDECADEF

P = ACDEFCGBDEFACBDEF

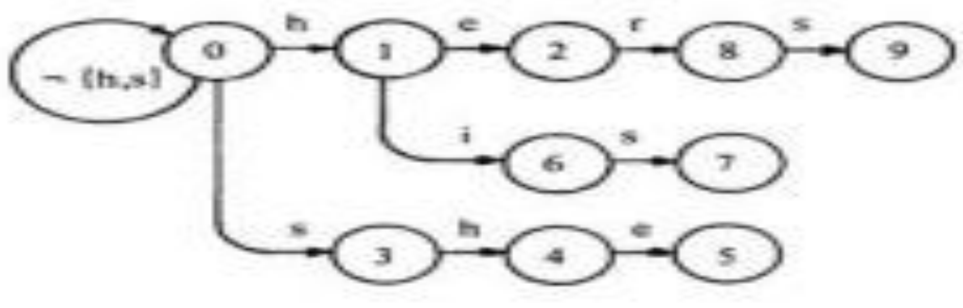
P is then shifted over twelve alignments (skipping eleven) to match the new occurrence of DEF in P with DEF in T, and the process continues.

In real-world coding applications, both rules of Boyer-Moore are implemented as pre-calculated functions that are called upon to determine the most efficient route for skipping alignments to find a given pattern. Generally, pre-processing gives Boyer-Moore notable advantages over more straightforward approaches such as the Brute-Force or Naive algorithm. Lin and Soe showed that Boyer-Moore required significantly less alignment shifts, comparisons, and run-time compared to Brute-Force to find patterns of various sizes (Lin, n.d). Boyer-Moore performed better the longer the pattern was. The increased efficiency is explained by pre-processing allowing Boyer-Moore to skip unnecessary or repetitive character comparisons, whereas the nature of Brute-Force compels it to compare every existing character in the pattern and text, making the former ideal for finding larger, more complex patterns. The main disadvantage of this, however, is that it requires more computational space and resources.

Boyer-Moore has also displayed notably faster runtimes compared to similar algorithms that utilize pre-processing, such as Knuth-Morris-Pratt (KMP). According to Dawood and Barakat, Boyer-Moore yielded faster execution times compared to KMP in every text and pattern size evaluated (text sizes ranging from 1 KB to 200 MB, and pattern sizes ranging from 10 characters to 75 characters) (Dawood, 2020). This paper also showed relatively faster run times at greater text and pattern sizes for both algorithms.

3.10 Aho-Corasick Algorithm

Aho-Corasick is a pattern matching algorithm that can evaluate multiple patterns. There are two ways of implementing Aho-Corasick, which are Non-deterministic Finite Automata and Deterministic Finite Automata. The automation process of the algorithm works in two steps: in the first step, a limited number of patterns are preprocessed. In the second step, the text desired to be matched is inserted, and the matching process uses three functions: goto, failure, and output. An example is illustrated below using a given pattern set (he, she, his, hers) in Figure 6 (Tran, 2012).



(a)The *goto* function

i	1	2	3	4	5	6	7	8	9
f(*i*)	0	0	0	1	2	0	3	0	3

(b)The *failure* function

i	*output* (*i*)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c)The *output* function

Figure 6 Aho-Corasick Functions Example (Tran, 2012)

In this example, the goto function maps a pair of a state and input symbol into a state or failure message. The failure function puts a state into another state and is called when the previous function reports a failure message. The output function displays a set of keywords at matching states (Tran, 2012).

CHAPTER 4
METHODS

The experimental setup consisted of a Windows 10 computer and an Ubuntu Desktop virtual machine installed using VMware. On the Windows 10 desktop, OWASP ZAP was installed, and this module functioned as an attacking agent. On the Ubuntu Desktop virtual machine, an Apache localhost web server and database were configured using the XAMPP package. On the web server, a customized vulnerable web application was created and served as the scanning target. Snort was also installed on Ubuntu, and this module served as the target network system to be attacked. First, the target web application was infiltrated using ZAP's Manual Explore feature to identify all web pages associated with the seed URL. Next, the target was automatically scanned using ZAP's Active Scan feature, which launches a series of attacks against the web application. As that process is running, Snort is simultaneously running on Ubuntu to capture the network traffic on the target localhost server. In essence, ZAP attacks the target system and will report whatever vulnerabilities it detects through its analyzer module. Snort monitors the target system and will report whatever vulnerabilities it detects from ZAP's attack process through its detection and alarm generation modules. This setup is illustrated in Figure 6.

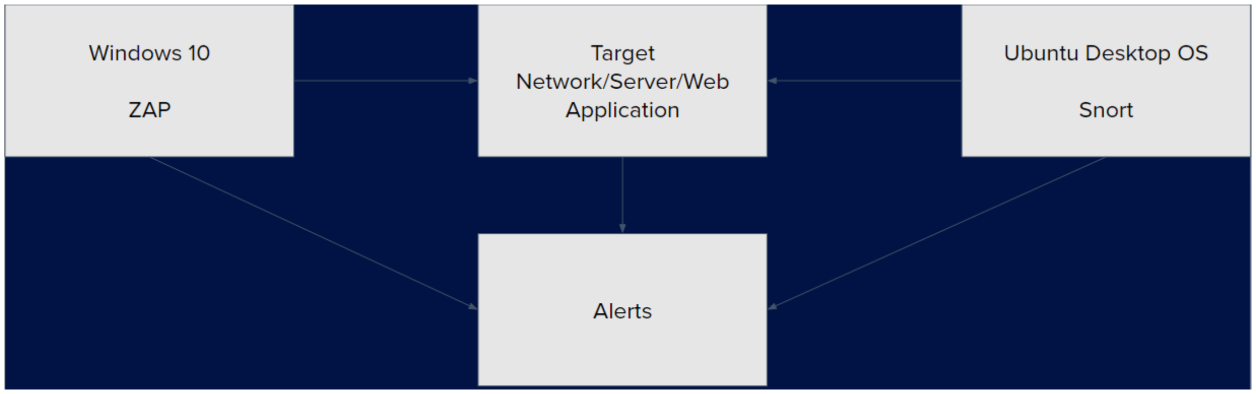


Figure 7 Experimental Setup

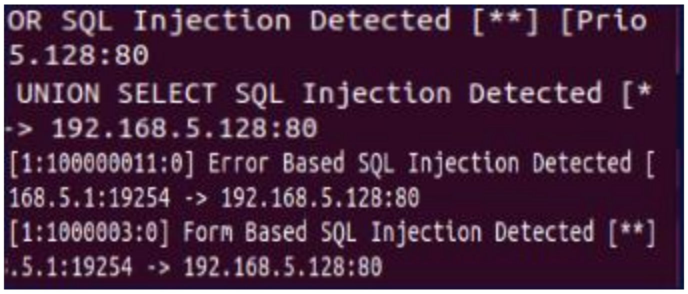
While ZAP's Active Scan feature attacks the target web application with several different types of attacks, SQLi was the only attack type evaluated in this thesis. As such, Snort was configured using a customized signature rule set designed to detect SQLi attacks only (Chandel, 2018). The SQLi attack types designed to be captured by the ruleset were single and double quote error-based attacks, Boolean attacks, encoded attacks, order by attacks, and form-based attacks. Data was collected looking at the alert log documentation generated by both tools, and the performance metrics evaluated were total unique SQLi attack types detected, false positives, false negatives, processing time, and total memory usage.

CHAPTER 5
RESULTS

The results are summarized in Table 1. ZAP detected 4 unique SQLi attack types, did not yield any false positives or negatives, took a total 19.399 seconds to finish running, and yielded a total memory usage of 710,740,000 bytes. Snort detected 6 unique SQLi attack types, two of which were false positives, did not yield any false negatives, took a total of 27 seconds to finish running, and yielded a total memory usage of 119,321,712 bytes. The unique attack types detected by ZAP were single quote error-based attacks, Boolean based attacks, union-based attacks, and form-based attacks. Snort detected these attack types as well (Figure 8), but also falsely detected two kinds of double quote error-based attacks.

	OWASP ZAP	Snort IDS
SQLi Attacks Detected	4	6
False Positives	0	2
False Negatives	0	0
Processing Time (seconds)	19.399	27
Total Memory Usage (bytes)	710,740,000	119,321,712

Table 1 Results



```
OR SQL Injection Detected [**] [Prio
5.128:80
 UNION SELECT SQL Injection Detected [*
-> 192.168.5.128:80
[1:100000011:0] Error Based SQL Injection Detected [
168.5.1:19254 -> 192.168.5.128:80
[1:1000003:0] Form Based SQL Injection Detected [**]
.5.1:19254 -> 192.168.5.128:80
```

Figure 8 Snort SQLi Detection

CHAPTER 6
ANALYSIS

The analysis section is the main contribution this thesis aims to provide – after obtaining the results of the head-to-head performance comparison between ZAP and Snort, the research questions posed in the introduction of this paper can now be revisited.

6.1 RQ1

Based on the results of the performance metrics, ZAP and Snort both performed equally well in detecting unique SQLi attack types after correcting for false positives. ZAP performed better in the false positives metric, as it did not yield any false positives while Snort yielded two. In practice, there is not too much significance to this finding, as two false positives shouldn't be difficult to correct. There was no difference between the tools in the false negatives metric, as both yielded zero false negatives that could be detected. ZAP performed better for processing time, as it was about 8 seconds faster than Snort. There is likely little practical significance to this however, as an additional 8 seconds is hardly a long wait time for the tool to finish running. Snort performed better in the memory usage metric, as it used almost 7x less memory than ZAP. This is a meaningful finding, as over 700 million bytes of memory usage is quite significant in taxing system resources.

6.2 RQ2

To answer this question, it is important to return to the theoretical section of this paper and understand how each tool works. ZAP's Active Scan uses a rule-based method that functions by sending attacks to the target web application, and each rule corresponds to a specific attack type. The analyzer module then raises potential alerts about potential vulnerabilities that it detects based on the success rate of the Active Scan attack. Snort functions similarly in that it also uses a signature rule-based method to detect threats. The three main algorithms implemented in the detection module to find threats through the rules are Boyer-Moore, Aho-Corasick, and PCRE expressions. Because no expression based SQLi attacks were present, it is likely the PCRE function played little role and thus the contribution of the other two algorithms can be highlighted. The main strength of Boyer-Moore lies in its ability to utilize the bad-character and good-suffix rules to skip alignments, making it faster and more efficient in finding patterns compared to more linear approaches like Brute-Force (Lin, n.d). The main strength of Aho-Corasick is its memory efficiency utility from its implementation of trie structures for pattern matching (Tran, 2012).

Given this, why did Snort yield two false positives whereas ZAP yielded none? This is likely due to the high degree of network traffic generated by ZAP's Active Scan, and ZAP's ability to cross check attacks with its analyzer module. Snort likely detected the injection of some double quote string from ZAP's attack process somewhere in the network traffic even though none were aimed at the target web application, resulting in false positives. Because ZAP can verify the success rate of its attacks with the analyzer module, it has a lower likelihood of yielding false positives. With regards to processing time, it is surprising that Snort performed worse, because the Boyer-Moore algorithm is usually one of the fastest. Its failure to provide a shorter run time may be explained by the fact that the generated SQLi attack inputs were not particularly long. As previously discussed, Boyer-Moore becomes the most efficient the longer the given texts and patterns. This advantage is largely missed if the string parameters are not long enough. With regards to memory usage, the superiority of Snort for using less memory is likely explained by its implementation of the Aho-Corasick algorithm, which has previously been shown to be beneficial for memory usage efficiency (Karimov, 2020). Conversely, because ZAP's Active Scan process generates such a large degree of network traffic from all the different kinds of attack types it launches, its substantial memory and resource usage is not surprising.

6.3 RQ3

The main identified factors after discussing the second question are ZAP's Active Scan feature, and Snort's pre-processing detection algorithms Boyer-Moore and Aho-Corasick. In continuing future research with ZAP, this thesis proposes that both the Boyer-Moore and Aho-Corasick algorithms should be implemented in ZAP. ZAP offers the feature of adding custom scripts that can function as rules in the vulnerability detection process, like Snort. These scripts would be modified versions of its SQLi ruleset using both algorithms. As a future work proposal, it would be interesting if ZAP's memory usage efficiency can be improved with the Aho-Corasick algorithm, and if its run time can be even faster using the Boyer-Moore algorithm if SQLi attack inputs are sufficiently long.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

In conclusion, ZAP and Snort were compared head-to-head in various performance metrics related to SQLi vulnerability detection. Based on the results, there were no differences in the total amount of SQLi attacks detected or false negatives detected. There were differences with regards to false positives detected and total processing time which favored ZAP, and memory usage which favored Snort. Based on the analysis, this thesis proposes an experimental setup where the Boyer-Moore and Aho-Corasick algorithms are implemented into ZAP to improve its processing time and memory usage efficiency for future work research. The experimental setup involves using the script feature in ZAP to create versions of its existing SQLi ruleset that are modified using both algorithms. The modified rules would be compared with the pre-existing rules to see if the proposed method results in a faster runtime and less memory resource usage when scanning for SQLi vulnerabilities.

REFERENCES

Alnabulsi, H., Islam, M. R., & Mamun, Q. (2014). Detecting SQL injection attacks using SNORT IDS. Asia-Pacific World Congress on Computer Science and Engineering,

Amankwah, R., Chen, J., Kudjo, P. K., & Towey, D. (2020). An empirical comparison of commercial and open-source web vulnerability scanners. *Software: Practice and Experience*, *50*(9), 1842-1857.

Beaman, C., Redbourne, M., Mummery, J. D., & Hakak, S. (2022). Fuzzing vulnerability discovery techniques: survey, challenges and future directions. *Computers & Security*, 102813.

Caesarano, A. R., & Riadi, I. (2018). Network forensics for detecting SQL injection attacks using NIST method. *Int. J. Cyber-Security Digit. Forensics*, *7*(4), 436-443.

Chandel, R. (2018, January 11). *Detect SQL Injection Attack using Snort IDS*. Hacking Articles. https://www.hackingarticles.in/detect-sql-injection-attack-using-snort-ids/

DAWOOD, S. S., & BARAKAT, S. A. (2020). Empirical performance evaluation of knuth morris pratt and boyer moore string matching algorithms. *Journal of Duhok University*, *23*(1), 134-143.

Doupé, A., Cova, M., & Vigna, G. (2010). Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. Detection of Intrusions and Malware, and Vulnerability Assessment: 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings 7,

Esposito, D., Rennhard, M., Ruf, L., & Wagner, A. (2018). Exploiting the potential of web application vulnerability scanning. ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018,

Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. Proceedings of the IEEE international symposium on secure software engineering,

Hawamleh, A., Alorfi, A. S. M., Al-Gasawneh, J. A., & Al-Rawashdeh, G. (2020). Cyber security and ethical hacking: The importance of protecting user data. *Solid State Technology*, *63*(5), 7894-7899.

Idrissi, S., Berbiche, N., Guerouate, F., & Shibi, M. (2017). Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. *International Journal of Applied Engineering Research*, *12*(21), 11068-11076.

Jakobsson, A., & Häggström, I. (2022). Study of the techniques used by OWASP ZAP for analysis of vulnerabilities in web applications. In.

Kals, S., Kirda, E., Kruegel, C., & Jovanovic, N. (2006). Secubat: a web vulnerability scanner. Proceedings of the 15th international conference on World Wide Web, Karimov, M., Tashev, K., & Rustamova, S. (2020). Application of the Aho-Corasick algorithm to create a network intrusion detection system. 2020 International Conference on Information Science and Communications Technologies (ICISCT),

Koswara, K. J., & Asnar, Y. D. W. (2019). Improving vulnerability scanner performance in detecting ajax application vulnerabilities. 2019 International Conference on Data and Software Engineering (ICoDSE),

Lin, L. L., & Soe, M. T. A Comparative Study on Brute-Force String Matching and Boyer-Moore String Matching.

Makino, Y., & Klyuev, V. (2015). Evaluation of web vulnerability scanners. 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS),

Mburano, B., & Si, W. (2018). Evaluation of web vulnerability scanners based on owasp benchmark. 2018 26th International Conference on Systems Engineering (ICSEng),

Miller, C., & Peterson, Z. N. (2007). Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep, 4*.

Mookhey, K., & Burghate, N. (2004). Detection of SQL injection and cross-site scripting attacks. *Symantec SecurityFocus*.

Phyu, P. E., & Maw, S. Y. *Implementation of Breadth First Search Algorithm for Crawling the Websites* MERAL Portal].

Sagar, D., Kukreja, S., Brahma, J., Tyagi, S., & Jain, P. (2018). Studying open source vulnerability scanners for vulnerabilities in web applications. *IIOAB JOURNAL*, *9*(2), 43-49.

Shuai, L., & Li, S. (2021). Performance optimization of Snort based on DPDK and Hyperscan. *Procedia Computer Science*, *183*, 837-843.

Silva, R., Barbosa, R., & Bernardino, J. (2016). Testing snort with SQL injection attacks. Proceedings of the Ninth International C* Conference on Computer Science & Software Engineering,

Singh, M., & Varnica, B. (2014). Web crawler: Extracting the web data. *International Journal of Computer Trends and Technology*, *13*(3), 132-137.

Tran, N.-P., Lee, M., Hong, S., & Shin, M. (2012). Memory efficient parallelization for Aho-Corasick algorithm on a GPU. 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems,

Tsai, T. H. (2006). Average case analysis of the Boyer-Moore algorithm. *Random Structures & Algorithms*, *28*(4), 481-498.

Wang, Y., Jia, P., Liu, L., Huang, C., & Liu, Z. (2020). A systematic review of fuzzing based on machine learning techniques. *PloS one*, *15*(8), e0237749.

Yu, L., Li, Y., Zeng, Q., Sun, Y., Bian, Y., & He, W. (2020). Summary of web crawler technology research. Journal of Physics: Conference Series,

Zukran, B., & Siraj, M. M. (2021). Performance Comparison on SQL Injection and XSS Detection using Open Source Vulnerability Scanners. 2021 International Conference on Data Science and Its Applications (ICoDSA),

Top 10 Web Application Security Risks. (2021). OWASP. Retrieved April 2023, from https://owasp.org/www-project-top-ten/#

Features. (n.d). ZAP. Retrieved April 2023, from https://www.zaproxy.org/docs/desktop/start/features/