Source Code Vulnerability Detection Using Static Analysis and Pattern Matching

Jithmal Danusha Pitiyegedara

Department of Network and Security, NSBM, Homagama, Sri Lanka pgjdanusha@students.nsbm.ac.lk

Isuru Sri Bandara Nishshanka
Lecturer, NSBM Green University,
Homagama, Sri Lanka.
isuru.s@nsbm.ac.lk

Abstract- **In the rapidly evolving field of web development, ensuring the security of web applications is paramount. Most of the vulnerabilities occur due to lack of best practices during coding. Traditional security testing methods, which are predominantly reactive, tend to identify vulnerabilities in source code only after substantial portions of the application have been developed. This reactive approach significantly increases the complexity and cost of mitigating security risks. This research addresses this challenge by integrating security directly into the development process through a VS(Visual Studio) Code Extension by performing a real-time Static Analysis of source code to detect vulnerabilities which are written using JavaScript language. The tool can identify common security vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) using the pattern matching techniques of static analysis. By integrating the tool directly into VS Code, it can provide developers with immediate feedback on security risks. The tool is not only useful for identifying vulnerabilities but also enhances the developer's overall coding practices. The integration of the tool into the development environment will reduce time and resources spent on post- development security testing. In conclusion, this tool represents a significant step forward in the effort to embed security within the software development lifecycle. Its development and deployment will underscore the effectiveness of proactive security practices, offering a model that could be replicated and extended to other development environments and programming languages.**

Keywords—static analysis, pattern matching, JavaScript, source code, vulnerabilities, proactive testing, vs code

I. INTRODUCTION

In the landscape of cyber technology, security has become a paramount concern, especially with the increasing complexity and functionality of applications. As computer applications evolve, they often incorporate various services that handle sensitive user data or perform critical operations, such as payment processing, personal data management, and other interactive functionalities. These enhancements, while improving user experience and functionality, also broaden the attack surface, making applications susceptible to a variety of threats.

Early threats like viruses and worms have given way to more complex attacks such as phishing, ransomware, and advanced persistent threats (APTs). The proliferation of Internet of Things (IoT) devices and the widespread adoption of cloud computing have introduced new vulnerabilities, making these systems prime targets for cybercriminals. Modern threats now leverage advanced techniques like AI and machine learning to evade detection and automate attacks, posing significant risks to both individuals and organizations. As the digital landscape continues to expand, the importance of robust cybersecurity measures is more critical than ever.

Cyber-Related Attack	Year	Attack Spread Method	Consequences
Vladimir Levin's Attack the Citibank1	1994–1995	unknown	around 10 million dollars were stolen
Melissa Virus	1999	used users' trust to click an email attachment	billions of dollars were lost in many countries
ILOVEYOU Worm3	2000	used users' trust to click an email attachment	more than 45 million computers were infected
MyDoom worm4	2004	used attention-grabbing subjects by email, such as errors, tests, etc.	DDoS attacks by allowing remote access was launched
Zeus Trojan	2007	spam email with drive-by downloads	login details such as email and bank accounts were stolen
Stuxnet Worm	2010	attack the programmable logic unit by stealing source codes	control of industrial processes was taken
Attack on USA Natural Gas Pipeline	2012	accessing confidential information through phishing	security credentials were stolen
Mirai Malware	2016	vulnerability of IoT devices was exploited	DDoS attacks were launched
WannaCry Ransomware	2017	windows vulnerability was exploited	computer hard drives were encrypted, and 150 countries were affected
Emotet Trojan	2018	emails in the form of spam and phishing campaigns	sensitive information such as credit card details was stolen
MyFitnessPal	2018	software vulnerability was exploited	150 million users were affected
Ransomware Attack on Magellan	2020	emails in the form of spam and phishing campaigns	health data of 365,000 patients were stolen
CovidLock Ransomware	2020	exploited users' trust by using COVID-19 statistic	android devices' data were encrypted, and data access was denied
Accellion Supply Chain Attack	2021	third-party vulnerabilities were exploited	confidential data from large organizations were stolen
Kaseya Ransomware Attack	2021	zero-day exploits were used	around 1500 companies' data were compromised with the request of 50,000 to 5 million dollar ransoms per victim

Fig. 1. Significant risks on AI

When looking at the Fig.1. [1] and other related internet sources, it can conclude that most of the attacks are related to email and web security. The history of web security threats dates to the early days of the internet when simple HTML websites transitioned into complex web applications. Initially, security was not a primary concern, but as web applications began to handle more sensitive information and perform critical functions, the importance of securing these applications became evident.

Numerous security tools have been developed to address the growing web security threats. These tools are generally categorized into Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) methodologies[4]. While these tools are effective, they predominantly focus on the post-development stages of applications. Their reactive nature means vulnerabilities are often identified only after they have been introduced into the codebase, leading to higher remediation costs and potential delays in deployment.

Given the limitations in existing security tools, there is a clear and present need for a proactive approach that integrates security directly into the development phase of web applications. This research emerges as a solution to this need, providing a mechanism using Static Analysis to detect and mitigate vulnerabilities during the coding process itself. This proactive approach ensures that vulnerabilities can be identified and rectified in real-time, significantly reducing the chances of security risks in the later stages of development.

II. Literature Review

The proliferation of web-based applications has necessitated the development of advanced security tools to protect against a wide range of vulnerabilities. As the complexity of web applications increases, so does the variety of attack vectors. In response, security tools and methodologies have evolved to detect and mitigate potential threats. Two main types of tools Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) have become foundational in the field of application security[4].

A. Static Application Security Testing (SAST)

SAST tools analyze source code to detect vulnerabilities without executing the program. These tools scan the entire codebase, including third-party libraries and dependencies, to identify patterns indicative of security flaws[5].

1) Advantages

SAST tools can identify vulnerabilities early in the development phase, allowing developers to address issues before deployment.
These tools provide a thorough analysis of the codebase, enhancing the security posture of the application.
SAST tools can be integrated into CI/CD pipelines, facilitating automated security checks during the development process.

2) Limitations:

These tools may generate false positives, leading to unnecessary remediation efforts.
While effective at code analysis, SAST tools might not detect issues related to runtime behavior or server-side configurations.

B. Dynamic Application Security Testing (DAST)

DAST tools interact with a running application, simulating real-world attacks to identify vulnerabilities. This approach involves testing the application's responses to various attack vectors, providing an assessment of its security in an operational state[5].

1) Advantages:

DAST tools simulate actual attack scenarios, providing a realistic evaluation of the application's security posture.
These tools assess the application's runtime behavior, configurations, and interactions with external systems.

2) Limitations:

DAST tools do not have access to the source code, which can restrict their ability to pinpoint specific vulnerabilities.
These tools are typically used later in the development lifecycle, which can make the remediation process more difficult and costly.

According to previous research done by researchers by testing both SAST and DAST tools, SAST tools found more Vulnerabilities when compared to DAST tools. The below Figure 2 represents the result outputs[5].

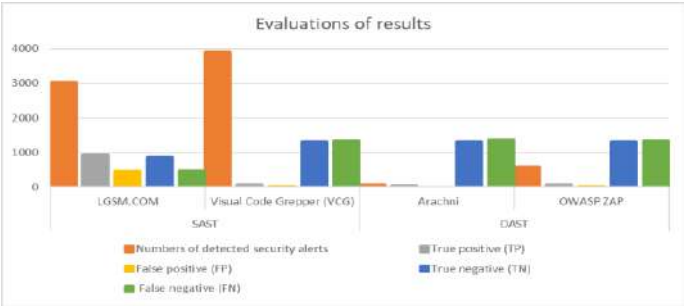


Fig. 2. Results Outputs

The figures show that existing DAST tools are highly effective in scanning the source code for vulnerabilities, but they typically operate post-development or during the later stages of the development lifecycle. This often results in vulnerabilities being detected only after significant portions of the application have been developed, which can lead to costly and time-consuming remediation processes. Moreover, these tools generally serve quality assurance teams rather than developers directly involved in the coding process. So, for the requirement addressed earlier to detect and mitigate code vulnerabilities during development phase, the SAST approach is most suitable.

II. Static Code Analysis

Static analysis refers to the examination of code without executing it, aiming to find potential vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and unsafe variable declarations[6]. Let's see how it leverages

static analysis to scrutinize JavaScript code for known vulnerability patterns.

A. Regex for Pattern Matching

Regex (Regular Expressions) is a powerful way for identifying specific patterns within text. In the context of the research, regex patterns are employed to match code snippets that correspond to known vulnerability patterns written in JavaScript[7]. For the explanation, let's take user input sanitization during development. Consider a JavaScript variable declaration like below,

var userInput = getUserInput();

Static analysis inspects the code to detect if 'userInput' is utilized in a manner that could lead to security issues. For instance, if 'userInput' is directly incorporated into an SQL query without sanitization, it could be susceptible to SQL injection attacks[8]. The vulnerability triggers when the un sanitized 'userInput' variable is concatenated in to a SQL query like below,

*var query = "SELECT * FROM users WHERE username = '"*

+ userInput + "';

Now because of the concatenation of 'userInput' directly without sanitizing is vulnerable, using a regex pattern, it's capable of performing a pattern matching to detect the patterns with concatenated user input in a SQL query. Next, to identify a SQL pattern like above we can implement a regex pattern. For example, a regex to detect unsanitized variable usage in an SQL query might look like this:

/SELECT\s+\s+FROM\s+\w+\s+WHERE\s+\w+\s*=\s*['"]?\s*\w+\s*['"]?/i*

This pattern leverages the power of regular expressions to match common SQL query structures that involve potentially unsafe variable concatenation. The characters which is included in the regex patterns are known as Metacharacters which are differ from language to language. Other than the metacharacters the regex can also include the special string literals like 'SELECT', 'FROM', 'WHERE' to identify the code snippet. Table 1 [9] represents the further breakdown of the above regex patten,

TABLE 1. REGEX BREAKDOWN

'["']?'	Matches zero or one single or double quote character.
'/i'	The i at the end indicates case-insensitive matching.

For brief analysis refer the book *'Regular Expression Pocket Reference by Tony Stubblebine'* which includes basic Metacharacters which used to create Regex Content which is supported for JavaScript[9]. To learn more further information about how regex work and other Metacharacters it can be found from online sources like regex101, w3schools etc.

Class	Meaning
[...]	A single character listed, or contained within a listed range.
[^...]	A single character not listed, and not contained within a listed range.
.	Any character except a line terminator, [^\x0A\x0D\u2028\u2029].
\w	Word character, [a-zA-Z0-9_].
\W	Nonword character, [^a-zA-Z0-9_].
\d	Digit character, [0-9].
\D	Nondigit character, [^0-9].
\s	Whitespace character.
\S	Nonwhitespace character.

Table 40. JavaScript anchors and other zero-width tests

Sequence	Meaning
^	Start of string, or the point after any newline if in multiline match mode, /m.
$	End of search string, or the point before a string-ending newline, or before any newline if in multiline match mode, /m.
\b	Word boundary.
\B	Not-word-boundary.
(?=...)	Positive lookahead.
(?!...)	Negative lookahead.

Fig. 3. JavaScript Anchors

By referring to these sources there is a good possibility to make a complete algorithm to detect different kinds of vulnerable code patterns including SQLI, XSS, CSRF and other related vulnerabilities. The algorithm will include all possible vulnerable code patterns which are likely to be bad coding practices done by developers. After programming the algorithm, it can be implemented into a tool, software, third party application or to a programming IDE(vs code,

sublime text etc.) which could be most appropriate because the development of web systems is done by using IDE s.

III. LOGIC/ALGORITHM

Creating a complete and precise algorithm and integrating it to a working tool includes several steps. As discussed above, including regex for pattern matching it has a great potential for making a complex algorithm which covers the mentioned vulnerabilities earlier. The complex algorithm is consist of two main sections. As the vulnerability in a written code always comes with a variable declaration, the variables declared in a code space is important to be considered in the algorithm. Other than variable declarations the next consideration is likely to be function declarations. These two types of declarations are the most to be the starting point of known vulnerabilities. So identifying and tracking the declared variables and functions mainly using the algorithm will covers lots of vulnerabilities. Lets see the step-by step approach.

A. *Variable Tracking*

The first step would be identifying the variable declarations inside the written code space. When considering with JavaScript, using regex the algorithm can find declarations for like 'var', 'let' and 'const' keywords which are the ways for declaring variables. Then using the variable name it can be tracked to see if the raw variable is used somewhere which is leads to a vulnerability. For example consider the below code sample.

```
const Input1 = userInput1;
document.getElementById('output').innerHTML
= Input1;
```

In here, a user variable name 'Input1' has been declared which the assigned value for 'userInput1' is unknown. Then the input is directly assigned to a *'innerHTML'* inbuilt element which is vulnerable in web development[10]. For example if the unknown userInput1 is something like in below, then the code automatically opens an alert box in the browser window saying '*XSS*' when the *document.getElementById()* runs by the compiler which we can referred to as a XSS attack.

```
const userInput1 =
"<script>alert('XSS')</script>";
```

So to mitigate the kinds of attacks that occurs due to these kind of variable declarations it's important to sanitize inputs before using them directly. To detect these kind of unsafe code patterns the variable tracking algorithm plays a major role.

B. *Function Tracking*

Same as variable tracking, function tracking is also important for detecting vulnerable code patterns. Using regex, the algorithm will detect function declarations and using function tracking it can be analyzed for unsafe use of the function in the code space. Using the variable and function tracking algorithms it is possible to cover lots of vulnerable code patterns which is related to SQLi and XSS during web development.

Apart from variable and function tracking, we can also program other algorithms for detecting CSRF vulnerabilities which initiates with API declarations, e-mail related vulnerabilities which related to email header declarations[11], and many other types.

IV. IMPLEMENTATION

With the complete algorithm which covers the identified vulnerabilities according to the research, to make the idea practical it can be implemented into a software program to analyze a code written by using a programming language(in this case JavaScript) to test the functionality of the concept. With the idea of technology behind the research as static analysis the tool should be capable of running real-time in a code space environment. When considering code space, the programming IDE s plays a major role in development stage. From the information gathered with surveys and other requirement analysis methods among university students, we asked what IDE is used by most students for their development for applications. The data shows like in Figure 4.

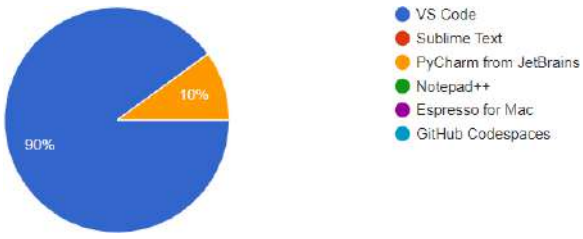


Fig. 4. Used IDE applications

It's clear that not only in our surveys the internet sources are also saying that the VS code is the most used IDE. So implementing the tool which supports VS code as the initial target will be more suitable to test the tool. The VS code offers its extension API which developers can use to implement different extensions to run on VS code IDE. This is a perfect opportunity to implement the

```
//Basic injection
//vulnerable
app.post('/login', (req, res) => {
    var username = req.body.username;
    var password = req.body.password;
    var sql = "SELECT * FROM users WHERE username = '"+username+"' AND password = '"+password+"'";

    connection.query(sql, function (error, results, fields) {
        if (error) throw error;
        // Proceed with authentication success logic
        res.send('Logged in successfully');
    });
});
```

static code analysis algorithm which programed into an extension to run it real time on the VS code IDE. This will cover lots of userbase which uses the VS code and it can be use to successfully test and improve the quality of the algorithm as well.

Other than this approach the algorithm can also be implemented with a web based approach or any other kind of preferred way.

V. TESTING AND CONCLUSION

Testing and validation are critical steps in the development process to ensure that the tool effectively identifies and helps mitigate security vulnerabilities. This chapter presents the results of testing the tool with various code snippets that contain common web security vulnerabilities. The functionality of the system was verified through these tests, demonstrating its capability to detect

```
// Normally updating user profile without CSRF protection
app.post('/api/update-profile', (req, res) => {
    const profileData = req.body;
    console.log('Profile Data:', profileData);
    res.json({ message: 'Profile updated successfully!' });
});
```

and warn about insecure code. The testing phase demonstrated that the system performs its intended functions effectively. The core functionality of the tool is to provide real-time analysis of code and detect potential security vulnerabilities. During testing, several types of vulnerabilities were introduced into the code to verify the tool's effectiveness. The tool successfully identified insecure code and provided warnings. Below are some test results displayed by vs code IDE.

A. Cross Site Scripting Detection

Scenario: Code that inserts user input into the HTML content using **innerHTML**, which is susceptible to XSS attacks.

```
xss_test.js > ...
1
2   const Input1 = userInput1;
3   document.getElementById('output').innerHTML = Input1;
4
5   document.getElementById('output').textContent = Input1;
6
7   document.getElementById('output').innerHTML = DOMPurify.encodeURI(Input1);
8
```

Fig. 5. HTML Code

Observation: The extension(tool) highlighted the insecure usage of *innerHTML* and warned about the XSS risk. It recommended using safer alternatives such as ***textContent*** or sanitizing the input.

B. SQL Injection detection

Scenario: An SQL query constructed using un-sanitized user inputs, leading to SQL Injection vulnerabilities.

Fig.6. SQL Query

Observation: The tool identified the insecure SQL query and warned about the potential SQL Injection risk. It recommended using parameterized queries to prevent this vulnerability.

C. Cross-Site Request Forgery Detection

Scenario: A sample API endpoint that updates user profiles without CSRF protection.

Fig.7. Sample API endpoint

Observation: The tool detected the lack of CSRF protection and warned about the associated risk. It recommended implementing CSRF tokens.

The testing and validation phase confirmed that the VS Code extension made using Static Analysis meets its functional requirements and provides a robust solution for enhancing web application security during development. By detecting and mitigating vulnerabilities in real-time, the tool helps developers adhere to secure coding practices and reduces the risk of security breaches in web applications. Other than the detected vulnerabilities the algorithms can be implemented to detect various vulnerability types as the vulnerability database being updated. Not only detection of vulnerabilities, Static Analysis made the detection more fasrer when compared to Dynamic Analysis in existing tools. The integration to the IDE via extension it made the tool to run real-time with the development

environment. Not only that the extension has the potential to significantly impact how new developers are introduced to security practices, making secure coding more accessible. Future enhancements could include expanding the range of detectable vulnerabilities,

supporting more programming languages, and possibly revisiting the integration of machine learning for advanced pattern recognition as well.

REFERENCES

[1] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions," *Electronics (Switzerland)*, vol. 12, no. 6. MDPI, Mar. 01, 2023. doi: 10.3390/electronics12061333.

[2] D. Kaur and P. Kaur, "Empirical Analysis of Web Attacks," in *Physics Procedia*, Elsevier B.V., 2016, pp. 298–306. doi: 10.1016/j.procs.2016.02.057.

[3] V. P. Dwivedi Scholar, C. Rits, B. Himanshu Yadav Asst, C. dept RITS, and B. Anurag Jain, "Web Application Vulnerabilities: A Survey," 2014.

[4] F. M. Tudela, J. R. B. Higuera, J. B. Higuera, J. A. S. Montalvo, and M. I. Argyros, "On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications," *Applied Sciences (Switzerland)*, vol. 10, no. 24, pp. 1–26, Dec. 2020, doi: 10.3390/app10249119.

[5] L. Dencheva and M. Monaghan, "Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools MSc Internship MSc in Cyber Security." [Online]. Available: https://www.enisa.europa.eu/publications/enisa- threat-landscape-2021

[6] S. Motogna, D. Cristea, D. Şotropa, and A. J. Molnar, "Formal concept analysis model for static code analysis," *Carpathian Journal of Mathematics*, vol. 38, no. 1, pp. 159–168, 2022, doi: 10.37193/CJM.2022.01.13.

[7] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of JavaScript," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Association for Computing Machinery, Jun. 2019, pp. 425–438. doi: 10.1145/3314221.3314645.

[8] L. Groner and G. Manricks, *JavaScript regular expressions : leverage the power of regular expressions to create an engaging user experience*.

[9] Tony Stubblebine, "Regular Expression Pocket Reference," Sebastopol, 2007.

[10] M. Johns, "Code-injection vulnerabilities in web applications - Exemplified at cross-site scripting," *IT - Information Technology*, vol. 53, no. 5, pp. 256–260, Sep. 2011, doi: 10.1524/itit.2011.0651.

[11] Stuttard, Dafydd, Pinto, and Marcus, "The Web Application Hacker's Handbook."