Journal of Theoretical and Applied Information Technology
29th February 2024. Vol.102. No 4

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

VULNERABILITY DETECTION IN SOFTWARE APPLICATIONS USING STATIC CODE ANALYSIS

DEEPAK KUMAR A[1], GNANAPRAKASAM C[2] , PRABU SANKAR N[3], SENTHAMILARASI N[4], CHENNI KUMARAN J[5], VINSTON RAJA R[6], SUSEENDRA R[7]

[1]Assistant Professor, Computer Science and Engineering, St. Joseph's Institute of Technology, Chennai,
[2]Assistant Professor, Artificial Intelligence and Data Science, Panimalar Engineering College, Chennai,
[3]Assistant Professor, Department of Information Technology, Panimalar Engineering College, Chennai,
[4]Assistant Professor, Computer Science and Engineering, Sathyabama Institute of Science and Technology
[5]Professor, Department of Computer Science and Engineering,Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences (SIMATS)
[6]Assistant Professor, Information Technology, Panimalar Engineering College, Chennai, India.
[7]Assistant Professor, Information Technology, Panimalar Engineering College, Chennai, India.

E-mail: deepakkumar@stjosephstechnology.ac.in, cgn.ds2021@gmail.com, n.prabusankar81@gmail.com, senthamilarasi.n.cse@sathyabama.ac.in, drchennikumaran@gmail.com, rvinstonraja@gmail.com, suseechandran17@gmail.com

ABSTRACT

In this modern era of technology where data and its integrity are vital for organizations, software security has become a major area to focus on in the software life cycle. Organizations must preserve the program's security to ensure the computer program's availability, authenticity, and data integrity delivered to the clients. The major focus in software security processes is to find the vulnerabilities displayed in source code prior to the production phase of the software product. Recognizing the bugs present in the code in the early stages of the software lifecycle may help resolve the vulnerability findings in the computer program and help the software developers settle those bugs. This detection process is effective at runtime, but can also be performed in the production phase where the computer program is under development and partially implemented. A static code analysis process is used to detect vulnerabilities. It can be done computerized or evaluated physically by development and testing teams. The use of source code scanning tools that are mostly automated for detecting vulnerabilities is utilized in this paper. These tools review the source code for its quality based on several code metrics and identify bugs present in the program. Unlike dynamic analysis methods, static code analysis helps find the security vulnerabilities in the initial stages of the software life cycle, where the software product is in the production phase and static analysis does not require code to be in the execution state.

Keywords: *Static Analysis, Bug Detection, Vulnerabilities, Software Security*

1. INTRODUCTION

Preserving the security of the software products have been the foremost basic tasks also deemed to be the critical ones for the organizations now-a-days. In the recent years the number of known computer vulnerabilities has grown its numbers vastly. Computer security alludes to steps taken to ensure that the program is guaranteed from attacker's malicious intents that may influence the proper functioning of the computer program. Different analysts and cyber security professionals have classified these security vulnerabilities, explained that software security vulnerability can be deemed as a flaw in the source code written by developers which in turn may provide unauthorized access to the attacker and obstruct the behavior of program. Subsequently, software developers need to find and settle these bugs found within the code before it is moved to production phase. The two main broadly classified ideas of detecting vulnerabilities are the Static code Analysis and Dynamic code Analysis. Both these processes analyze the code to discover security vulnerabilities. Several static analysis tools have been distinguished with help of the level of accuracy in predicting the bugs by them and they have their own set of pros and cons [1]. Dynamic

analysis is performed when the product is in production stage whereas the static code analysis practice involves analyzing computer program that is yet to be deployed. Many software developers and analysts have claimed that the static code analysis process is more advantageous and effective than dynamic analysis for finding security flaws in source code. It is less demanding for the software developer since the method of settling these flaws found can be done in early stages, additionally, decreases the amount of time spent and the cost put within the organization for settling these flaws. Static code analysis enjoys the utilization of computerized scanning tools or it can be done physically if needed. [2] Proposed a tool capable of detecting bugs in android app,

Although they failed to provide the methodology for analyzing android vulnerabilities. Analyzed the web apps and henceforth he proposed vulnerabilities are found in all corners of android systems. Notable works had done on various open-source static analysis tools made exclusive for Java files. Researchers have developed their JAVA applications and intentionally added vulnerabilities to detect them.

What is static code analysis?

Static code analysis, also known as source code analysis or static code review, is the process of detecting bad coding style, potential vulnerabilities, and security flaws in a software's source code without actually running it, a form of white-box testing. Static code analysis will enable your teams to detect code bugs or vulnerabilities that other testing methods and tools, such as manual code reviews and compilers, frequently miss. The fast feedback loop is a key tenet of the DevOps movement. Static code analysis helps you achieve a quick automated feedback loop for detecting defects that, if left unchecked, could lead to more serious issues.

Static code analysis is not only useful for checking code styles; it can also be used for static application security testing (SAST).

At a high level, a static code analyzer examines source code and checks for:

- Code issues and security vulnerabilities
- Quality of documentation
- Consistency in formatting with overall software design
- Compliance with project requirements, coding standards, and best programming practices
- Violations of rules and conventions that affect program execution and non-functional quality aspects of a software system such as complexity and maintainability

2. RELATED WORK

There has been much inquire about the bug-prediction field focused on diverse viewpoints utilizing different granularity levels and different strategies [3]. Previous studies can be classified according to different aspects, such as work on source code features [4] or the bug description. Considering the source code granularity aspect, the research works has been in file level [5]. In predicting bug severity, most existing works carryout bug reports using natural language techniques or various classical and deep neural network models. Further discussions on work related to general bug prediction, fault severity prediction, and static analysis tools

Detection

The most important part of detection is to avoid getting used to failing dependency checks. Ideally, the build should fail if there is a vulnerable dependency detected. To be able to enable that, the resolution needs to be as painless and as fast as possible. No one wants to encounter a broken pipeline due to a false positive. Since the Open Worldwide Application Security Project (OWASP) dependency check primarily uses the National Vulnerability Database (NIST NVD) database, it sometimes struggles with false positives. However, as has been observed, false positives are inevitable, as the analysis is only occasionally straightforward.

Analysis

This is the hard part and actually, the one when tooling can't help us much. Consider the SnakeYAML remote code execution vulnerability as an example. For it to be exploitable, the library would have to be used unsafely, such as parsing data provided by an attacker. Regrettably, no tool is likely to reliably detect whether an application and all its libraries contain vulnerable code. So, this part will always need some human intervention.

Resolution

Upgrading the library to a fixed version is relatively straightforward for direct dependencies Tools like Dependabot and Renovate can help in the process. However, the

tools fail if the vulnerable dependency is introduced transitively or through dependency management. Manually overriding the dependency may be an acceptable solution for a single project. In cases where multiple services are being maintained, we should introduce centrally managed dependency management to streamline the process.

3. BUG PREDICTION

Bug prediction studies have employed various code metrics, including LOC [6], McCabe, Halstead and C&K metrics, at different levels of granularity. These levels range from package/class level to method [7] and line level. While package/class level provides a higher granularity, it can be impractical for developers [8] due to the significant effort required to locate bugs. Analysis made with line-level granularity often result in providing too many false-positives since multiple line might have repeated occurrence. Method level granularity has become the primary focus of the research community, especially in the development of bug prediction models [9]. Studies that made using this approach have given out positive results. Overall, the method level provides a suitable balance between granularity and practicality for developers in predicting and locating bugs [10].

4. BUG SEVERITY PREDICTION

[11] Previously took advantage of bug reports and their severity labels to suggest accurate severity labels for reported bugs using nearest neighbor classification used nearest neighbor classification. The authors have significantly improved the f-maximum measurement.[12] proposed a method using deep learning modeling, natural language techniques and sentiment analysis using error reports to predict bug severity. They mentioned that this method improved the f-measurement of the peak approaches by an average of 7.90%. [13] proposed bug classification and bug severity prediction using topic modeling. Their assessment of 30,000 bugs reported on popular IDE forums like eclipse and netbeans projects demonstrates the effectiveness of their approach in predicting severity.

5. STATIC ANALYSIS TOOLS FOR PREDICTION

Many researchers evaluated the real effectiveness of static analysis tools of error detection tasks. Some scholars have used these tools as a prophecy for the techniques provided by them. For example, Tomassi used SpotBugs and ErrorPone to detect errors in the BugSwarm dataset, and they found that these tools were not very effective at finding bugs because their results showed only one detection. error.[14] leveraged FindBugs has to find bugs in Google's internal codebase, and they found that integrating the tool into Google's Mondrian code review system would help developers see the viability of the code. error in the code. [15] studied the tools SpotBugs, Infer, and Google Error Prone to find out their real detection of Java errors. They concluded that these tools were complementary and missed most of the errors. Dura et al. [6] introduced JavaDL, a declarative, log-based specification language for detecting error patterns in Java code and comparing it with the SpotBugs and ErrorPone tools. The authors found that JavaDL has comparable performance to these engines. Habib and Pradel proposed a method that treats error detection as a classification problem using a neural network and using the Google Error Prone as a prophecy.

6. SOURCE CODE METRICS

Various source code metrics with module, class and method level granularity were used to measure software quality and predict errors in previous studies [16]. Are these code metrics different? Advantages (e.g., fast computation) and disadvantages (e.g., requires a language-specific parser). Some of these metrics are introduced to overcome the weaknesses of the previous code metrics. For example, McClure proposed to improve McCabe [17] by considering the number of control variables. In this research, we use most of the popular method-level code indicators used in the previous section research (mentioned in Related Work) to see the predictability of error codes and Gravitation. Although the list of selected stats is not exhaustive (with many being explored in this section domain and test size limits in one article), but we guarantee most metrics has previously been shown to be effective in predicting error at the method level. These metrics include [18],

7. LINE OF CODE (LC)

Dimensions or lines of code (LC) eases the analysis process for most of the tools and regarded as the effective code metric compared to others. Using LC as proxy, the retention index is so prominent that there are dedicated studies that focus exclusively on LC and correlated with

other quality measures. LC has been broadly studied for error prediction, troubleshoot and find vulnerabilities. In this study, we calculate the LC as source code lines without comments and blank lines, similar to prevent code formatting and comments on impacts, which are beyond the scope of this study[19].

8. MCCABE (MA)

McCabe, also known as cyclic complexity, is another widespread measurement which indicated as the count of possible independent paths present, hence complexity of a program components with high McCabe values are more prone to errors. Extensive studies have been conducted on McCabe to find errors and to locate suspicious code [18], to understand its correlation with code quality and to take advantage of its value for test creation methods, such as structured testing (path testing). McCabe can be calculated like 1 + #predicates[20].

9. MCCLURE (ML)

McClure has been suggested as an improvement over McCabe. In contrast to McCabe, McClure considers the number of control variables and the number of comparisons in a predicate, which is not supported by McCabe. Intuitively, a predicate that has many comparisons and many control variables will be complex and thereby increases the chances of errors than a predicate with only one comparison or a single control variable [21].

10. PROXY INDENTATION (PI)

McCabe-like complexity metrics require a language- specific parser (to find predicates), suggested the proxy indentation index as a proxy for the likeness of McCabe complexity index. It has been shown that, to measure complexity, indentation can be performed analogous to more complex metrics like McCabe, without requiring a language- specific parser. The indentation measurement is performed for each row, then the aggregate value is calculated for the whole program element (for example, a method). Hindle et al. shows that the standard deviation as a composite value exceeds the mean, median, or maximum [22].

10.1. Nested Block Depth (NBD)

According to indices above, there is no difference between two pieces of code containing two identical for loops if they are serialized or in nested arrangement. NBD has been studied with McCabe and McClure to reduce this problem.

10.2. FanOut(FO)

FanOut Metric computes the total number of methods called for a given method. This provides an estimate of coupling, i.e., the dependence of a particular method on other methods. It is observed that tightly coupled code components are less maintainable and error-prone.

10.3. Readability (R)

This metric combines different code features to calculate a value to estimate code readability. We used the readability metric suggested to creates readability note for a certain method. Readability scores range from 0 to 1 to specify maximum code that can be read at least readable code, respectively. The authors conclude that this metric is significantly correlated with errors, code rotation, and self- reported stability.

10.4. Halstead Measurement

It contains a total of seven metrics which depends on total operators and operands present in the component. Various studies for code measurement and to calculate the complexity of software maintenance tasks and to estimate the readability of software, uses the Hallstead metrics. they have high correlation with one another, we study the following two indicators:

Difficulty (D) and Effort (E):

n1 indicates total distinct operators, n2 indicates total distinct operands and N2 is counted as the total operands.

The **Halstead Effort** is calculated as:

$$E = D + V$$
$$V(HalstedVolume) = N * log_2(n)$$
$$N = N1 + N2$$
$$n = n1 + n2$$

10.5. Maintenance Index (MI)

The maintenance index was introduced by Omran and Hagemeister [1] in which the authors identified metrics that assists in measurement of maintainability of a software system and incorporated these indexes into a single value. This metric has evolved and has been adopted by popular tools like Visual Studio. MI can be calculated as follows

$$171 - 5.2 \times ln(Halstead\ Volume) - 0.23 \times (McCabe) - 16.2 \times ln(LC)$$

Where Halstead volume and LC are defined previously in this section.

Code metrics are classified into five classes in basis of what they measure such as complexity, coupling and cohesion, object inheritance, and size of the code. A brief description of these categories is provided in the Table 1. Size of code is found to be the most effective measure of source code. In measuring the size, the total lines of code are a useful metric. Yet it suffers few disadvantages. For example, assume that the same functionality is written multiple times the complexity remains the same but additional lines of code impacts the prediction. Few other metrics can be of use in addressing this issue. A source file complexity measure is hypothesized to influence the modifications that can be made and the maintainability of software.

10.6. Halstead Program Length

Hallstead's first hypothesis states that the length of a well-structured program is only dependent on the unique operators and operands. The total number of operators and operands used in the program can be expressed as N = N1 + N2.The estimated program length is denoted by N^ and can be calculated using the following formula: N^ = n1log2n1 + n2log2n2, where n1 is the count of unique operators, and n2 is the count of unique operands.

There are several alternate expressions available for estimating program length:

$$N_J = log_2(n1!) + log_2(n2!)$$

$$N_B = n1 * log_2 n2 + n2 * log2\ n1$$

$$N_C = n1 * sqrt(n1) + n2 * sqrt(n2)$$

$$N_S = (n * log_2 n)/2$$

$$D = \frac{n1}{2} * \frac{N2}{n2}$$

10.7. Programming Effort (E)

Programming effort (E) is measured in elementary mental discriminations. The programming effort can be expressed as the ratio of program volume (V) to program level (L), where L ranges between zero and one, with L=1 representing a program written at the highest possible level.

The program difficulty (D) is proportional to the number of unique operators in the program. Thus, the programming effort can be calculated as the product of program volume and program difficulty, that is, E=V/L=D*V.

10.8. Size of Vocabulary (n)

The size of the vocabulary of a program refers to the number of unique tokens used to create the program.

- This vocabulary can be broken down into two categories: operators and operands.
- The number of unique operators is denoted by n1, while the number of unique operands is denoted by n2.

The total size of the vocabulary, represented by n, can be calculated by n1+n2. The Potential Minimum Volume, denoted as V*, refers to the smallest possible size a program could be in order to solve a problem using a specific vocabulary.

The formula to calculate V* is: V* = (2 + n2*) * log2 (2 + n2*)

where n2* is the count of unique input and output parameters used in the program. By calculating V*, we can determine the minimum size required to solve a problem using the specified vocabulary.

10.10. Language Level

The Language level is a metric that demonstrates the level of implementation of the algorithm in the program's language. When the same algorithm is written in a low-level programming language, it may require additional effort to implement. In comparison, it is easier to write a program in high-level programming languages such as Pascal than in low-level languages like Assembler.

The language level metric can be calculated as follows:

$$L' = V / D / D$$

Another metric, lambda (λ), can be used to estimate the programming effort of the implementation. It is calculated by multiplying the estimated program level (L) and the potential minimum volume (V*) of the program. In addition, a lambda can be expressed as the product of the square of L and V, i.e., λ = L^2 * V.

10.9. Potential Minimum Volume

TABLE 1: LIST OF STUDIED METRICS AND THEIR BRIEF DESCRIPTION

METRIC	DESCRIPTION
LC	Number of source code lines without comments and blank lines
MA	Number of independent paths (logical complexity)
ML	Total variables and total comparisons in a predicate
NBD	Counting the depth of the most nested block
PI	Counting indentation of source code lines
FO	Counting the total number of methods called a given method
R	Measuring the readability of the code in the range of 0-1 (least to
D	Difficulty in writing or understanding the code
E	Effort in developing the code

11. METHODOLOGY

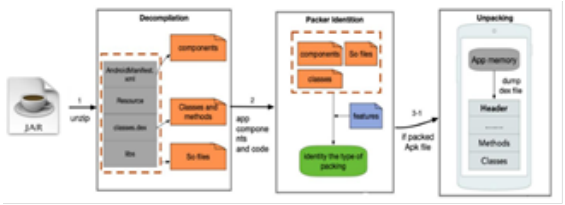

Figure 1. Architecture Diagram

Once development is complete prior to the deployment phase it has to be tested. Using static analysis tools in development stage eases the maintainability of software and settling the bugs in it. An integral part of adopting a static analysis strategy during the development of a software product is a static analysis contained within an integrated development environment (IDE) that is developed either programmatically or externally initiated at specific interval periods. Development environments often require creating its own static analysis with help of plugins report. [23] Developers can review the detected errors and fix them proactively to move into the next stage. Most static analysis tools are capable of working alongside in an integrated development environment [24]. Anyhow, these tools are often used post development process and strategies need to be applied for analyzing performance beyond what is a completed software product. Reviewers analyses your code to find bugs and policy violations. Querying or emphasizing through the model is done looking for specific properties or patterns that indicate a bug. Sophisticated symbolic execution techniques examine paths through control flow graphs. A data structure that represents a path that can be traversed during program execution. When Path Scouting detects an anomaly, an alert is generated. To model and explore an astronomical number of situational combinations, these automated analysis tools use a variety of strategies to ensure scalability. For example, step summaries are refined and compressed during analysis, and paths are examined in an order that minimizes paging.

Initially the code undergoes a serious of complex transformation in such a way that the computer understands it prior to the code execution. It is shown that in Figure 1 the analyzer feeds the output of these stages[25].

11.1. Scanning

Compiler on execution it breaks down the code into smaller pieces called tokens. Through tokens which are similar to words in this context it understands the code. A token may contain a single character or a reserved keyword for that language (like class in Java). Trailing space and program semantics such as comments are often discarded by scanners.

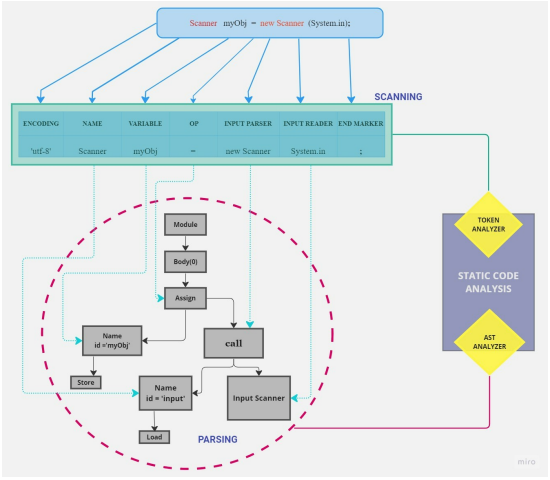

Figure 2. An illustration of scanning and parsing of source code

```
Procedure: Scanner(file)
InputFile ← read(file)
for each line in InputFile tokens = list[]
   for each char in line
      if (char = whitespace)
```

```
        tokens.append(word)
      endif endfor
   return tokens endfor
end
```

11.2. Parsing

In this stage, Figure 2 tokens are merely like simple words in a language they have to be constructed to get the grammar of a language. Parser identifies these tokens and validates them so that in a sequence they provide the grammar and organized in a tree like fashioned structure called Abstract Syntax Tree[26]. The tree itself focuses only on the logical structure of the program and the least insignificant details like indentation and parenthesis are abstracted.

```
Procedure:
parser(toke
ns)
Load JavaParser from JavaPrser
Library
Javaparser(tokens)
  for each token in tokens
    if (token is in keywords)
      construct
    AbstractSyntaxTree else
      pass endif
    return    AbstractSyntaxTree
  endfor
end
```

11.3. Analyzing

Based on the logical complexity and size of a code the syntax tree gets vast and complex in nature thereby it is difficult in writing code to analyses it. There are tools that ease this process which resembles the same properties and tasks that compilers perform [27]. The syntax tree is parsed and the static analysis tools look for a specific pattern in the code that results in vulnerabilities. Various taint paths that lead to flaws are captured and using a bug severity model and bug filter they can be filtered based on the bug severity.

```
Procedure: analyse(AST)
Load BugPatterns from Library
read(AST)
if (AST in BugPatterns)
  get.BugPattern(BugDescription)
  print(BugDescription)
endif
end
```

11.4. Bug Filters and Severity Ranking

Bug filter is capable of matching instances in context to certain criteria. A filter can select a set of bug instances for including or excluding them in a report. Usually, they are used to exclude instances of a bug. The filters can also be conditioned to combine two or more filter types by following clauses Or, And Not. Certain match clauses are used to filter as follows,

<Bug> It specifies a particular pattern or patterns to match a bug instance.

For example, BAD_PRACTICE.

<Rank> The Rank element matches bug with having at a specified bug rank. The values range in between 1 and 20 where 1 to 4 are critical, 5 to 9 is high, 10 to 14 is medium and rest is deemed to be low risk factored bugs. <Class> This enables filtering bugs associated within a specified class. It takes the class name as an attribute to filter. The inputs are broken into tokens using various patterns by scanners.

11.4.1. Application Misconfiguration: Extreme Authorizations

An application may use custom permissions that can then allow a separate application to access hardware level functionality through its API. These separate applications can bypass the normal prompting procedures for use of sensitive functionality by using the API. Applications should only request the minimum permissions needed for stated application functionality. Do not request any unnecessary permission. Any unused permissions should not be requested. Future application updates should prompt the user to revoke unneeded permissions.

11.4.2. Application Misconfiguration: Global Error Handling Disabled

Disabling a global error handling mechanism increases the risk that verbose implementation details will be revealed to attackers through a stack trace. To minimize the risk of disclosing sensitive implementation details through error messages, ensure the application deployment descriptor declares an error-page declaration that catches all uncaught exceptions thrown by the application.

The web.xml should define error handling elements such as:

<error-page>
<error-code>500</error-code>
<location>/path/to/default_500.jsp</location>
</error-page>
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/path/to/default_exception_handler.jsp</location>
</error-page>

11.4.3. Cross-Site Scripting ("XSS")

Cross-site scripting (sometimes referred to as "XSS") vulnerabilities occur when an attacker embeds malicious client-side script or HTML in a form or query variables submitted to a site via a web interface, sending the malicious content to an end-user. If this content is submitted by one user (the attacker), stored in the database, and subsequently rendered to a different user (the victim), a Persisted Cross Site Scripting Attack occurs.

The most reliable means of thwarting most types of XSS attacks is to HTML-encode or URL-encode all output data, regardless of the data source. It is important to note that there are different output contexts which encoding functionality must handle, including HTML, HTML attributes, URLs, CSS, and JavaScript. A single encoding approach will not necessarily mitigate XSS in every context.

If output encoding isn't practicable, the next most effective approach is to carefully filter all input data against a white- list of allowed characters. This approach does have the advantage that it can be performed externally without modifying application source code. Data retrieved from third- parties or shared with other applications should be filtered along with user input data. The white-list should only include characters which may be a legitimate part of user input. The following characters are especially useful for conducting XSS attacks and should be considered in any encoding or filtering scheme: < > " „ ; & ? One or more of these characters, such as the single quote, may be required by the application.

XSS Using Script in Attributes
XSS attacks may be conducted without using <script>...</script> tags. Other tags will do exactly the same thing, <body onload=alert('test1')> or other attributes like: onmouseover, onerror.
onmouseover
<b onmouseover=alert('Wufff!')>click me!
onerror
<img src="http://url.to.file.which/not.exist" onerror=alert(document.cookie);>

XSS Using Script Via Encoded URI Schemes
If we need to hide against web application filters we may try to encode string characters, e.g.: a=&\#X41 (UTF-8) and use it in IMG tags:
<IMG SRC=j&#X41vascript:alert('test2')>
There are many different UTF-8 encoding notations that give us even more possibilities.

XSS Using Code Encoding
We may encode our script in base64 and place it in META tag. This way we get rid of alert() totally. More information about this method can be found in RFC 2397
<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">

11.4.4. Cryptography: Improper Pseudo-Random Number Generator Usage

If the application provides a switch to enable debug mode in production, attackers could guess or learn of this parameter and take advantage of any additional information the application may provide. Custom debug mode implementations have even been observed to bypass authentication or assign administrator-level permissions for testing purposes.

Production code should normally not be capable of entering debug mode or producing debug messages. However, if this capability is necessary, debug mode should be triggered by editing a file or configuration option on the server. In particular, debug should not be enabled by an option in the application itself. For example, it should not be possible to pass in a URL parameter to trigger debug mode, Regardless how obscure the parameter may be, it is never a secure option. Insufficient randomness results when software generates predictable values when unpredictability is required. When a security mechanism relies on random, unpredictable values to restrict access to a sensitive resource, such as an initialization vector (IV), a seed for generating a cryptographic key, or a session ID, then use of insufficiently random numbers may allow an attacker to access the resource by guessing the value. The potential

consequences of using insufficiently random numbers are data theft or modification, account or system compromise, and loss of accountability – i.e., non- reputation.

When using random numbers in a security context, use cryptographically secure pseudo-random number generators (CSPRNG).

```
byte[] randomBytes = new byte[8];
SecureRandom random = new
SecureRandom();
random.nextBytes(randomBytes);
```

11.4.5. Application Misconfiguration: Debug

Application errors commonly occur during normal operation, particularly when the application is misused, even unintentionally. If debugging is enabled, then, when occurs occur, the application may provide inside information to end- users who should not have access to it and who may use it to attack the application. Error messages displayed to an end user could include server information, a detailed exception message, a stack trace, or even the actual source code of the page where the error occurred. This information could be used to help formulate an attack.Frameworks and components used by the application may have their own debug options as well. It is important that debug options are disabled throughout the application before the application is deployed.

There are many instances were a Debug mode may exist within a Java application, and this varies depending on the container. Here is one example of debug mode disabled for the jsp servlet:

```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-
class>oracle.jsp.runtimev2.JspServlet</servlet-
class>
<init-param>
<param-name>debug_mode</param-name>
<param-value>false</param-value>
</init-param>
```

Since developers may have implemented their own custom debug mode, be sure to inspect configuration files and search the code base for things like:
Debug Mode
debug = true debug = 1
debug

11.4.6. Disclosure: Clear Text Password

If an application contains one or more hardcoded passwords within the source code, an attacker with access to the source code or compiled binaries can extract the credentials in an attempt to access the corresponding services. Obscuring passwords using encoding, such as Base-64, is not sufficient. Storing passwords in clear text (e.g. in an application's properties or configuration file) can result in account or system compromise. This exposes the password to any personnel with access to the application's configuration files developers, architects, testers, auditors, and development managers. Because it is possible that others may have access to a user's password, the owner of an account can no longer be presumed to be the only person able to login to the account.

Solution

System passwords should be encrypted, or the configuration file they are contained within should be encrypted, whenever possible.

Credentials should be encrypted with a key and stored on disk in a non-web-accessible directory, read-only accessible to the user running the web application (webserver). The key should be stored in a separate non-web-accessible location that is also read-only to the user running the web application. The application can then read the key (from a known static location), read the encrypted credentials, decode, and use them. The encryption key should be rotated on a 30- to 90- day basis.

User passwords should be stored using a strong one-way hashing algorithm, such as SHA-256. A cryptographic salt should be added to each password before it is hashed. The salt should be at least 64-bits in length and should be random or unique to each user.

There are many approaches and libraries available for encrypting/decrypting data in Java. Many common approaches are less than secure. Java developers often encode system passwords in Base-64 or encrypt them with DES - neither approach is secure, especially encoding.

The following code can be used to encrypt/decrypt using a secure algorithm, AES:

```
public static String encrypt(String value,
File keyFile)
```

```
throws GeneralSecurityException, IOException
 {
   if (!keyFile.exists())
    {
```

The application makes use of untrusted data in conjunction with the creation and or use of an interpreter. Untrusted data is retrieved from the attacker and utilized as an argument to a dangerous interpreter access method. Failure to properly validate or encode data utilized by an interpreter increases the risk of injection attacks. Such injection typically results in the attacker's ability to execute arbitrary code in the context of the program consuming the interpreter results.

Define and enforce a strict set of criteria defining what the application will accept as valid input, and contextually encode all untrusted data passed to the interpreter prior to execution.

11.4.8. Denial of Service (DoS): Readline

The java.io.BufferedReader readLine() method can be used to read data from a socket or file; however, readLine() reads data until it encounters a newline or carriage return character in the data. If neither of these characters are found, readLine() will continue reading data indefinitely. If an attacker has any control over the source being read, he or she can inject data that does not have these characters and cause a denial of service on the system. Even if the number of lines to be read is limited, an attacker can supply a large file with no newline characters and cause an OutOfMemoryError exception.

OWASP's Enterprise Security API provides a safer alternative to readLine() called SafeReadLine(). This method reads from an input stream until end-of-line or the maximum number of characters is reached, effectively mitigating this risk.

Another solution is to override both BufferedReader and the readLine() method and implement a limit for the maximum number of characters that can be read. In the absence of a more secure method, avoid taking input from the client whenever possible and ensure data being read is trusted.

```
 KeyGenerator keyGen =
KeyGenerator.getInstance(Crypto
Utils.AES);
        keyGen.init(128);
       SecretKey sk = keyGen.generateKey();
       FileWriter fw = new FileWriter(keyFile);
       fw.write(byteArrayToHexString(sk.getE
       ncoded())); fw.flush();
        fw.close();
    }
            SecretKeySpec sks =
           getSecretKeySpec(keyFile); Cipher
           cipher =
          Cipher.getInstance(CryptoUtils.AES);
          cipher.init(Cipher.ENCRYPT_MO
         DE, sks, cipher.getParameters());
            byte[] encrypted =
         cipher.doFinal(value.getBytes());
            return
           byteArrayToHexString(encrypted);
 }
```

11.4.7. Injection: Unknown Interpreter

OWASP ESAPI's safeReadLine() can be used to safely read untrusted data as follows.

```
ByteArrayInputStream s = new
ByteArrayInputStream("testinput".g
etBytes()); Validator instance =
ESAPI.validator();
try
{
        String u = instance.safeReadLine(s, 20);
}
catch (ValidationException e)
{
        // Handle exception
}
```

11.4.9. URL Redirector Abuse

Applications frequently redirect users to other pages using stored URLs. Sometimes the target page is specified in an untrusted parameter, allowing attackers to choose the destination page or location. Such redirects may improperly leverage the trust the user has in the vulnerable website.

If untrusted data becomes part of a redirect URL, ensure that the supplied value has been properly validated and was part of a legal and authorized request from the user. It is recommended that legal redirect destinations be driven by a “destination id” that is mapped to the actual redirect destination server-side, rather than the actual URL or portion of the URL originating from the user request. Lookup-maps or access controls tables are best for this purpose.

11.4.10. Insufficient Session Expiration

PCI Data Security Standards Version 3, Section 8.1.8 specifies a maximum session timeout of 15 minutes for critical components of an application: "If a session has been idle for more than 15 minutes, require the user to re-authenticate to re-activate the terminal or session."

User sessions with long or no inactivity timeouts may help attackers replay attacks or hijack sessions. Social engineering attacks are also more likely to succeed with a longer time- out. An attacker has a greater opportunity to gain physical access to a user's machine if a user does not close the application.

If the session timeout is not specified in a web application's configuration, the default value will be used, which is often 24, 30, or 60 minutes, depending on the web server, version, and its configuration.

In general, idle user sessions should timeout within 15-20 minutes, or less for sensitive applications. Consider disabling "sliding expiration" if the configuration option exists. If it is necessary to enable this option, consider implementing a hard session timeout in addition to the sliding timeout. When sessions timeout, the application should invalidate the session, removing session data as well as any cookies and authentication tokens.

The session timeout can be configured at the server-level in the default web.xml or for each web application individually. The following code should be included in the application's web.xml file:

```
<session-config>
<session-timeout>15</session-timeout>
</session-config>
```

If WebLogic is being used, the session timeout should also be specified in the weblogic.xml file. The following code sets the timeout to 15 minutes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<weblogic-web-app
xmlns="http://www.bea.com/ns/weblogic/90">
<session-descriptor>
<timeout-secs>900</timeout-secs>
</session-descriptor>
</weblogic-web-app>
```

11.4.11. Missing Access Strategy

This application is not utilizing an access control strategy for one or more components. Failure to utilize access control can lead to exposure of sensitive functionality to unintended users. Malicious users seek out this type of functionality to cause harm to users of the application, or the application itself.

In Websphere, if you enable servlets by class name, then this is performing the same act as Android in that it allows you to invoke by the class. If the following snippet exists or the variable is not declared, this allows you to invoke servlets without any permissions: enable-serving-servlets-by-class-name value="true"

Utilize an access control strategy for all components of the application where sensitive functionality may reside. Prevent servlets from serving by class name by adding the following line:

enable-serving-servlets-by-class-name value="false"

12. RESULT AND DISCUSSION

As future work we are going to work on the elaboration of a representative benchmark for web applications including a wide set of security vulnerabilities that permits an AST tools comparison. Figure3 The main objective is that the evaluation of each combination of different tools can be the most effective possible using the methodology proposed in this work, allowing one to distinguish the best combinations of tools considering several levels of criticality in the web applications of an organization

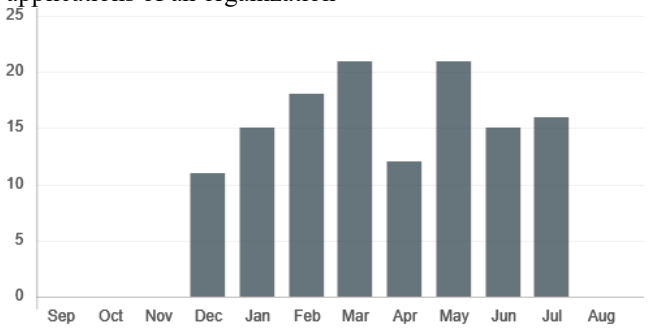

Figure3: Security vulnerabilities that permits an AST tools comparison

One of the important constructing blocks of software is code quality. To Enhanced software quality is directly linked to high-quality code. The quality of this code correlates with your application is secure, stable, and reliable. To endure quality, many improvement teams embrace techniques like code review, automated testing,

and manual testing.

During the code review and computerized tests are important for producing the best quality code; because code referees and automated test authors are humans, bugs and security vulnerabilities often find their way into the development environment. Figure4. According to the State of Cloud Native Application Security Report, misconfiguration, and known unpatched vulnerabilities were responsible for the greatest number of security incidents in cloud native environments.

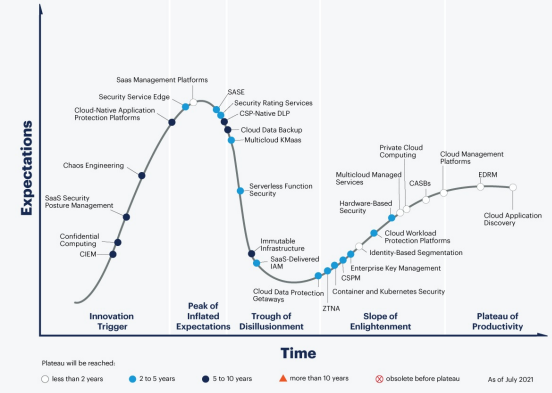

Figure4. Cloud Native Application Security Report

Source code analysis could prevent half of the problems that often slip through the cracks in production. Rather than putting out fires caused by bad code, a better approach would be to incorporate quality assurance and enforce coding standards early in the software development life cycle using static code analysis.

13. CONCLUSION

Bug detection and correction remains the main maintenance activity in software product development life cycle. However, among many bugs that are likely to exist in high-severity codes are of great interest to developers because their consequences are the most important. Unfortunately, most of the existing vulnerability detection studies treat all bugs equally and ignore their severity. In this article, [28] we have studied 10 source code metrics and automated static analysis tools, as popular methods to predict buggy codes, to find the possibility of estimating bug severity respectively in most cases. At last, the code metrics and static analysis tools exploit different characteristics of code,[29] so they can complement each other in predicting bug severity. We found that there was no relationship between code complexity and error severity, and static analysis tools miss many errors due to lack of complex deterministic model. Manual inspection of critical errors reveals Security and Edge/Boundary failure types with high and low severity [30] and [31] Potential future prediction directions of this study investigate the power of dynamic analysis and testing in estimating defect severity. Furthermore, one can use the conclusions made through this study to relate the static analysis[32][33][34][35]and [36] tools along with their limitations and try enriching their rule set to better identify critical bugs[37].

REFERENCES

[1]. Servant, F., Jones, J.A.: Fuzzy fine-grained code-history analysis. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 746–757 (2017)

[2]. Wahono, R.S.: A systematic literature review of software defect prediction. *Journal of software engineering* 1(1),1–16 (2015)

[3]. Habib, A., Pradel, M.: Neural bug finding: A study of opportunities and challenges. *arXiv* preprint arXiv:1906.00307 (2019)

[4]. Matter, D., Kuhn, A., Nierstrasz, O.: Assigning bug reports using a vocabulary-based expertise model of developers. In: 2009 *6th IEEE International Working Conference on Mining Software Repositories*, pp. 131–140 (2009)

[5]. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Using findbugs on production software. In: *Companion to the 22nd ACM SIGPLAN conference on Objectoriented programming systems and applications companion*, pp. 805–806 (2007)

[6]. Dura, A., Reichenbach, C., S¨oderberg, E.: Javadl: automatically incrementalizing java bug pattern detection. *Proceedings of the ACM on Programming Languages 5(OOPSLA), 1–31 (2021)*

[7]. Tian, Y., Lo, D., Sun, C.: Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 2012 *19th Working Conference on Reverse Engineering, pp. 215–224. IEEE (2012)*

[8]. Ramay, W.Y., Umer, Q., Yin, X.C., Zhu, C., Illahi, I.: Deep neural network-based severity prediction of bug reports. *IEEE Access 7, 46846–46857 (2019)*

[9]. Mo, R., Wei, S., Feng, Q., Li, Z.: An exploratory study of bug prediction at the method level. *Information and software technology* 144, 106794 (2022)

[10]. Jureczko, M., Spinellis, D.: Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wroc lawskiej* pp. 69–81 (2010)

[11]. On analyzing static analysis tools", *National security Agency Centre for Assured Software*, July 26, 2011, pp. 1-13

[12]. A. Amin, A. Eldessouki, M. T. Magdy, N. Abdeen, H. Hindy, and I. Hegazy, „„Androshield: Automated Android applications vulnerability detection, a hybrid static and dynamic analysis approach,"" *Information,* vol. 10, no. 10, p.326, 2019.

[13]. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering (PROMISE"07: *ICSE Workshops 2007*), pp. 9–9. IEEE (2007)

[14]. Pascarella, L., Palomba, F., Bacchelli, A.: On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software* 161, 110493 (2020)

[15]. Khatiwada, S., Tushev, M., Mahmoud, A.: Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology* 93, 45–57 (2018)

[16]. E. Chin and D. Wagner, „„Bifocals: Analyzing Web view vulnerabilities in Android applications,"" in InformationSecurity Applications, Y. Kim, H. Lee, and A. Perrig, Eds. Cham, Switzerland: *Springer, 2014, pp. 138–159.*

[17]. Zhang, T., Chen, J., Yang, G., Lee, B., Luo, X.: Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software* 117, 166–184 (2016)

[18]. P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, „„A largescale study of mobile Web app security,""in Proc. *Mobile Secur. Technol. Workshop* (MoST), 2015, pp. 1–8.

[19]. Tomassi, D.A.: Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In: Proceedings of the 2018 26th *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 980–982 (2018)

[20]. McClure, C.L.: A model for program complexity analysis. In: Proceedings of the 3rd *international conference on Software engineering*, pp. 149–157 (1978)

[21]. Habib, A., Pradel, M.: How many of all bugs do we find? a study of static bug detectors. In: 2018 33rd *IEEE/ACM International Conference on Automated Software Engineering (ASE),* pp. 317–328. IEEE (2018)

[22]. Antinyan, V., Staron, M., Sandberg, A.: Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* 22(6), 3057–3087 (2017)

[23]. Yuan, H., Zheng, L., Dong, L., Peng, X., Zhuang, Y., Deng, G. (2020). Research and Implementation of Security Vulnerability Detection in Application System of WEB Static Source Code Analysis Based on JAVA. In: Xu, Z., Choo, KK., Dehghantanha, A., Parizi, R., Hammoudeh, M. (eds) Cyber Security Intelligence and Analytics. CSIA 2019. *Advances in Intelligent Systems and Computing,* vol 928. Springer, Cham. https://doi.org/10.1007/978-3-030-15235-2_66

[24]. Arvinder Kaur, Ruchikaa Nayyar, A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code, Procedia *Computer Science,* Volume 171, 2020, Pages 2023-2029, ISSN1877-0509, https://doi.org/10.1016/j.procs.2020.04.217.

[25]. Mateo Tudela, F.; Bermejo Higuera, J.-R.; Bermejo Higuera, J.; Sicilia Montalvo, J.-A.; Argyros, M.I. On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications. Appl. Sci. 2020, 10, 9119. https://doi.org/10.3390/app10249119

[26]. Vishruti V. Desai, & Vivaksha J. Jariwala. (2022). Comprehensive Empirical Study of Static Code Analysis Tools for C Language. *International Journal of Intelligent Systems and Applications in Engineering*, 10(4), 695–700. https://ijisae.org/index.php/IJISAE/article/view/2342

[27]. N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty and Y. Xiong, "Project Achilles: A Prototype Tool for Static Method-Level Vulnerability Detection of Java Source Code Using a Recurrent Neural Network," 2019 34th *IEEE/ACM International Conference on Automated Software Engineering* Workshop (ASEW), San Diego, CA, USA, 2019, pp. 114-121, doi: 10.1109/ASEW.2019.00040.

[28]. Authors: Midya Alqaradaghi Alqaradaghi.Gregory Morse, and Tamás Kozsik

Detecting security vulnerabilities with static analysis – A case study Pages: 1–7 Online Publication Date: 03 Dec 2021 Publication Date: 07 Jun 2022, DOI: https://doi.org/10.1556/606.2021.00454

[29]. Richard Amankwah, Jinfu Chen, Heping Song, Patrick Kwaku Kudjo Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites published:29-12-2022 https://doi.org/10.1002/ spe.3181

[30]. Filus K, Boryszko P, Domańska J, Siavvas M, Gelenbe E. Efficient Feature Selection for Static Analysis Vulnerability Prediction. *Sensors (Basel)*. 2021 Feb 6;21(4):1133. doi: 10.3390/s21041133. PMID: 33561957; PMCID: PMC7915846.

[31]. Vinston Raja R., Ashok Kumar K.Financial derivative features based integrated potential fishing zone (IPFZ) Future forecast (2023) *Journal of Intelligent and Fuzzy Systems*, 45 (3), pp. 3637 - 3649, DOI: 10.3233/JIFS-231447

[32]. Vinston Raja R, Deepak Kumar A, Prabu Sankar N, Senthamilarasi N, Dr. Chenni Kumaran J., Prediction and Distribution of Disease Using Hybrid Clustering Algorithm in Big Data, *International Journal on Recent and Innovation Trends in Computing and Communication*, ISSN: 2321-8169 Volume: 11 Issue: 10,DOI: https://doi.org/10.17762/ijritcc.v11i10.8469

[33]. Vinston Raja R, Deepak Kumar A, Prabu Sankar N, Chidambarathanu K, Thamarai I, Krishnaraj M, Irin Sherly S., Comparative Evaluation Of Cardiovascular Disease Using Mlr And Rf Algorithm With Semantic Equivalence., *Journal of Theoretical and Applied Information Technology*., 30th September 2023 -- Vol. 101. No. 18-- 2023

[34]. Prabu Sankar, N. Jayaram, R. Irin Sherly, S. Gnanaprakasam, C. Vinston Raja, R. Study of ECG Analysis based Cardiac Disease Prediction using Deep Learning Techniques, *International Journal of Intelligent Systems and Applications in Engineering*, 2023, 11(4), pp. 431–438

[35]. Vinston, R.R., Adithya, V., Hollioake, F.A., Kirran, P.L. Dhanalakshmi, G., Identification of Underwater Species Using Condition-Based Ensemble Supervised Learning Classification, *International Journal of Intelligent Systems and Applications in Engineering*, 2023, 11(3), pp. 1–12

[36]. R. Vinston Raja and K. Ashok Kumar, Fisher Scoring with Condition Based Ensemble Supervised Learning Classification Technique for Prediction in PFZ *Journal of Uncertain Systems* 2022 15:03

[37]. Vinston Raja, R., Ashok Kumar, K. ., & Gokula Krishnan, V. (2023). Condition based Ensemble Deep Learning and Machine Learning Classification Technique for Integrated Potential Fishing Zone Future Forecasting. *International Journal on Recent and Innovation Trends in Computing and Communication*, 11(2), 75–85. https://doi.org/10.17762/ijritcc.v11i2.6131