JAVA Code Analyzer Tool to Destroy Liability and Provide Remedy for Programmers

[1]Dr. Chandramouli H, [2]Prabhuraj M Patil, [3]Madhu R, [4]B Vishwanatha

[1]Professor, [2,3,4]Assistant Professor, Dept. of CSE, East Point College of Engineering & Technology, Bangaluru, India, hcmcool123@gmail.com, pmetipatil@gmail.com, gowda.madhur@gmail.com, vishwanathb807@gmail.com

[*]Project funded by the Government of Karnataka under VGST Scheme (2014-2019)

Abstract – Programming helplessness is a flaw that can be exploited to gain access to the code making the product highly unstable. To make the product safe, liabilities should be recognized and remedied. This tool is used to assess the severity of the condition and the finding remedy for the developer which helps to avoid the liabilities along with solution for the liability.

Keywords - Degree of Liability Tool, Common Weakness Enumeration (CWE), Liability Detection, Liability Remedy, ISM, XML

I. Introduction

The liability has threatened the security of the software at various stages in different stages, involuntarily because of mistakes made by developers or will full infringements. The design and implementation of Software poor are the main causes of most security liabilities [1]. Threats issues can also arise from Web sites and Web applications (webapps). Needs to be protected against all kinds of threats and other active data centers used to host websites and related systems [2]. It is doubtful that the current techniques of safety information will be able to protect critical software systems unless they make security an integral part of the program.

Java language has emerged choose to build systems based on large and complex web, partly because of the safety language in which direct memory access and eliminate problems such as buffer overruns refused advantages. However, in spite of these features, it is possible to make logical programming errors that lead to liabilities such as SQL injection and cross-site scripting attacks [3]. A simple programming error could be left vulnerable Web application for accessing unauthorized data and unauthorized updates, or delete data, and fall leading to denial of service attacks applications [4]. Efforts should be made during the design and implementation of the program to make safe and protect software against it. This document discusses the liabilities that are injected into the Java programs during the coding phase and describes tool developed to detect liabilities and warn the developer for these.

II. PRESENT STATE OF RESEARCH

Because of the extended episodes data robbery, security programmings are increasingly causing connection after consideration of all the normal shortcomings and weaknesses. It has developed a group of helplessness discover the source code and operational framework for programming and administration deficiencies. Various tools are available in the market Java code static analysis. Lists are:

Checkstyle: This plug- in works like the check rules. These rules tells you where you hurt your code similar to compiler, but also produce .class file, it generates alarm. A c injury reported. Check determines which controls were validated against the code and with the severity [4].

FindBugs: It is an open source from University of Maryland [5].

IntelliJ IDEA: Cross-platform Java IE with own set of several hundred code inspections available for analysing code on-the-fly in the editor and bulk analysis of the whole project.

JArchitect: Simplifies managing code by comparing different version of the code. This supports version control [6].

PMD: It is static code analyzer. It uses rule –set that define when a piece of source is erroneous [7]. This software allows checking the type of liability and how much percentage uncertainty present in program. This also provides solution for the type of error occurred.

III. LIABILITIES CHECKED FOR AND SOLUTION

A. LIABILITIES DEFINED FOR THE FLAWS

Following liabilities are defined in this tool.

1. Argument Injection or Modification

2. XML Injection
3. Improper neutralization
4. Information Exposure through Debug Information
5. Password related flaws
    i. Empty Password in Configuration File
    ii. Password in Configuration File
    iii. Unverified Password Change
    iv. Use of Hard-coded Password
6. Unsafe Reflection
7. Use of Obsolete Functions
8. Missing Release of resource after effective Lifetime
9. Relative Path Traversal
10. Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
11. Improper Neutralization of Special Elements used in a Command
12. Improper Output Neutralization for Logs
13. Information Exposure Through Sent Data
14. Cleartext Storage of Sensitive Information in Executable
15. Cross-Site Request Forgery (CSRF)

B. VARIOUS LIABILITIES DESCRIPTION AND SOLUTION

1. Argument Injection or Modification [8]:

Description:

Here program accepts filename as command line argument, if program runs with root privileges; attacker may pass his own string to get the information he need [8].

Solution:

✓ do input validation
✓ use white list of acceptable inputs
✓ input should be decoded

2. XML Injection [9]:

Description:

The product does not legitimately exceptional components that are utilized as a part of XML, permitting assailants alter the substance of the XML before it is handled by an end framework [9].

Solution:

✓ do input validation specific to XML
✓ use white list of acceptable inputs

3. Improper Neutralization [10]:

Description:

The item gets commitment from an upstream portion, in any case it doesn't slaughter or wrongly executes uncommon segments that could be deciphered as flight, meta, or control character progressions when they are sent to a downstream section[10].

Solution:

✓ do input validation
✓ use white list of acceptable inputs
✓ input should be decoded

4. Information Exposure through Debug Information [11]:

Description:

The application contains researching code that can open sensitive information to untrusted parties [11].

Solution:

✓ Do not leave debug statements that are executable via source code
✓ Be careful when interfacing with a compartment outside of the safe area

5. Password related flaws [12]:

i. Empty Password in Configuration File

Description:

Using an empty string as a password is insecure [12].

Solution:

✓ Use longer passwords
✓ Should not use empty string as password

ii. Password in Configuration File [13]:

Description:

The software stores a password in a configuration file that might be accessible to actors who do not know the password [13].

Solution:

✓ during design phase
✓ Avoid storing passwords in easily accessible locations.
✓ store cryptographic hashes of passwords

iii. Unverified Password Change [14]:

Description:

When setting a new password for a user, the product does not require knowledge of the original password, or using another form of authentication [14].

Solution:

✓ When prompting for a password change, force the user to provide the original password in addition to the new password.
✓ if using forgot password then make sure the current user not allowed to change identity.

iv. Use of Hard-coded Password [15]:

Description:

The software contains a hard-coded password, which it uses for its own inbound authentication or for outbound communication to external components [15].

Solution:

✓ For outbound authentication: store passwords outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders.

- ✓ For inbound authentication: Rather than hard-code a default username and password for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password.
- ✓ Perform access control checks and limit which entities can access the feature that requires the hard-coded password.
- ✓ For inbound authentication: apply strong one-way hashes to your passwords and store those hashes in a configuration file or database with appropriate access control.

6. Unsafe Reflection [16]:

Description:

The application uses external input with reflection to select which classes or code to use, but it does not sufficiently prevent the input from selecting improper classes or code [16].

Solution:

- ✓ Refractor your code to abstain from utilizing reflection
- ✓ Do not utilize client controlled inputs to choose and stack classes or code.
- ✓ Apply strict information acceptance by utilizing white lists or circuitous determination to guarantee that the client is just selecting permissible classes or code.

7. Use of Obsolete Function [17]:

Description:

The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained [17].

Solution:

- ✓ Refer to the documentation
- ✓ Consider truly the security ramifications of utilizing an outdated capacity. Consider utilizing interchange capacities.

8. Missing Release of resource after effective Lifetime [18]:

Description:

The product does not discharge an asset after its viable lifetime has finished, i.e., after the asset is no more required [18].

Solution:

- ✓ Utilize a dialect that does not permit this short coming to happen
- ✓ liberating all assets you allot
- ✓ Use asset restricting settings by environment

9. Relative Path Traversal [20]:

Description: The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory[20].

Solution:

- ✓ White list inputs
- ✓ Inputs should be decoded

10. Improper Neutralization of Script Related HTML Tags in a Web Page (Basic XSS) [21]:

Description: The software receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special characters such as "<", ">", and "&" that could be interpreted as web-scripting elements when they are sent to a downstream component that processes web pages [21].

Solution:

- ✓ Check each input parameter
- ✓ Use and specify an output encoding that can be handled by the downstream component

11. Improper Neutralization of Special Elements used in a Command [22]:

Description: The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component [22].

Solution:

- ✓ Run time policy enforcement may be used in a white-list fashion
- ✓ Assign permissions to the software system that prevents the user from accessing/opening privileged files.

12. Improper Output Neutralization for Logs [23]:

Description: The software does not neutralize or incorrectly neutralizes output that is written to logs [23].

Solution:

- ✓ do input validation
- ✓ use white list of acceptable inputs

13. Information Exposure through Sent Data [24]:

Description: The accidental exposure of sensitive information through sent data refers to the transmission of data which are either sensitive in and of itself or useful in the further exploitation of the system through standard data channels [24].

Solution:

- ✓ Specify which data in the software should be regarded as sensitive.
- ✓ Ensure that any possibly sensitive data specified in the requirements is verified

14. Cleartext Storage of Sensitive Information in Executable [25]:

Description: The application stores sensitive information in Cleartext in an executable [25].

Solution:

✓ key management mechanism

✓ updating proprietary data

15. Cross-Site Request Forgery (CSRF) [26]:

Description: The web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request [26].

Solution:

✓ Use a vetted library or framework

✓ Ensure application is free of cross-site scripting issues

✓ Generate a unique nonce for each form

✓ Identify especially dangerous operations

✓ Use the "double-submitted cookie" method

✓ Do not use the GET method

✓ Check the HTTP Referrer header

IV. RESULTS AND DISCUSSIONS

A. DEGREE OF LIABILITY IN A PROGRAM

Each of the weaknesses discussed in this paper has been assigned a severity level defined in CWE. In this paper we define a metric for calculating the Degree of Uncertainty (referred to as ISM).

EQUATION 1: DEGREE OF LIABILITY

$$\boxed{ISM=\sum_{I=1}^{m} Wi*Ni}$$

Where,

ISM - stands for the **Degree of Liability**,

i - is the type of liability where i=1,2,....m

Wi - is the Severity of Liability in the software

Ni - is the frequency of occurrence of liability i.

B. WORKING OF TOOL

The tool takes as input any Java program and scans to identify the liabilities. If any liability is detected then it displays warning message and suggests steps for its mitigation.

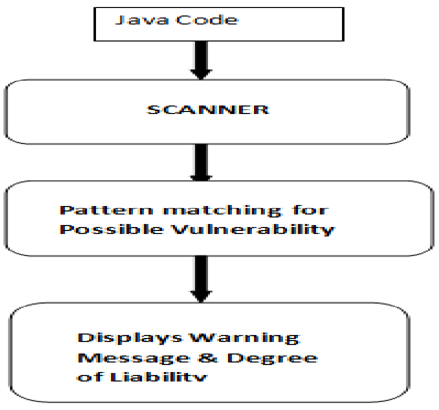
Java Code
SCANNER
Pattern matching for Possible Vulnerability
Displays Warning Message & Degree of Liability

Fig. 1: Working Procedure

The steps followed are:

1. Select the input Java program
2. Select from the drop down list all types of liabilities intended to be detected
3. As shown figure As shown in Figure 1, for a Java program given as an input to the Tool, it displays type of liability found and the place of its occurrence. It also gives the Degree of Severity in the input program.

C. RESULTS AND TABLES

Affects of Availability: This will cause undesired behavior and system crash may happen and it may enter infinite loops.

Table 1: Severity of Liabilities

Type of Liability	Severity
Argument Injection or Modification	21
XML Injection	6
Improper neutralization	11
Information Exposure through Debug Info	3
Empty Password in Configuration File	4
Password in Configuration File	4
Unverified Password Change	2
Use of Hard-coded Password	6
Unsafe Reflection	1
Use of Obsolete Functions	2
Missing Release of resource after effective Lifetime	8
Relative Path Traversal	2
Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	7
Improper Neutralization of Special Elements used in a Command	2
Improper Output Neutralization for Logs	4
Information Exposure Through Sent Data	2
Cleartext Storage of Sensitive Information in Executable	2
Cross-Site Request Forgery (CSRF)	7

V. CONCLUSION

The tool described here detects liabilities that exist in the code, calculates the degree of Severity of the input Java program and gives the remedy for that error. The efficiency of the tool is designed to use for calculating the degree of uncertainty in two categories of programs: one written by experienced Java developers and the other students.

REFERENCES

Books:

[1] G. Mcgraw, Software Security: Building Security In, Addison Wesley, 2006.

[2] A. K. Talukder, M. Chaitanya, Architecting Secure Software Systems, Auerbach Publications, 2009.

Journal Papers and Website Links:

[3] R. Priyadarshini, A. Basu and S. Sushma, “SecCheck: A Tool to Detect Vulnerabilities in Java Code,” International Conference on On-Demand Computing, ICDOC Bangalore, Nov 15-16, 2012.

[4] N. Ghosh and A. Basu, “WebCheck: A Tool to Detect Weaknesses in Java Web Applications,” International Conference on Information and Communication Engineering ICICE Bangalore, June 28-29, 2013.

[5] findbugs.sourceforge.net/findbugs2.html

[6] https://en.wikipedia.org/wiki/IntelliJ_IDEA

[7] https://en.wikipedia.org/wiki/PMD_(Software)

[8] Argument Injection: https://cwe.mitre.org/data/definitions/88.html

[9] XML Injection: https://cwe.mitre.org/data/definitions/91.html

[10] Improper Filtering of Escape, Control, or Meta Successions: http://cwe.mitre.org/data/definitions/150.html

[11] Information Exposure through Debug Information: https://cwe.mitre.org/data/definitions/215.html

[12] Keeping Password Empty in Configuration File: https://cwe.mitre.org/data/definitions/258.html

[13] Config File containing Password: https://cwe.mitre.org/data/definitions/260.html

[14] Untested Password Change: https://cwe.mitre.org/data/definitions/620.html

[15] Use of Hard-coded Password: https://cwe.mitre.org/data/definitions/259.html

[16] Unsafe Reflection: https://cwe.mitre.org/data/definitions/470.html

[17] Use of Obsolete function: https://cwe.mitre.org/data/definitions/477.html

[18] Use of resource after its lifetime: https://cwe.mitre.org/data/definitions/772.html

[19] Improper Input Validation: http://cwe.mitre.org/data/definitions/20.html

[20] Relative Xpath traversal: https://cwe.mitre.org/data/definitions/23.html

[21] Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS):

https://cwe.mitre.org/data/definitions/80.html

[22] Improper Neutralization in command: https://cwe.mitre.org/data/definitions/77.html

[23] Improper Output Neutralization for Logs: https://cwe.mitre.org/data/definitions/117.html

[24] Information Exposure through Sent Data: https://cwe.mitre.org/data/definitions/201.html

[25] Cleartext Storage of Sensitive Information in Executable: https://cwe.mitre.org/data/definitions/318.html

[26] Cross-Site Request Forgery

https://cwe.mitre.org/data/definitions/352.html

[27] Prajna Bhavi, Dr. Chandramouli H, Dr. B.R. Prasad Babu “A tool For Ferrating Out Software Vulnerability in Aegis & Armament Program Redemption” from IJER ,Volume No.5, Issue Special: 4, pp:790-991.