RESEARCH ARTICLE OPEN ACCESS

# Identifying False Positives Result in Security Testing (A Case Study)

Norazrina Abu Haris
*Research & Development, Telco provider, Malaysia*
Email: azrinaharis@gmail.com

----------------------------------------************************----------------------------------

## Abstract:

This paper presents the experimental design to identify false positives result in security testing. False-positive is a test result that indicates the presence of a vulnerability. However, in the security testing actual scenario, no vulnerability exists and the code's functionality is correct. The noise requires remediation work that is not necessary. Usually, vulnerability occurs during security testing. Understanding and identifying false positives can assist software developers during application development. Thus, the code error could be corrected and removed before the actual code execution.

*Keywords* — **false positive, security testing, vulnerabilities.**

----------------------------------------************************----------------------------------

## I. INTRODUCTION

In software testing process, security testing is a vital activity to identify the threats in the application system and measure potential vulnerabilities. Therefore, the threats can be encountered and the system cannot be exploited. Security testing could detect all possible security risks in the application system. One of the main types of security testing is Vulnerability scanning using automated software or Static analysis tool (SAT).

The vulnerabilities detected by SAT include hard-coded Credentials, weak password, Improper Authentication weakness, and Information Exposure weakness. SAT could be examined the code to capture the defects and detect the security threats in the source code. Even though SAT is good at finding security vulnerabilities, but the tool can produce many false positives. This can be a time-consuming process for software developers to fix the error before executed in production. This paper focus to identify false positives result in the source code that seems as security flaws by SAT.

There are also some suggestions to minimize false positive during SAT scanning process. These suggestions can assist the developers to fix the code and save development time and productivity. Thus, SAT scanned testing usually conducted performed regularly to identify any potential vulnerabilities so the application codes could be more effectives and free from any security flaws.

## II. OVERVIEW OF STATIC ANALYSIS TOOL

Nowadays, there are many static analysis tools available in the market. Some of them are open source and others are commercial tool. All tools can use to detect the vulnerabilities. However, these tools have some limitation and the developers requires some knowledges to understand the tools limitations.

The static analysis tools such as open source like Pixy, AppCodeScan and Lapse as in [15] and commercial tools like Synopsis Coverity, Fortify, Burpsuite are used to find out the system vulnerabilities and to determine whether data and resources are protected from potential intruders. This study conducted experimental security testing, using SAT known as Yet Another Static Code Analyzer (Yasca) as in [14]. Yasca is an open source program which looks for security vulnerabilities in a wide range of programming language, including python. Usually, the source code testing using SAT was conducted by testers to confirm that the error would not cause any security issues details and could be deployed into production.

The SAT vulnerability scans usually generated a large number of reactions with different injection techniques such as SQL injections, cookie manipulation, command injection, HTML injection, cross-site scripting and etc. The reactions indicated that the vulnerability exists either true positive, or false positive. Usually, source code verification was conducted by developers. The developers would check if the true positive result exist, vulnerabilities are required to be identified correctly and developer have to correct the codes. However, if the false positives result exists during security testing, the vulnerabilities result identified, in actual fact is a false alarm or false positive that could be ignored. Therefore, the developers required to provide strong proof justification for the end-users that the system is secured.

## III. VULNERABILITY TESTING

This study discusses the empirical evaluation of a static code analysis tool using Yasca. The tool is used to scan the codes written in Python and HTML. The code that was tested is a web application deployed on MariaDB server. A summary of code tested in the case study is shown in Table 1.

TABLE 1.
SUMMARY OF CODE TESTED

| *No* | *Item* | *Description* |
|---|---|---|
| 1. | Number of Files | 654 (Python) |
| 2. | Lines of Codes | 109,151 |
| 3. | Total Vulnerabilities | 184 |
| 4. | Severity | Critical |
| 5. | Category | Weak Credential |

The scanner results findings have detected 184 vulnerabilities. Most of the issues have the similar vulnerabilities and has been identified as critical.

Based on Open Web Application Security Project (OWASP), the security flaws commonly identified as Weak Password that is easy to detect by both humans and computer. The password can be easily cracked because hackers can use a dictionary attack, to hacked by using the password with words found in the common dictionary. Basically, the password error type was detected in Common Weakness Enumeration (CWE)-259 that shows as hardcoded password inbound authentication or for outbound communication to external components as in [12]. However, when the codes were thoroughly checked, most of vulnerabilities are identified as false positives. Several examples on the false positives code identification are explained and shown as follows:

### A. *Example 1*

The example 1 shows the label at placeholder and type has name *password*. SAT has identified the vulnerability as weak credential and hardcoded password. However, the developer identified that the vulnerabilities was caused by the hardcoded text password that refers to the field name and not the password field. This vulnerability is identified as false positive.

```
1. <div class="wrap-input100 validate-input" data-validate =
   "Password is required">

2. <input class="input100" type="password"
   [(ngModel)]="userPass" name="pass"
   placeholder="Password">
```

### B. *Example 2*

The example 2 shows the line for password is tagged as # or indicate as comment. SAT has identified this code has security vulnerability weak credential and hardcoded password. However, the developer identified that the vulnerabilities was caused by the word *password* that has been commented and should be ignored. This vulnerability is identified as false positive.

```
parser = reqparse.RequestParser()
        parser.add_argument('username', required=True)
        #parser.add_argument('password', required=True)
    data = parser.parse_args()
    user_result = None
    username = data['username']
    #password = data['password']
```

### *C. Example 3*

The example 3 shows the password text is refers as the name of URL sites. SAT detect this code has vulnerabilities due to weak password and hardcoded password. However, the developer identified that the vulnerabilities was caused by the hardcoded password that were not contains real password. These vulnerabilities are identified as false positives.

```
        urlpatterns = [
    path('password_change/', wrap(self.password_change,
    cacheable=True), name='password_change'),
        path(
        'password_change/done/',
        wrap(self.password_change_done, cacheable=True),
        name='password_change_done',
        ),
```

### *D. Example 4*

The example 4 shows the code which identified by SAT as vulnerabilities weak password and hardcoded password. However, the developers identified that the vulnerabilities were caused by MyConstants module that is as intermediary to hide the real password. This vulnerability identified as false positive.

```
conn = mariadb.connect(host=MyConstants.mariaDb2Ip,
  user=MyConstants.mariaDb2Username,
password=MyConstants.mariaDb2Password,
  database=MyConstants.mariaDb2Name)
conn.autocommit = True
cur = conn.cursor(dictionary=True)
```

### *E. Example 5*

The example 5 shows the code that SAT detect has vulnerabilities weak password and hardcoded password. However, the developer identified that the vulnerabilities was caused by the hardcoded password in a variable name argument to validate the password. This vulnerability is identified as false positive.

```
def validate_password(password, user=None,
password_validators=None):
```

### *F. Example 6*

Lastly, the example 6 shows the source code has empty field "*if the password is none*" and the field is replaced with asterisk if the password exists. SAT detected this code has vulnerability weak password and hardcoded password. However, the developer identified that the vulnerabilities was caused by the hardcoded text password that does not contain real password. The text password should be a variable name where the value will be replaced with sensitive data in a netloc with "****", if exists. This vulnerability identified as false positive.

```
password = '' if password is None else ':****'
return
'{user}{password}@{netloc}'.format(user=urllib_parse.qu
ote(user),
```

## IV. CONCLUSIONS

This study suggested that the false positive result that existed in the codes during application security testing using SAT automated process could be ignored due to the tools limitations. The false positive result could be minimized in several ways. One of the ways is the developer require to perform manual code review that detect the parts of the application "dead code" or libraries that are not being invoked, and used by the rest of the application. This study believed that by performing code review the codes security flaws can be reduced and codes that has been treated as unused code could be ignored. Another suggestion is the SAT tool could be used to automatically verify security flaws in the codes and presented to the user with a proof that the codes are secured for exploitation.

## REFERENCES

[1] Ivo Gomes, Pedro Morgado, Tiago Gomes and Rodrigo Moreira, "An overview on the Static Code Analysis approach in Software Development", Portugal.

[2] Muhammad Nadeem, Byron J. Williams and Edward B. Ellen, "High False Positive of Security Vulnerabilities: A Case Study", USA, 2012.

[3] Jinqiu Yang, Lin Tan, John Peyton and Kristofer A Duer, "Towards Better Utilizing Static Application Security Testing". USA.

[4] Rahma Mahmood and Qusay H. Mahmoud, "Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code", Canada.

[5] Ulf Mattsson, "A case study - Selecting a code review approach," SSRN-id1308728.

[6] Netsparker Enterprise, "False Positive in Web Application Security", [Online]. Available: https://www.netsparker.com/false-positives-in-application-security-whitepaper/, Austin, 2020.

[7] Netsparker Enterprise, "The Problem of False Positive in Web Application Security and How To Tackle Them", [Online]. Available: https://www.netsparker.com/blog/web-security/false-positives-web-application-security/, Austin, 2020.

[8] Geraint Williams, "Vulnerability scan and false positives: the importance of sanitizing input", UK, 2012.

[9] Dejan Baca1, Kai Petersen, Bengt Carlsson and Lars Lundberg, "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?", Conference Paper, 2009.

[10] Nuno Antunes and Marco Vieira," Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services", Portugal, 2009.

[11] DJango, "Source Code for django.contrib.auth.password_validation", [Online]. Available: https://docs.djangoproject.com/en/2.0/_modules/django/contrib/auth/password_validation/, 2020.

[12] Mitre, "Common Weakness Enumeration", [Online]. Available: https://cwe.mitre.org/index.html/, US, 2020.

[13] Doug Hellman, "argparse- Command line option and argument parsing", [Online]. Available: https://pymotw.com/2/argparse/, 2020.

[14] YASCA [Online]. Available: https://sourceforge.net/projects/yasca/, 2020.

[15] Benitha Joseph, "Static Analysis Tool | Source Code Review Tools", [Online]. Available: https://medium.com/@benithajose/static-analysis-tools-source-code-review-tools-a9dedc872bf2, 2020.