INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Informática

Segurança de Software

MEIC 2017-2018 – 1st Semester

Paulo Gouveia nº 75657

Filipe Soares nº 76543

Pedro Teixeira nº 81899

Grupo nº 8

1. Introduction

Nowadays security is a subject that concerns several entities due to the informatization of workflows, share of information, cloud computation or just storage of data. Security as a discipline explores vulnerabilities at different layers and classifies them according to their execution (Hardware, Software, Network, etc.).

Depending on the layer where the vulnerability is found, there are ways to prevent or defend from possible attacks. For example, there are firewalls to defend from network related attacks or antivirus to defend the host from running programs. Having all these layers defended forces attackers to search for weak points (less secure parts of the system) that often appear in the application level. Although it may exist a firewall on this level, its configuration must be carefully done and they only offer web application protection, meaning that they don't identify errors [13].

On the application level the majority of the security resides with the programer making it a tedious and mistake prone endeavor. Frameworks like the one designed by Huang et al. [16] that tests Web application vulnerabilities, don't guarantee identification of all bugs but more important, don't offer immediate security. Scott and Sharp's [14] [15] solutions guarantee an immediate security, by using a gateway that filters invalid and malicious inputs. All this research and concern show how relevant automated tools that analyze and identify possible vulnerabilities as fast as possible become by the day.

Being the most critical vulnerabilities results of insecure information flows, it is important to defend as fast as possible from such misleading flows, and make them more secure. One of the most common sources of vulnerabilities is the lack of proper validation of the parameters that are passed by the client to web applications. In fact, OWASP's Top Ten Project, which lists the top ten sources of vulnerabilities in web applications, scores unvalidated input as the number one cause of vulnerabilities in web applications [10].

When talking about *Static Analysis* and *Dynamic Analysis* it is important to talk about their goals and modes of operation. In this project we implement a *Static Analysis* tool which the main goal is to evaluate an application by examining its source code without running it. The fact that while examining, it explores all possible executions paths and variables makes it invaluable in security assurance because security attacks seek flaws that wouldn't be easily manifested. On the other hand, *Dynamic Analysis* consist on testing and evaluating the application during runtime. It reveals some defects or vulnerabilities that would be too complex to be detected by static analysis and its main goal is to find errors and debug applications [19].

On this document we investigate vulnerabilities related with input validation in PHP Web Applications and explain how our tool analyzes source code slices and classifies them as being secure or not according to the flow of information in that slice. We also explain how the adopted procedure may suffer from certain imprecisions created at different levels [Image 1].

2. Our Tool

In light of the problem presented we developed a Python program that performs a static analysis over slices of PHP source code, represented as AST (abstract syntax trees), and detects possible vulnerabilities due to missing sanitizing methods. This analysis is tightly related to information flows. The notion of information flow is central to two of the three main security properties: confidentiality and integrity [5].

For a more detailed explanation, from the slice's AST we recursively try to find a path from a known sensitive sink to a possible entry point. If throughout the transversal of the tree a valid sanitization function for the sink is found, the current branch is discarded. This sanitization functions is assumed to always be used correctly which may result in missed vulnerabilities as explained in section 2.1.1.

Finally, we compile an output based on the path returned. If our program does not find a valid sanitization function, the vulnerable path is returned and the program outputs: the name of the vulnerability, the entry point and the sensitive sink. If no path is returned, the program outputs that there is no vulnerability.

Our tool is dependant on a gathered set of vulnerability patterns (for SLQ injection, XSS, and CMD injection) compiled from our research [1, 3, 4, 7, 8, 9, 11, 12] and knowledge of the subject.

2.1. Guarantees

Because we are able to define which patterns are accepted, we can flag an information flow as tainted and its behaviour as illegal, thus making sure specific vulnerabilities are not permitted to run.
If multiple variables are involved in the flow between an entry point and a sink, it is possible to conceive a scenario where some are sanitized but not all. Out tool guarantees that if there is a possible vulnerable clear flow of information, that vulnerability is reported.

2.2. Limitations

Due to the limited amount of time and low requirements, the tool produced is very simplistic with it only performing a static analysis over a pre-made AST and comparing a path with a given pattern. As a result the precision is dependant on our input templates, in addition to what we define as untainted through our patterns. Moreover, being a mere syntactic analysis tool it lacks the advantages given by dynamic analysis as in analysing code while executing, identifying vulnerabilities that might have been overlooked by the static analysis.

2.1.1. Imprecise tracking of information flows

Although we try to detect as many illegal information flows as possible, not all are caught and this can be for several different reasons. One of these reasons is entirely beyond the control of our tool. Our program performs its analysis over ASTs but the process to generate these trees can be flawed. Wrong generation of slices (image 1 - mark 1), slices not having all information, etc…

Another reason is the lack of control over conditional or loop blocks (if, while, for, etc.). For example, our tool marks *slice11.php* as not vulnerable. However the *ifs* are allowing a clearly vulnerable input to run. This is an example of a possible **false negative** and can be exploited through ' OR 1=1 --.

Inputs sanitized through the use of conditional statements are not checked. This can lead to **false positives** that could be avoided through further code implementation for a better refinement of the analysis of the AST.

2.1.2. Imprecise endorsement of input validation

Our tool is highly dependant on the vulnerability patterns we give it and therefore it can overlook vulnerabilities due to missing or incorrect patterns. Also, the valid sanitization functions, either standard (provided by the language) or custom (matching with regular expressions), are decided in an unsound way which assumes they are always used correctly [6]. Custom sanitization methods specifically can be exploited via the use of missing special characters and encoding from the expressions matched against.

2.1.3. Other limitations

In case of there being multiple vulnerabilities within the same slice, only one of them (the first found) will be displayed on the output. However, assuming the tool can be ran more than once, this does not affect the correctness of our tool: after the detected vulnerability is corrected, the tool would check the code again and find another vulnerability.

2.3. Improvements

One way to improve our tool, making it more precise, would be to perform the analysis over the source code as a whole. This would give access to all possible information flows but would be much heavier to compute (hence less efficient in the time it takes to produce results) and would still suffer from the problem of undecidability associated with static analysis [17].

One way to mitigate this problem of undecidability that causes a great number of false positives could be, as suggested by I. Medeiros et al. [18], the use of data mining to compare the flagged path of information flow with known false positives.

Yet another improvement, that would make the tool more accurate, could be the implementation of dynamic analysis alongside the static analysis performed. This would require us to have a more intrinsic knowledge of the host where the application is running on so that we could write the proper code to detect vulnerabilities at runtime. However, such approach would require a greater amount of time and work to develop as well as an implication of a slower runtime for the application.

3. Conclusion

All Web applications inherit the same problem and theirs vulnerabilities are independent of the technology used. Despite the effort to defend possible entry points and to make the hacker's life more difficult, new tools and techniques appear almost every month and because of this it is very important to keep up with this evolution and try to always be one step forward. On the end of the day, the security of web applications ultimately rests in the hands of the programmers [6].

About our tool, it is only as precise as we make it to be through our definition of what information flow is tainted and what was sanitized. As explained previously, due to the fact that we are working with slices of the source code, the fact of being static analysis and the matching just with our expressions leads to misleading results that can be false positives, false negatives or both.

4. References

[1] http://php.net/manual/en/ref.strings.php

[2] Symantec: Internet threat report. 2012 trends, volume 18 (Apr 2013).

[3] https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet

[4] https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet

[5] Sandhu, R.S.: Lattice-based access control models. IEEE Computer 26(11), 9–19 (1993)

[6] https://www.cs.ucsb.edu/~chris/research/doc/oaklando8_saner.pdf

[7] http://php.net/manual/en/function.odbc-exec.php

[8] http://php.net/manual/en/security.database.sql-injection.php

[9] http://form.guide/php-form/php-form-action-self.html

[10] OWASP. Top ten project.http://www.owasp.org/ ,May 2007.

[11] https://stackoverflow.com/questions/129677/whats-the-best-method-for-sanitizing-user-input-with-php

[12] https://stackoverflow.com/questions/8470748/a-better-sql-string-sanitization-function

[13] Bobbitt, M. "Bulletproof Web Security." Network Security Magazine, TechTarget Storage Media, May 2002. http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml

[14] Scott, D., Sharp, R. "Abstracting Application-Level Web Security." In: Proc. 11th Int'l Conf. World Wide Web (WWW2002), pages 396- 407, Honolulu, Hawaii, May 17-22, 2002.

[15] Scott, D., Sharp, R. "Developing Secure Web Applications." IEEE Internet Computing, 6(6), 38-45, Nov 2002.

[16] Bobbitt, M. "Bulletproof Web Security." Network Security Magazine, TechTarget Storage Media, May 2002. http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml

[17] Landi, W.: Undecidability of static analysis, 1992. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.9722&rep=rep1&type=pdf

[18] Medeiros, I., Neves, N., Correia, M., Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.432.7100&rep=rep1&type=pdf

[19] https://software.intel.com/en-us/inspector-user-guide-windows-dynamic-analysis-vs-static-analysis

5. Appendix

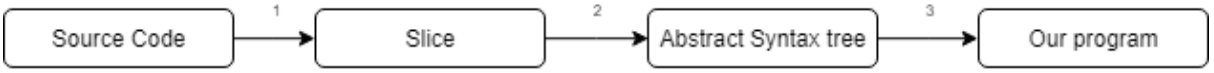

Image 1 - Procedure Flow.

```php
<?php
$raw = $_POST['matapelajaran'];
$clean = $_POST['idmatapelajaran'];
$raw = mysql_real_escape_string($raw);
$clean = mysql_real_escape_string($clean);
$query = "UPDATE matapelajaran SET matapelajaran='$raw' WHERE id_matapelajaran='$clean'";
mysql_query($query,$koneksi);
?>
```

```
-> analysing 'slices/slice-0.json'
Vulnerability: None
```

Image 2 - All entry points are sanitized, therefore there is no vulnerability.

```php
<?php
$raw = $_POST['matapelajaran'];
$clean = $_POST['idmatapelajaran'];
$clean = mysql_real_escape_string($clean);
$query = "UPDATE matapelajaran SET matapelajaran='$raw' WHERE id_matapelajaran='$clean'";
mysql_query($query,$koneksi);
?>
```

```
-> analysing 'slices/slice-1.json'
Vulnerability:  SQL injection (MySQL)
Entry point:    _POST['matapelajaran']
Sensitive Sink: mysql_query
```

Image 3 - The $raw variable is used without being sanitized which creates the vulnerable path: *_POST['matapelajaran'], raw, query, mysql_query.*

```php
<?php
$nis=$_POST['nis'];
while ($indarg == "") {
        $query="SELECT *FROM siswa WHERE nis='$arg3'";
        $arg3 = $arg2;
        $arg2 = $arg1;
        $arg1 = $nis;
        $indarg = substr($indarg,1);
}
$q=mysql_query($query,$koneksi);
?>
```

```
-> analysing 'slices/slice10.json'
Vulnerability:  SQL injection (MySQL)
Entry point:    _POST['nis']
Sensitive Sink: mysql_query
```

Image 4 - For this slice a chain of assignments is found and a vulnerability detected though the path: *_POST['nis'], nis, arg1, arg2, arg3, query, mysql_query.*

```php
<?php
$arg=$_POST['nis'];
while ($arg != "") {
    $first = substr($arg,0,1);

    if ($first=="'") {
    $indarg = $indarg . "'";
    } elseif ($first==" ") {
    $indarg = $indarg . " ";
    } elseif ($first=="0") {
    $indarg = $indarg . "0";
    } elseif ($first=="R") {
    $indarg = $indarg . "R";
    } elseif ($first=="1") {
    $indarg = $indarg . "1";
    } elseif ($first=="=") {
    $indarg = $indarg . "=";
    } elseif ($first=="-") {
    $indarg = $indarg . "-";
    }
    $arg = substr($arg,1);
}
$query="SELECT *FROM siswa WHERE nis='$indarg'";
$q=mysql_query($query,$koneksi);
?>
```

```
-> analysing 'slices/slice11.json'
Vulnerability: None
```

Image 5 - Example of a false negative where a possible bad input is allowed to reach the sensitive sink.