Instituto Superior Técnico - Alameda

Software Security Project Report

Static Code Analysis Tool

Discovering vulnerabilities in PHP Web Applications

Group 10

Rui Ventura 81045

Diogo Castilho 78233

November 20, 2017

Experimental part

Our analysis tool was conceived using the `Python` programming language, version `3.6.3`. It consists of a main component, the analyser (`analyser.py`), containing the logic for traversing some elements of an AST, and a pattern module (`pattern.py`) that houses the `Pattern` class, used to instantiate objects that represent vulnerable patterns which can be found in the `patterns` file, another component of the tool.

Components

Analyser

The main analyser component is run by invoking it and passing it a PHP program slice in JSON object format as a command line argument.

```
$ analyser.py /path/to/slice.json
```

Listing 1: Command line to run analyser

The slice being provided must already be in the form of an AST just like the ones produced by Glayzzle's PHP Parser [1]. Otherwise, the tool will not work as it's designed to only accept ASTs with that specific format.

Patterns

The `patterns` file contains a set of vulnerable patterns with the following format:

```
Vulnerability
Entry_1,Entry_2,...,Entry_i
Sanitizer_1,Sanitizer_2,...,Sanitizer_j
Sink_1,Sink_2,...,Sink_k
```

Listing 2: Vulnerable pattern template

where `Vulnerability` is the name of the vulnerability, `Entry` is an entry point, `Sanitizer` is a sanitization/validation function, and `Sink` a sensitive sink. Example:

```
SQL injection (PostgreSQL)
$_GET,$_POST,$_COOKIE,$_REQUEST
pg_escape_string,pg_escape_bytea
pg_query,pg_send_query
```

Listing 3: SQL Injection pattern, specific to PostgreSQL

Method

To start off, the patterns are fetched from the `patterns` file and parsed, generating a list of `Pattern` objects.

Then, the JSON formatted slice is loaded and the AST is converted into a Python dictionary, which is used throughout the analysis. Adopting a visitor-like pattern, the tool's able to analyse the nodes separately, which allows for some modifiability, yet not much, in the sense that, for another construct to be introduced, one or two function need(s) to be added to analyse the corresponding node.

During the traversal, a list of tainted symbols (variables) is carried along, as well as a dictionary of defined variables and their respective values (even if previously undefined). This allows us to perform, as we go, some basic taint analysis as we can detect when a tainted object is used in a sensitive sink.

Once such a case is detected, the program reports the vulnerability, suggests a set of possible sanitizations that can be used and, for simplicity's sake, exits, since we assumed not more than one vulnerability was present in the given slices of code.

Examples

Slice 6

```
echo $_POST['username'];
```

```
$ ./analyzer.py /path/to/slice6.json
Possible vulnerability detected: XSS
Please consider sanitizing tainted code with 1 of the following:
- htmlentities
- htmlspecialchars
...
```

Listing 4: Shortened example XSS vulnerable slice analysis output

Slice 8

```
$nis=$_POST['nis'];
if($indarg=="") {$query="SELECT * FROM siswa WHERE nis='$nis'";}
else {$query="SELECT * FROM siswa WHERE nis='$indarg'";}
$q=mysql_query($query,$koneksi);
```

```
$ ./analyzer.py /path/to/slice8.json
Possible vulnerability detected: SQL injection...
Please consider sanitizing tainted code with 1 of the following:
- mysql_escape_string
- mysql_real_escape_string
```

Listing 5: Shortened example SQLI vulnerable slice analysis output

Discussion

Tool Configurability and Range

Despite being able to analyse the slices given to us as an example to base the tool around, it doesn't go far beyond it. The amount of language elements and scenarios considered is pretty confined to the ones present in the example slices.

In spite of these restrictions, adding language elements isn't as difficult as it seems, though, likewise, not as easy as it could have been made to be. Since the analysis is broken down into various functions, each attributed to a language element (i.e. a node in the AST), meaning, if a language element is to be added, implementing the functions for the nodes being introduced should be sufficient. With it, taint analysis is also a requirement, meaning the tracking of tainted objects/symbols must be implemented along with the basic visiting model.

On the up side, the addition of patterns, if need be, is imensely easy, since all that is needed to do is add it to the `patterns` file in the format seen in Listing 2 and the analyser will also consider it accordingly. The current set of patterns was in part retrieved from [3].

Imprecisions and Assumptions

We attempted to implement a very simple taint analysis mechanism loosely based on [4], with an equally simplistic data mining model, which consists in collecting data on the symbols (variables) that were visited. Code correction was skipped altogether, finally only providing some feedback to the programmer, albeit not too specific.

Our taint analysis doesn't build any more trees, only working with the existing one and some utilitary structures that serve as records or state keepers/trackers, like the list of tainted symbols and their data.

As such, it is very likely that our tool is imprecise and 'vulnerable' to false positives, such as, e.g., tainting a symbol, passing it through a function that manipulates it and outputting it. The following example is a false positive our tool detects:

```
$nis=$_POST['user']
$out=str_replace('<script','',$nis);
echo $out;
```

Listing 6: Example of custom sanitization that triggers XSS false positive

Our tool assumes `$nis` is tainted, since `$_POST` is an entry point. Consequently, `$out` gets tainted because a transformation of `$nis` is assigned to `$out`. However, part of the `script` tags that might have been present have been removed, "sanitizing" the output.

This idea, of "custom sanitization", is exposed in [2], detailing programmer can write "custom" code in order to sanitize code. It is still said that there is no guarantee that, through this mean, the output is safe for usage in a sensitive sink, but assuming that all other custom sanitization operations are doomed from the start isn't absolutely true.

Conclusion

Static code analysis is far from a perfect method of detecting vulnerabilities in non-trivial high-level programming languages due to a vast array of nuances and non-obvious yet sanitizing mechanisms implemented in a custom way by the programmers themselves. Yet, these mechanisms might not assure sanitization either since they might be able to be subverted, meaning the code was vulnerable in the end. Undeniably, howver, it has become more and more robust through the introduction of mechanisms such as taint analysis, data mining and code correction [4].

As far as our tool goes, it is far from achieving the quality of state of the art tools such as WAP [3,4], being our tool limited to a very simple and very small subset of an immense programming language with so many subtleties such as the PHP programming language.

References

[1] Glayzzle and Various Contributors. PHP Parser. Available at `https://github.com/glayzzle/php-parser`, 2017. NodeJS PHP Parser - extract AST or tokens (PHP5 and PHP7).

[2] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, 2008.

[3] Ibéria Medeiros. WAP - Web Application Protection. `http://awap.sourceforge.net/support.html#sanitization`, 2017. Sensitive Sinks and Sanitization Functions by Vulnerability.

[4] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 63–74, New York, NY, USA, 2014. ACM.

Listings