Discovering vulnerabilities in PHP web applications

Francisco Abrunhosa (95580)[1], João Pereira (95600)[1], and Leonor Barreiros (95618)[1]

[1]Group 17

1 Introduction

In this day and age, code injection vulnerabilities are one of the most critical security risks for web applications. As a matter of fact, OWASP classifies it as the third top "Web Application Security Risks". All applications that use some sort of input supplied by the user are susceptible to vulnerabilities. These vulnerabilities exist because user input is considered to be "low integrity" information - sources -, since we cannot control/predict its content. Furthermore, if the programmer doesn't make an effort to validate/sanitize such inputs, then their interaction with other "high integrity", sensitive variables/functions - sinks - can generate potentially dangerous and illegal flows that will ultimately lead to information leaks or compromise the behaviour of the system. The aim of this project is to develop a static analysis tool that can detect such vulnerabilities for the PHP programming language.

2 Usage

Our tool is a PHP language analyser written in Python. To use it, one must give as input a previously generated AST of a slice of a PHP program (a small portion of a program that will be analysed) and a vulnerability pattern that specifies what vulnerabilities to consider. The tool then traverses the AST and outputs possible illegal information flows that might be encoded. It is, therefore, a static analysis tool that searches for possible vulnerabilities in the form of **illegal information flows**.

3 Implementation

3.1 Data Structures

The AST is represented by a list of classes that are its nodes. Each relevant node for the analysis is represented by a class created for that effect. Some groups of language constructs are equivalent from a static analysis perspective (all binary operations, for example) and therefore all belong to the same class. Moreover, any expression that may be tainted (e.g., variable, function call, binary operation) inherits from the super-class EXPRESSION, which will contain its annotations.

3.2 Security Classes

We consider **principal-based security classes**, in which the principals are the set of sources that may have tampered with the value of an EXPRESSION. We considered two types of sources: unsanitized (corresponding to unsanitized flows) and sanitized (sanitized flows). To update the security class of each EXPRESSION at each program point, we defined a policy which we now present.

Definition 1 (Vulnerability policy for integrity)**.** *Let S be a set of sources (sanitized or not):*

- *Set of security levels: all subsets of S, including {} and S*
- *Can-flow relation: high integrity classes can flow to low integrity classes*
- *Least upper bound operator:* $S1 \vee S2 = S1 \cup S2$
- *Greatest lower bound operator:* $S1 \wedge S2 = S1 \cap S2$
- *Bottom element: {}*
- *Top element: S*

3.3 AST Traversal

The AST is traversed once for each vulnerability pattern, with a fresh SYMBOL_TABLE (to store the annotated variables in the current context), POLICY (the one previously defined for that vulnerability), and IMPLICIT_CHECKER (to store implicit sources of taintedness).

Algorithm 1: AST Traversal, performed by the function CREATE_NODES

```
foreach TYPE OF NODE do
    HEAD_NODE = HEAD(AST);
    if TYPE OF NODE == TYPE(HEAD_NODE) then
        annotate EXPRESSIONs in HEAD_NODE with the explicit flows defined by POLICY;
        update SYMBOL_TABLE with the annotated variables in HEAD_NODE;
        if we're checking implicit flows then
            annotate EXPRESSIONs in HEAD_NODE with the implicit flows defined by POLICY;
            update IMPLICIT_CHECKER with the annotated variables in HEAD_NODE;
        end
        AST -= HEAD_NODE;
        CREATE_NODES(AST,SYMBOL_TABLE,POLICY,IMPLICIT_CHECKER);
    end
end
```

3.4 Basic Vulnerabilities

When traversing an AST, taintedness is propagated from sources to sinks using EXPRESSIONs like variables and function calls. A SYMBOL_TABLE keeps track of variables: when one is encountered, it is checked to see if it has been previously instantiated or if it needs to be added to the SYMBOL_TABLE. Basic vulnerabilities, or **explicit leaks, are detected when tainted information reaches a sensitive sink**, which can happen by assignments or function calls. If a left value is a sensitive sink, a vulnerability will be outputted for each taint source from the right value of the assignment. If a function call is a sensitive sink, a vulnerability will be outputted for every taint source of each of its arguments. Tainted information passing through a sanitizer is considered to be sanitized, and if it reaches a sensitive sink, a vulnerability will be outputted, specifying that it does not contain unsanitized flows and specifying the flow of sanitization functions it passed through.

3.5 Implicit Vulnerabilities

Our tool can detect implicit vulnerabilities by tracking assignments and function calls that depend on a tainted condition. We use a stack to keep track of sources, sanitizers, and sanitized sources in conditions (possibly nested). When we evaluate a condition we push all the sources, sanitizers, and sanitized sources of the expression onto a stack and pop them when leaving the context that depends on it. When we assess the existence of implicit flows, we propagate tainted information (with support of the policy) and check for basic vulnerabilities (explicit flows) as well as the existence of sources/sanitizers/sanitized sources in the implicit stack. If any are found, it means we are currently in a tainted condition and an implicit vulnerability will be outputted for each implicit source/sanitized source.

4 Test and Evaluate

To test our tool's correctness and robustness we used the tests that were provided as well as tests that we created. The additional tests were created to evaluate functionalities not covered by the provided ones. We created tests that allowed us to cover both the mandatory and bonus constructs: one test that contains an assign operation statement (e.g. '+='), one that contains both bitwise and boolean 'not' operators (' ' and '!', respectively), one that contains for loops, one that contains elseif statements and one that contains a switch case statement. The rationale behind each test was to ensure that the detection of vulnerabilities, for each evaluated property, was correct and robust.

5 Critical Analysis

5.1 Identification of imprecisions

Question	Answer
1. Are all illegal information flows captured by the adopted technique?	Yes (...)
2. Are there flows that are unduly reported?	Yes
3. Are there sanitization functions that could be ill-used and do not properly sanitize the input?	Yes
4. Are all possible sanitization procedures detected by the tool?	No (...)

Table 1. Identification of imprecisions

5.1.1 Are all illegal information flows captured by the adopted technique?

As far as the scope of our analysis goes, we found no evidence of possible flows that would not be reported (w.r.t. Deterministic Input-Output Noninterference). This is because our "*Rules of thumb*" (in theory) constitute a sound mechanism. However, they only do so for the implemented constructs (which don't fully represent the PHP language) - so, we classify our tool as **soundy**, as described by [1]. It is possible, because software inherently has bugs, that there are flaws in our implementation. Furthermore, if we consider the possibility of nontermination, the following vulnerability would be undetected:

```
1 while ($source == true) { skip; }
2 $sink = false;
```

Listing 1. Unreported illegal information flow

In this program, the attacker would be able to tamper with the value of $SINK, as it would only be set to false if the WHILE loop terminated (which cannot happen if the attacker is able to set $SOURCE to true).

5.1.2 Are there flows that are unduly reported?

In the following code snippet, we see an example of a unduly reported flow: line 5 is unreachable by our code, so this program isn't vulnerable. However, because our tool still evaluates unreachable instructions, it will still signal it as a vulnerability.

```
1 while (true) {
2   if (true) {
3     break;
4   }
5   $sink = $source;
6 }
```

Listing 2. Unduly reported flow

5.1.3 Are there sanitization functions that could be ill-used and do not properly sanitize the input?

In the following code snippet, the sanitization function STRIP_TAGS is used to strip all HTML tags except the <*script*> tag. This means that input containing a <*script*> tag will not be sanitized. So, the attacker will be able to inject code in the HTML of the application, which will possibly generate illegal information flows.

```
1 $a = strip_tags($source, '<script>');
2 echo $a;
```

Listing 3. Bad use of sanitization function

5.1.4 Are all possible sanitization procedures detected by the tool?

Like in 5.1.1, we have to be aware that the scope of our analysis and what our tool covers are limited. With that said, we claim that we detect all possible sanitization procedures as long as the sanitization functions are given in the vulnerability pattern. However, not all sanitization procedures are functions. For example, if we do a case for each allowed value of a variable or parse a variable using an appropriate regular expression (like we did in **Lab 1 of LBS**), then our tool won't be able to recognize that.

5.2 Understanding of imprecisions

5.2.1 Can the identified false negatives lead to exploits?

The false negatives we presented in 5.1.1 and 5.1.3 could lead to exploits. In 5.1.1, if the attacker was able to give as input something that would be evaluated to TRUE, for instance "EVAL("TRUE")", then they would be able to perform a **denial of service** attack. In 5.1.3, if the attacker injected "<script type="text/javascript" src="http://malicious.com/malicious.js">", they would be able to run that script (and we don't know what it could do) once $A was evaluated to perform the ECHO. So, they could do a **cross-site scripting** attack. It is important, however, to notice that in this case, our tool would still flag this slice as being vulnerable. So, it would still be useful in the sense that it could lead the programmer to better analyse the code and possibly change it.

5.2.2 Can the identified false positives be improved?

There is always room for improvement, bearing in mind that, with a static analysis tool and the state-of-the-art of this subject, we would never be able to mitigate the existence of false positives. There are several techniques which we could employ, such as clearing implicit flows when the branches of an if have the same semantics or discarding statements that follow a break statement. In the case of the latter, this technique would make the snippet presented in 5.1.2 no longer report a false positive.

5.3 Improving precision

Bearing in mind that we would never be able to achieve precision (as we're limited by **Rice's Theorem**, introduced in [2]), we had some ideas (besides the basic ones we've mentioned) that would improve the precision of our approach:

- Introduce the analysis of **function definitions** (including the RETURN construct), which would allow us to capture information flows inside the body of functions. This could enable us, for instance, to detect more vulnerabilities (i.e. get closer to being sound), or to assess whether programmer-developed sanitizers are really sanitizing their input
- Enlarging our sample of program slices and respective vulnerabilities and apply a **Recurrent Neural Network** (or variations), which is able to capture dependencies between instructions. We would apply this model after our tool, as a way to filter false positives
- Augment the AST with a **control flow graph**, as it would allow us to discard vulnerabilities in unreachable instructions (for example). That would reduce the number of false positives in our approach (making it closer to being complete)

6 Related Work

The discovery of vulnerabilities in PHP applications has been a subject of research for many years. Whereas more recent approaches (which are closer to being precise) are using different techniques to solve the same problem for the considered language, older work addressed it using similar techniques (or the same).

On the one hand, academia has been able to construct more robust tools for finding vulnerabilities by using new data structures and deep-learning architectures. For example, [3]'s approach is similar to ours since it also used the AST to capture information flows, however, it's different as it augments it with other structures: a **control flow graph** which captures "*the order in which statements can be executed and the values of predicates that result in a flow*", and a **program depence graph** which "*exposes dependencies between statements and predicates*". By using more information, it has less false negatives (it's closer to being sound) and false positives (it's closer to being complete). Furthermore, [4]'s and [5]'s are similar to ours as they use similar structures to represent a program: the former using only the AST and following "*the data paths inside a program to detect security problems*", and the latter using a CFG, which is a related way to represent the data. Nevertheless, they introduce a novelty when it comes to vulnerability detection, since they use **data mining** to do so: either to predict whether each example is a false positive, or to predict if there's a vulnerability and if so, its type (by using an architecture "*which can learn long term dependencies between the tokens*").

On the other hand, previous research had more resemblance with our solution. [6] defined the security class of a variable the same way as we did, as "*a property of its state, and therefore varies at different points or call sites in a program*" and also considered that "*variables should assume the labels of values they currently store*". It's different, since it makes an "***unsound assumption** that as long as an untrusted value passes a certain kind of validation, it is actually safe*" - while our approach still signals a sanitized flow as vulnerable. [7] also identifies "*paths between the location where an input enters the application and a location where this input is used*", but using a different structure: the **dependence graph** that contains "*a list of all variables (...) that might directly influence (or reach) the current program point*". Finally, [8] uses a technique unsuitable for our case, as it uses **context free grammars** to express the possible values of a variables and **derivability** to assess its safeness. Although we both consider that the "*grammar reflects dataflow*", its use of the grammar is different and implies we have a way of knowing the possible and safe values of a variable (which, in our work, we don't).

7 Conclusion

With our implementation of the tool we achieved a notable goal. In particular, with the way it performs static analysis, all programs that employ the constructs we considered in the tests and that encode vulnerabilities in the form of an illegal taint flow (whether it be explicit or implicit) are flagged as such and said flow is duly reported and outputted. Our tool is thus soundy[1], given that it does not produce false negatives for a subset of the language. However, it is not complete, meaning that it might output false positives, this is, programs that do indeed not encode any vulnerability being flagged as vulnerable, but this is a limitation of static analysis. Improving our tool would involve improving its soundness, by enlarging the subset of the language it covers. To do so, we discussed adding another program construct (the function definition) which very commonly introduces vulnerabilities. It would also involve improving its completeness, by making it more permissive. For that end, we would attempt using deep-learning or other structures to represent programs.

References

1. Livshits, B. *et al.* In defense of soundiness: A manifesto. *Communications of the ACM* **58**, 44–46 (2015).

2. Kozen, D. C. Rice's theorem. In *Automata and Computability*, 245–248 (Springer, 1997).

3. Backes, M., Rieck, K., Skoruppa, M., Stock, B. & Yamaguchi, F. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, 334–349 (IEEE, 2017).

4. Medeiros, I., Neves, N. F. & Correia, M. 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*, 63–74 (2014).

5. Rabheru, R., Hanif, H. & Maffeis, S. Deeptective: Detection of php vulnerabilities using hybrid graph neural networks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 1687–1690 (2021).

6. Huang, Y.-W. *et al.* Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, 40–52 (2004).

7. Balzarotti, D. *et al.* Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 387–401 (IEEE, 2008).

8. Wassermann, G. & Su, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 32–41 (2007).