Software Security

Aims

To achieve an in-depth understanding of a security problem. To carry out a hands-on approach to the problem, by implementing a tool for tackling it. To analyse its underlying security mechanism according to the guarantees that it offers, and to its intrinsic limitations. To understand how the proposed solution relates to the state of the art of research on the security problem. To develop collaboration skills.

Format

The Project is presented in subsection "Discovering vulnerabilities in PHP web applications" as a problem, and its solution should have the following two components:

1. An experimental part, consisting in the development of a tool, that is to be presented and discussed.
2. A report, that describes briefly the experimental part (maximum 2 pages), and discusses the guarantees provided by the tool, as well as its limitations, in light of the state of the art for the proposed problem. The first part should present the design of the tool, the main design options, and the output of the tool for a few examples. The document should have no more than 4 pages, excluding references and appendices.

Submissions

Projects are to be solved in groups of 2 or 3 students that should register via Fenix by 9 November. The reports should be submitted also via Fenix, as a pdf. **Please submit a zip file containing the code and the report. The submission deadline is 20 November 23:59.**

Evaluation

The experimental part will be evaluated based on a demo and discussion that will take place during the lab classes of the last weeks.

The report will be evaluated according to the following criteria:

- Quality of writing - structure of the report, clarity of the ideas, language
- Content - relevance and value of the ideas that are conveyed
- Depth - understanding of the state of the art, connection with experimental work
- Originality - detachment from words used in cited papers, references that are cited beyond the suggested ones, own ideas

All sources should be adequately cited. Plagiarism will be punished according to the rules of the School.

Problem

A large class of vulnerabilities in web applications originates in programs that enable user input information to affect the values of certain parameters of security sensitive functions. In other words, these programs encode an illegal information flow, in the sense that low integrity -- tainted -- information (user input) may interfere with high integrity parameters of sensitive functions (so called sensitive sinks). This means that users are given the power to alter the behavior of sensitive functions, and in the worst case may be able to induce the program to perform security violations.

Often, such illegal information flows are desirable, as for instance it is useful to be able to use the inputted user name for building SQL queries, so we do not want to reject them entirely. It is thus necessary to differentiate illegal flows that can be exploited, where a vulnerability exists, from those that are inoffensive and can be deemed secure, or endorsed, where there is no vulnerability. One approach is to only accept programs that properly validate the user input, and by so restricting the power of the user to acceptable limits, in effect neutralizing the potential vulnerability.

The aim of this project is to study how vulnerabilities in PHP code can be detected statically by means of taint and input validation analysis.

Experimental part

This part consists in the development of a static analysis tool for identifying data flow integrity violations which are not subjected to proper input validation. Static analysis is a general term for techniques that verify the behavior of applications by inspecting their code (typically their source code). Static analysis tools are complex, so the purpose is not to implement a complete tool. Instead, the objective is to implement a tool that analyses PHP program slices.

A program slice is the sequence of all instructions that may impact a data flow between a certain entry point and a sensitive sink. An example slice is:

```
11: $u = $_GET['username'];
23: $q = "SELECT pass FROM users WHERE user='".$u."'";
24: $query = mysql_query($q);
```

Inspecting this slice it is clear that the application has an SQL injection vulnerability. An attacker can inject a malicious username like ’ OR 1 = 1 -- , modifying the structure of the query and obtaining all users’ passwords.

The tool essentially has to search for certain vulnerable patterns in the slices. All patterns have 4 elements:

- name of vulnerability (e.g., SQL injection)
- a set of entry points (e.g., $_GET, $_POST),
- a set of sanitization/validation functions (e.g., mysql_real_escape_string),
- and a set of sensitive sinks (e.g., mysql_query).

If the data flow passes through a sanitization/validation function, there is no vulnerability; if it does not pass through a sanitization/validation function, there is a vulnerability.

The patterns are loaded from a file with a simple format. For example: the file is a sequence of patterns; each pattern is represented by 4 lines, one per element; each element of a set (entry point, sanitization/validation function, sensitive sink) is separated by a comma. An example file with two patterns is the following:

```
SQL injection
$_GET,$_POST,$_COOKIE
mysql_escape_string,mysql_real_escape_string,mysql_real_escape_string
mysql_query,mysql_unbuffered_query,mysql_db_query


SQL injection
$_GET,$_POST,$_COOKIE
pg_escape_string,pg_escape_bytea
pg_query,pg_send_query
```

You should create all patterns necessary to analyse correctly all the example slices we provide you (see the zip file attached to this page). A list of entry points, sanitization functions, and sensitive sinks can be found in the table in this page: http://awap.sourceforge.net/support.html

The tool should be called in the command line. You can assume that the parsing of the PHP slices has been done, and the input file contains a representation of the slice in the form of an Abstract Syntax Tree. The output should be a statement if the slice is vulnerable or not and an indication of which instruction(s) sanitizes the data in case there is no vulnerability.

The tool has to be tested with all the slices provided, but be generic and configurable for other slices and vulnerabilities that can be expressed with patterns of the format above.

Discussion

Consider the security mechanism that is studied in this project, and that consists of:

1. Statically retrieving the potentially vulnerable program slices by means of a taint analysis, as described in I. Medeiros et al.
2. Discarding the slices where input is considered to have been validated, as defined and implemented in the experimental part.

Given the intrinsic limitations of the static analysis problem, the above mechanism is necessarily imprecise. It can be unsound (produce false negatives), incomplete (produce false positives) or both. Define and discuss what the resulting mechanism is able to achieve, while answering the following questions:

1. Explain what are the imprecisions that are built in to the proposed mechanism. Have in mind that they can originate at different levels, for instance:
    a) imprecise tracking of information flows -- Are all illegal information flows captured by the adopted slicing technique? Are there flows that are unduly reported?
    b) imprecise endorsement of input validation -- Are there sanitization functions that could be ill-used and not properly validate the input? Are all possible validation procedures detected by the tool?

2. For each of the identified imprecisions that lead to:
    a) undetected vulnerabilities (false negatives) -- Can these vulnerabilities be exploited? If yes, how (give concrete examples)?
    b) reporting non-vulnerabilities (false positives) -- Can you think of how they could be avoided?

3. Propose one way of making the tool more precise, and predict what would be the tradeoffs (efficiency, precision) involved in this change.

The discussion should be supported by references to related work, including the following articles:

- Y. Huang et al. "Securing web application code by static analysis and runtime protection", ICWWW 2004.
- G. Wassermann and Z. Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities", PLDI 2007.
- D. Balzarotti et. al. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications", S&P 2008.
- I. Medeiros et al. “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, April 2014.

Reading Slices

Syntax of slices

Your program should read from a text file the representation of a PHP slice in the form of an Abstract Syntax Tree (AST). The AST is represented in JSON, using this syntax.

Parsing slices

In order to parse the ASTs, you can use an off-the-shelf parser for JSON in your language of choice. Note that not all of the information that is available in the AST needs necessarily to be used and stored by your program.

Calling your program

Your program should take an argument that consists of the name of a file containing the text input. The way to call it depends on the language in which you choose to implement it (you can pick any you like), for instance:

```
./analyser slice.json
python ./analyser.py slice.json
java analyser slice.json
```

Producing slices

Besides the slices that are made available, you can produce your own ASTs for testing your program by using this demo.

If you wish to build a stand-alone tool, you can use this PHP parser, which produces ASTs as required for this project. This is however not a requirement of the project.

Slice Expressivity

Slices can be seen as the reduction of a program to the set of instructions that are relevant to the analysis. There can be however different levels of refinement when deciding what is "relevant". In other words, the slice can be an *approximation* of what could influence the analysis. We can opt for a coarser approximation that simplifies the analysis, and may lead to wrong results. Or, we can opt for a finer approximation that leads to more precise results. In this project you may assume two different levels of refinement for the slices that are used as input.

1. Basic analysis
   In this case, you can assume that slices do not contain condition statements or cycles.
   For instance, the code

```
$u = $_GET['passwd'];
$v = 0;
if ($u=="123") {$q = True;} else {$q = False;};
```

   would render the slices

```
$u = $_GET['passwd'];
$q = True;
```

   and:

```
$u = $_GET['passwd'];
$q = False;
```

2. Advanced analysis

   In this case, slices preserve the structure of condition statements and cycles. For instance, the code

```
$u = $_GET[‘passwd’];
$v = 0;
if ($u=="123") {$q = True;} else {$q = False;};
```

   would render the slice

```
$u = $_GET[‘passwd’];
if ($u=="123") {$q = True;} else {$q = False;};
```

Grading components

Experimental part (75% of the project grade)

Grading of the experimental part will reflect the level of complexity of the developed tool. The Basic analysis (see Slice Expressivity) is worth 75% of the total grade for the experimental part.

Attention will be given to:

- the precision of the tool
- the subset of the language that the tool handles (minimum requirements are the language features that appear in the slices provided; up to one value of extra credit will be given to solutions that go beyond those)
- the internal representation of the AST, modularity, quality of code and documentation

Report (25% of the project grade)
The maximum grade of the report does not depend on the complexity of the tool. It will of course reflect whether the analysis of the imprecisions matches the precision of the tool that was developed (which in turn depends on the complexity of the tool). The components of the grading are:

1. Quality of writing (5%)
2. State of the art (5%) - See minimum set of references in the project specification.
3. Identification of imprecisions (5%) - See questions 1. a) and 1. b) in project specification.
4. Understanding of imprecisions (5%) - See questions 2. a) and 1. b) in project specification.
5. Improving precision (5%) - See question 3 in project specification.

Have in mind the grading criteria specified in the project presentation.