fenix.tecnico.ulisboa.pt

FenixEdu™

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. This way, the problem becomes very similar to dynamic analysis, in which sequences of instructions really executed are analysed.

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. The way to call it depends on the language in which you choose to implement it (you can pick any you like), for instance:

```
./analyzer slice.php
./analyzer.py slice.php
java analyzer slice.php
```

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:

*   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:
    - imprecise tracking of information flows -- Are all illegal information flows captured by the adopted slicing technique? Are there flows that are unduly reported?
    - 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?

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

- 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.

Attachments

- proj-slices.zip