Initial Page
Groups
Evaluations
Bibliographic References
Schedule
Evaluation Methods
Objectives
Planning
Syllabus
Shifts
Announcements
Summaries
Marks
Class Logistics
Lectures
Labs
Office Hours
Project
Exercises/Tests
Materials
Project 2

Discovering vulnerabilities in JavaScript web applications

1. 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 analyze 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.

Components

The Project is presented in Section 2 as a problem, and its solution should have the following two parts:

1. An experimental component, consisting in the development and evaluation of a tool in your language of choice, according to the Specification of the Tool.
2. An analysis component, where the strengths and limitations of the tool are critically discussed and presented in a Report.

Submissions

All submissions should be done via your group's private repository at Git ~~Fenix~~, under the appropriate Group number (https://git.rnl.tecnico.ulisboa.pt/SSof2021/Project2-GroupX). The developed tool should be submitted as a zip file containing all the necessary code and a Readme.txt that specifies how the tool should be used. The reports should be submitted as a pdf.

Important dates:

- Any adjustments to groups (2 or 3 students) should be made by **20 November**.
- The submission deadline for the code is **6 December 23:59**. Please submit a zip file containing your code. All tests that you would like to be considered for the evaluation of your tool should be made available in a common repository (https://git.rnl.tecnico.ulisboa.pt/SSof2021/Project2-Tests), according to a forthcoming announcement.
- The submission deadline for the report is **13 December 23:59**.
- Demonstration and a discussion of the tool and report will take place during the lab and theoretical classes of the **last two weeks before Christmas break.**

Authorship

Projects are to be solved in groups of 2 or 3 students. All members of the group are expected to be equally involved in solving, writing and presenting the project, and share full responsibility for all aspects of all components of the evaluation. Participation in the discussion is mandatory for all group members.

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

2. Problem

A large class of vulnerabilities in 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 a potentially dangerous 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. For this reason, such flows can be deemed *illegal* for their potential to encode vulnerabilities.

It is often desirable to *accept* certain illegal information flows, so we do not want to reject such flows entirely. For instance, it is useful to be able to use the inputted user name for building SQL queries. 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 sanitize the user input, and by so restricting the power of the user to acceptable limits, in effect neutralizing the potential vulnerability.

JavaScript has been the primary language for application development in browsers, it is increasingly becoming popular on server side development as well. However, JavaScript suffers from vulnerabilities, such as cross-site scripting and malicious advertisement code on the the client side and on the server side from SQL injection.

The aim of this project is to study how web vulnerabilities can be detected statically by means of taint and input sanitization analysis. We choose as a target web server and client side programs encoded in the JavaScript language.

The following references are mandatory reading about the problem:

- C. Staicu et. al, "An Empirical Study of Information Flows in Real-World JavaScript", PLAS'2019.
- C. Staicu et. al, "Extracting Taint Specifications for JavaScript Libraries", CSE'20.
- S. Guarnieri et. al, "Saving the World Wide Web from Vulnerable JavaScript", ISSTA'11 and the companion "IBM JavaScript Security Test Suite".

3. Specification of the Tool

The experimental part consists in the development of a static analysis tool for identifying data and information flow violations that are not protected in the program. In order to focus on the flow analysis, **the aim is not to implement a complete tool**. Instead, it will be assumed that the code to be analyzed has undergone a pre-processing to isolate, in the form of a program slice, a sequence of JavaScript instructions that are considered to be relevant to our analysis.

The following code slice, which is written in JavaScript, contains code lines which may impact a data flow between a certain source and a sensitive sink. The `URL` property of the `document` object (which can be accessed in a client side script) can be understood as an entry point. It uses the `document.write` field to change the html page where it is embedded.

```
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos, document.URL.length));
```

Inspecting this slice it is clear that the program from which the slice was extracted could encode a DOM based XSS vulnerability. An attacker can inject a malicious request like `"http://www.vulnerable.com/welcome.html?name=<script>alert(document.cookie)</script>"`, modifying the behavior of the webpage in order to retrieve a cookie. However, sanitization of the untrusted input can remove the vulnerability:

```
var pos = document.URL.indexOf("name=") + 5;
var name = document.URL.substring(pos, document.URL.length);
var sanitizedName = encodeURI(name);
document.write(sanitizedName);
```

The aim of the tool is to search in the slices for vulnerabilities according to inputted patterns, which specify for a given type of vulnerability its possible sources (a.k.a. entry points), sanitizers and sinks:

- name of vulnerability (e.g., DOM XSS)
- a set of sources (e.g., document.URL),
- a set of sanitization functions (e.g., encodeURI),
- and a set of sensitive sinks (e.g., document.write).

The tool should signal potential vulnerabilities and sanitization efforts: If it identifies a possible data flow from an entry point to a sensitive sink (according to the inputted patterns), it should report a potential vulnerability; if the data flow passes through a sanitization function, it should still report the vulnerability, but also acknowledge the fact that its sanitization is possibly being addressed. **Obs: Assume that only the data that is returned by a sanitizing functions is UNtainted.**

We provide program slices and patterns to assist in testing the tool. It is however your responsibility to perform more extensive testing for ensuring the correctness and robustness of the tool. Your tool should be generic in order to function for vulnerabilities that can be expressed using sets of sources/sanitizers/sinks, as above. (Note that for the purpose of testing, the names of vulnerabilities, sources, sanitizers and sinks are irrelevant. In this context, you can produce your own patterns without specific knowledge of vulnerabilities, as this will not affect the ability of the tool to manage meaningful patterns.)

Running the tool

The tool should be called in the command line. All input and output is to be encoded in JSON, according to the specifications that follow.

Your program should take two arguments, which are the only input that it should consider:

- the name of the JSON file containing the program slice to analyse, represented in the form of an Abstract Syntax Tree;
- the name of the JSON file containing the list of vulnerability patterns to consider.

You can assume that the parsing of the JavaScript slices has been done, and that the input files are well-formed. The analysis should be fully customizable to the inputted vulnerability patterns.

The output should list the potential vulnerabilities encoded in the slice, and indicate which sanitizing instruction(s) (if any) have been applied. The format of the output is specified below.

The way to call your tool depends on the language in which you choose to implement it (you can pick any you like), but **it should be called in the command line with two arguments `<program>.json <patterns>.json` and produce the output referred below and no other to a file `<program>.output.json`**.

```
$ ./mytool program.json patterns.json
```

```
$ python ./mytool.py program.json patterns.json
```

```
$ java mytool program.json patterns.json
```

Input format

Program slices

Your program should read from a text file (given as first argument in the command line) the representation of a JavaScript slice in the form of an Abstract Syntax Tree (AST). The AST is represented in JSON, using the same structure as in Esprima.

For instance, the program

```
alert("Hello World!");
```

is represented as

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "Identifier",
          "name": "alert"
        },
        "arguments": [
          {
            "type": "Literal",
            "value": "Hello World!",
            "raw": "\"Hello World!\""
          }
        ]
      }
    }
  ],
  "sourceType": "script"
}
```

and the slice

```
t = document.URL;
document.write(t);
```

is represented as:

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "AssignmentExpression",
        "operator": "=",
        "left": {
          "type": "Identifier",
          "name": "t"
        },
        "right": {
          "type": "MemberExpression",
          "computed": false,
          "object": {
            "type": "Identifier",
            "name": "document"
          },
          "property": {
            "type": "Identifier",
            "name": "URL"
          }
        }
      }
    },
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "computed": false,
          "object": {
            "type": "Identifier",
            "name": "document"
          },
          "property": {
            "type": "Identifier",
            "name": "write"
          }
        },
        "arguments": [
          {
            "type": "Identifier",
            "name": "t"
          }
        ]
      }
    }
  ],
  "sourceType": "script"
}
```

Obs: changed the above slice in order to remove the variable declaration.

In order to parse the ASTs, you can use an off-the-shelf parser for JSON.

Besides the slices that are made available, you can produce your own ASTs for testing your program by using the Esprima javascript-to-json parser. You can visualize the JSON outputs as a tree using this online tool.

Vulnerability patterns

The patterns are to be loaded from a file, whose name is given as the second argument in the command line. You can assume that all inputted patterns correspond to different vulnerability names.

An example JSON file with two patterns:

```
[
  {"vulnerability": "Command Injection",
  "sources": ["readline","req.headers","file_path"],
  "sanitizers": ["escape", "sanitize"],
  "sinks": ["exec", "execSync", "spawn", "spawnSync", "execFile", "execFileSync"]},

  {"vulnerability":"DOM XSS",
  "sources":["document.referrer", "document.url", "document.location"],
  "sanitizers":["encodeURI"],
  "sinks":["eval", "document.write", "document.innerHTML", "setAttribute"]},
]
```

Output format

The output of the program is a `JSON` list of patterns represented as objects that should be written to a file `test.output.json` when the program under analysis is `test.json`. The structure of the objects should be:

- name of vulnerability (according to inputted patterns)
- input sources
- sensitive sinks (according to inputted patterns)
- sanitizing functions if present (according to inputted patterns), otherwise empty

As an example, the output with respect to the program and patterns that appear in the examples in Specification of the Tool would be the following, which includes one reported vulnerable flow:

```
[{"vulnerability":"DOM XSS",
"source":["document.URL"],
"sink":["document.write"],
"sanitizer":["encodeURI"]}]
```

Obs: It is possible that there are more than one vulnerable flows for a given vulnerability pattern. You are encouraged to produce output with distinct reports for the same vulnerability name whenever this enables you to achieve higher precision.

Precision and scope

The security property that underlies this project is the following:

Given a set of vulnerability patterns of the form (vulnerability name, a set of entry points, a set of sensitive sinks, a set of sanitizing functions), a program is secure if it does not encode, for any of the given vulnerability patterns, an information flow from one of its entry points to one of its sensitive sinks, unless the information goes through one of its sanitizing functions.

You will have to make decisions regarding whether your tool will signal, or not, illegal taint flows that are encoded by certain combinations of program constructs. You can opt for an approach that simplifies the analysis. This simplification may introduce or omit features that could influence the outcome, thus leading to incorrect results.

Note that the following criteria will be valued:

- Soundness, i.e. successful detection of illegal taint flows. In particular, treatment of implicit taint flows will be valued.
- Precision, i.e. efforts towards avoiding signalling programs that do not encode illegal taint flows. In particular, sensitivity to the order of execution will be valued.
- Scope, i.e. treatment of a larger subset of the language. The mandatory language constructs are those that appear in the slices provided, and include: **assignments, binary operations, function calls, condition tests and while loops.**

Obs: Using the terms in the definition of the Syntax Tree Format, the mandatory constructs are Literals, Identifiers, BinaryExpressions, CallExpressions, MemberExpressions (.), AssignmentExpressions, ExpressionStatements, IfStatements, WhileStatements and BlockStatements.

You are expected to incorporate precision and efficiency considerations, as discussed in the critical analysis criteria for the report, into the design and implementation of this component.

4. Report

The report should briefly describe the experimental part, present the design of the tool and the main design options, and discuss the guarantees provided by the tool, as well as its limitations, in light of the state of the art for the proposed problem.

Critical analysis

Consider the security property expressed in Precision and scope, and the security mechanism that is studied in this project, which comprises:

- A component (assume already available) that statically extracts the program slices that could encode potential vulnerabilities in a program.
- A tool (developed by you), that receives a configuration file containing vulnerability patterns, and signals potential vulnerabilities in given slices according to those patterns, as well as possible sanitization efforts, as implemented by you in the experimental part.

Given the intrinsic limitations of the static analysis problem, the developed tool is necessarily imprecise in determining which programs encode vulnerabilities or not. It can be unsound (produce false negatives), incomplete (produce false positives) or both.

1. Explain and give examples of what are the imprecisions that are built into the proposed mechanism. Have in mind that they can originate at different levels:
   - imprecise tracking of information flows -- Are all illegal information flows captured by the adopted technique? Are there flows that are unduly reported?
   - imprecise endorsement of input sanitization -- Are there sanitization functions that could be ill-used and not properly sanitized the input? Are all possible sanitization procedures detected by the tool?
2. 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?
3. Propose one way of making the tool more precise, and predict what would be the trade-offs (efficiency, precision) involved in this change.

Structure

Report your work and your conclusions in a written document, using **no more than 4 pages, single column**, (excluding references and appendices), according to the following guidelines:

- Use a structure that helps to read and find the relevant information.
- Assume a reader acquainted with the context of the Project (in other words go precisely and straight to the point!).
- Briefly describe the experimental part, presenting the architecture of the tool and the main design options (maximum 2 pages).
- Define and discuss what your tool is able to achieve, making sure that you answer all the above questions.

We suggest the following structure for the report:

1. Introduce the context of your work (1 paragraph) -- What is the problem that you want to solve? Why is it important?
2. Explain the approach (1 page, possible diagram) -- What is the underlying technique behind the tool?
3. Demonstrate the behavior of the tool (1 paragraph) -- How do you use it? What does it do?
4. Present the evaluation of the tool (2 paragraphs and link to performed tests) -- How did you test the correctness and robustness of the tool?
5. Critically analyse and discuss your solution (1-2 pages) -- What are the strengths and limitations of the tool?
6. Refer related work (2-3 paragraphs) - What other tools address the same problem for the considered language? What other tools use the same technique for solving similar problems?
7. Conclusion (1 paragraph) -- What have you achieved? What is the tool good for? What could you improve in the tool?

5. Grading

Project 2 is worth 30% of the final grade of the course (see Métodos de Avaliação).

Discussion

The baseline grade for the group will be determined based on the experimental part and report, acording to the rules below. During the discussion, all students in the groups are expected to be able to demonstrate knowledge of all details of these two components. In order to be graded for the project, each student must participate in the discussion, and his/her grade might be adjusted accordingly.

Experimental part

Grading of the Tool and Patterns will reflect the level of complexity of the developed tool, according to the following:

- Basic vulnerability detection (50%) - signals potential vulnerability based solely on explicit flows in slices with mandatory constructs
- Advanced vulnerability detection (25%) - signals potential vulnerability that can include implicit flows in slices with mandatory constructs
- Sanitization recognition (20%) - signals potential sanitization of vulnerabilities
- Distinguish patterns vulnerability data (5%) - respects which sources correspond to which sinks for different vulnerabilities, and which function may sanitize which vulnerability
- Bonus (5%) - treats other program constructs beyond the mandatory ones, extra effort for avoiding false positives

This part corresponds to 2/3 of the grade of Project 2.

Report

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 worth 20% each, and are the following:

- Quality of writing - structure of the report, clarity of the ideas
- Related work - depth of understanding of the related work, detachment from words used in cited papers. See mandatory references in Problem above.
- Identification of imprecisions - connection with experimental work. See questions 1.a) and 1.b) in Section Requirements for the discussion - Report above.
- Understanding of imprecisions - connection with experimental work. See questions 2.a) and 2.b) in Section Requirements for the discussion - Report above.
- Improving precision - originality, own ideas. See question 3 in Section Requirements for the discussion - Report above.
- Bonus (5%) - cites other references beyond the mandatory ones. See references in Problem above.

This part corresponds to 1/3 of the grade of Project 2.

Attachments

- proj-slices 2.zip