TÉCNICO LISBOA (http://tecnico.ulisboa.pt)

Segurança em Software (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre)

Discovering vulnerabilities in Python 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 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.

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. You are expected to take into account the analysis criteria that are to be reported later on into the design and implementation of this component.
2. An analysis component, where the strengths and limitations of the tool are critically discussed and presented. See Requirements for the Discussion. This component includes:
    - a report, that describes briefly the experimental part, presenting the design of the tool and the main design options, 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, and assuming a reader acquainted with the context of the Project.
    - a presentation, where the entire project, including the tool and conclusions from the report, are presented, having in mind a more general audience.

Submissions

All submissions should be done via Fenix, under the appropriate Group number. 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 and presentations should be submitted as a pdf.

Important dates:

- Groups (2 or 3 students) should register via Fenix by **8 November.**
- The submission deadline for the code is **27 November 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 (CREATE%20LINK).

- The submission deadline for the report is **4 December 23:59**.
- The submission deadline for the presentation slides is **before the beginning of the class in which they are presented**. It is recommended that you submit them two hours earlier, to avoid last minute troubles that could affect your presentation.
- Demonstration and a discussion regarding the tool and report will take place during the lab 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. Presence at the oral presentation and tool demonstration is mandatory for all group members.

All sources should be adequately cited. Plagiarism (https://en.wikipedia.org/wiki/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 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 sanitize 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 web vulnerabilities can be detected statically by means of taint and input sanitization analysis. We choose as a target web server side programs encoded in the Python language. There exist a range of Web frameworks (https://wiki.python.org/moin/WebFrameworks) for Python, of which Django is the most widely used. While examples in this project specification often refer to Django views (https://docs.djangoproject.com/en/2.2/topics/http/views/), the problem is to be understood as generic to the Python language.

The following references are mandatory reading about the problem:

- J. Conti and A. Russo, "Taint Mode for Python via a Library", OWASP 2010 (http://www.cse.chalmers.se/~russo/publications_files/owasp2010.pdf)
- V. Chibotaru et. al, "Scalable Taint Specification Inference with Big Code", PLDI 2019 (https://files.sri.inf.ethz.ch/website/papers/scalable-taint-specification-inference-pldi2019.pdf) Note: This paper contains a large component of machine learning that is not within the scope of this course, and which you may skip through.
- S. Micheelsen and B. Thalmann, "PyT - A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications", Master?s Thesis, Aalborg University 2016 (https://projekter.aau.dk/projekter/files/239563289/final.pdf)

3. Specification of the Tool

The experimental part consists in the development of a static analysis tool for identifying data and control flow integrity violations when inputs are not subject to proper input sanitization. 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 Python program slices, i.e., a sequence of instructions that are extracted from a program and that are considered to be relevant to our analysis.

The following code slice, which is written in Python, contains code lines which may impact a data flow between a certain entry point and a sensitive sink. The variable request (which for intuition can be seen as the request parameter of a Django view), is uninstantiated, and can be understood as an entry point. It uses the MySQLCursor.execute() method, which executes the given database operation query.

```
uname = retrieve_uname(request)
q = cursor.execute("SELECT pass FROM users WHERE user='%s'" % uname)
```

Inspecting this slice it is clear that the program from which the slice was extracted can potentially encode 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., request parameter),
- a set of sanitization functions (e.g., escape_string),
- and a set of sensitive sinks (e.g., execute).

The program 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 signal a potential vulnerability; if the data flow passes through a sanitization function, it should signal the fact that its sanitization is possibly being addressed.

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. While the examples that we provide are mostly within the Django framework, your tool should be generic in order to function for other slices and vulnerabilities that can be expressed with patterns of the format above. You can find data for forming vulnerability patterns in the appendix of Chibotaru et. al (https://files.sri.inf.ethz.ch/website/papers/scalable-taint-specification-inference-pldi2019.pdf).

Running the tool

The tool should be called in the command line. All input and output is to be encoded in JSON (http://www.json.org/), 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 Python slices has been done, and that the input files are well-formed. The analysis should be fully customizable to the inputted vulnerability patterns. In addition to the entry points specified in the patterns, by default any uninstantiated variable that appears in the slice is to be considered as an entry point to all vulnerabilities being considered.

The output should list the potential vulnerabilities encoded in the slice, and an indication of which instruction(s) might sanitize the data. 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`** .

```
$ ./bo-analyser program.json patterns.json
```

```
$ python ./bo-analyser.py program.json patterns.json
```

```
$ java bo-analyser program.json patterns.json
```

Input

Program slices

Your program should read from a text file (given as first argument in the command line) the representation of a Python slice in the form of an Abstract Syntax Tree (AST). The AST is represented in JSON, using the same structure as in Python's AST module (https://greentreesnakes.readthedocs.io/en/latest/).

For instance, the program

```
print("Hello World!")
```

is represented as

```
{
  "ast_type": "Module",
  "body": [
    {
      "ast_type": "Expr",
      "col_offset": 0,
      "lineno": 1,
      "value": {
        "args": [
          {
            "ast_type": "Str",
            "col_offset": 6,
            "lineno": 1,
            "s": "Hello World!"
          }
        ],
        "ast_type": "Call",
        "col_offset": 0,
        "func": {
          "ast_type": "Name",
          "col_offset": 0,
          "ctx": {
            "ast_type": "Load"
          },
          "id": "print",
          "lineno": 1
        },
        "keywords": [],
        "lineno": 1
      }
    }
  ]
}
```

and the slice

```
uname = retrieve_uname(request)
q = cursor.execute("SELECT pass FROM users WHERE user='%s'" % uname)
```

is represented as:

```
{
  "ast_type": "Module",
  "body": [
    {
      "ast_type": "Assign",
      "col_offset": 0,
      "lineno": 1,
      "targets": [
        {
          "ast_type": "Name",
          "col_offset": 0,
          "ctx": {
            "ast_type": "Store"
          },
          "id": "uname",
          "lineno": 1
        }
      ],
      "value": {
        "args": [
          {
            "ast_type": "Name",
            "col_offset": 23,
            "ctx": {
              "ast_type": "Load"
            },
            "id": "request",
            "lineno": 1
          }
        ],
        "ast_type": "Call",
        "col_offset": 8,
        "func": {
          "ast_type": "Name",
          "col_offset": 8,
          "ctx": {
            "ast_type": "Load"
          },
          "id": "retrieve_uname",
          "lineno": 1
        },
        "keywords": [],
        "lineno": 1
      }
    },
    {
      "ast_type": "Assign",
      "col_offset": 0,
      "lineno": 2,
      "targets": [
        {
          "ast_type": "Name",
          "col_offset": 0,
          "ctx": {
            "ast_type": "Store"
          },
          "id": "q",
          "lineno": 2
        }
      ],
      "value": {
        "args": [
          {
            "ast_type": "BinOp",
            "col_offset": 19,
            "left": {
              "ast_type": "Str",
              "col_offset": 19,
              "lineno": 2,
              "s": "SELECT pass FROM users WHERE user='%s'"
            },
            "lineno": 2,
            "op": {
              "ast_type": "Mod"
            },
            "right": {
              "ast_type": "Name",
              "col_offset": 62,
              "ctx": {
                "ast_type": "Load"
              },
              "id": "uname",
              "lineno": 2
            }
          }
        ],
        "ast_type": "Call",
        "col_offset": 4,
        "func": {
          "ast_type": "Attribute",
          "attr": "execute",
          "col_offset": 4,
          "ctx": {
            "ast_type": "Load"
          },
          "lineno": 2,
          "value": {
            "ast_type": "Name",
            "col_offset": 4,
            "ctx": {
              "ast_type": "Load"
            },
            "id": "cursor",
            "lineno": 2
          }
        },
        "keywords": [],
        "lineno": 2
      }
    }
  ]
}
```

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

Besides the slices that are made available, you can produce your own ASTs for testing your program by using a python-to-json parser (https://pypi.org/project/astexport/). You can visualize the JSON outputs as a tree using this online tool (http://jsonviewer.stack.hu/). The parser can also be used for building a stand-alone tool --

though this is **not** valued within the context of this project.

Vulnerability patterns

The patterns are to be loaded from a file, whose name is given as the second argument in the command line. Patterns with the same name are to be combined internally into a single one.

An example file with three patterns, two of which correspond to the same vulnerability:

```
SQL injection
get,get_object_or_404, QueryDict, ContactMailForm, ChatMessageForm
mogrify, escape_string
execute

SQL injection
QueryDict, ContactMailForm, ChatMessageForm, copy, get_query_string, get_user_or_404,
User
mogrify, escape_string
raw,RawSQL

XSS
get, get_object_or_404, QueryDict, ContactMailForm, ChatMessageForm
clean,escape,flatatt,render_template,render,render_to_response
send_mail_jinja,mark_safe,Response,Markup,send_mail_jinja,Raw,HTMLString
```

Their representation in JSON should be:

```
[
  {"vulnerability": "SQL injection",
  "sources": ["get", "get_object_or_404", "QueryDict", "ContactMailForm", "ChatMessage
Form"],
  "sanitizers": ["mogrify", "escape_string"],
  "sinks": ["execute"]},

  {"vulnerability":"SQL injection",
  "sources":["QueryDict", "ContactMailForm", "ChatMessageForm", "copy", "get_query_str
ing"],
  "sanitizers":["mogrify", "escape_string"],
  "sinks":["raw", "RawSQL"]},

  {"vulnerability":"XSS",
  "sources":["get", "get_object_or_404", "QueryDict", "ContactMailForm", "ChatMessageF
orm"],
  "sanitizers":["clean","escape","flatatt","render_template","render","render_to_respo
nse"],
  "sinks":["send_mail_jinja","mark_safe","Response","Markup","send_mail_jinja","Raw"]}
]
```

Output

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 ~~variable~~ **source**
- sensitive sink (according to inputted patterns)
- sanitizing function if present (according to inputted patterns), otherwise empty

As an example, the output with respect to the program and patters that appear in the examples in Specification of the Tool would be:

```
{"vulnerability":"SQL injection",
"source":"request",
"sink":"execute",
"sanitizer":""}
```

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 given vulnerability pattern, an information flow from an entry point to a sensitive sink, unless the information goes through a sanitizing function.

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 wrong 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 test and while loop**.

4. Requirements for the Discussion

Report

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

- Statically extracting the program slices that encode potential vulnerabilities in a program, similarly to as described in I. Medeiros et al. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.432.7100&rep=rep1&type=pdf).
- Signaling the slices that potentially encode a vulnerability, and validating those in which input is considered to have been sanitized, according to what you implemented 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.

Report your work and your conclusions in a written document, using no more than 4 pages (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 design of the tool and the main design options (maximum 2 pages).
- Define and discuss what your tool is able to achieve, while answering the above questions.

Presentation

Present your work assuming an audience that is not familiar with the course, while answering the following questions.

1. Introduce the context of your work -- What is the problem that you want to solve? Why is it important?
2. Explain the approach -- What is the underlying technique behind the tool?
3. Demonstrate the behavior of the tool -- How do you use it? What does it do?
4. Present the evaluation of the tool -- How did you test the correctness and robustness of the tool?
5. Strengths and limitations of the tool -- What is the tool good for? What could be improved in the tool?
6. Refer related work - What other tools address the same problem for the considered language? What other tools use the same technique for solving similar problems?
7. Conclusion -- What have you learned? What is your personal opinion about the suitability of the technique for solving this problem?

Base your presentation on digital slides, according to the following guidelines:

- For a presentation of X minutes, use no more than X+2 slides.
- Avoid exceeding ~7 lines per slide, ~7 words per line.

5. Grading

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 (20%) - signals potential vulnerability that can include implicit flows in slices with mandatory constructs
- Sanitization recognition (20%) - signals potential sanitization of vulnerabilities
- Patterns (5%) - provided two original complete patterns
- Distinguish patters vulnerability data (5%) - Distinguish patters vulnerability data (respecting 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 20% of the final grade (referred to as "Project" in Métodos de Avaliação (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/metodos-de-avaliacao)).

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 10% of the final grade (50% of what is referred to as "Analysis" in Métodos de Avaliação (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/metodos-de-avaliacao)).

Presentation

The evaluation of the presentation will be based on the format and content only -- the slide struture and the answers that it gives to the questions in Section Requirements for the discussion - Presentation above.

This part corresponds to 10% of the final grade (50% of what is referred to as "Analysis" in Métodos de Avaliação (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/metodos-de-avaliacao)).

Attachments

- proj-slices.zip (https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382233709/proj-slices.zip)

Página Inicial (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/pagina-inicial) (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/rss/announcement)

Grupos (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/grupos)

Avaliação (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/avaliacao)

Bibliografia (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/bibliografia)

Horário (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/horario)

Métodos de Avaliação (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/metodos-de-avaliacao)

Objectivos (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/objectivos)

Planeamento (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/planeamento)

Programa (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/programa)

Turnos (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/turnos)

Anúncios (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/anuncios) (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/rss/announcement)

Sumários (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/sumarios) (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/rss/summary)

Notas (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/notas)

Labs and Practical classes (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/new-entry)

Office Hours (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/office-hours)

Slides (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/slides)

Exercises/tests (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/exercisestests)

Project (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/project)

Additional study materials (https://fenix.tecnico.ulisboa.pt/disciplinas/SSof7/2019-2020/1-semestre/additional-study-materials-a34)