# INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

## FORENSICS CYBER-SECURITY

MEIC, METI

# Lab Assignment I

### DEEP BREACH – Stage I

2020/2021

nuno.m.santos@tecnico.ulisboa.pt

## Introduction

Throughout a series of three lab assignments, you will be conducting the investigation of a case entitled "Deep Breach". It will be the case of a data breach that took place on a company by means of a security vulnerability that was present on their website. Their website was powered by a NodeJS application. In this first guide, you will gain hands-on training on how to analyze security vulnerabilities in the source code of real-world NodeJS packages. This skill will be fundamental for the stages that follow. This exercise will require the examination of a set of advisory reports from the NodeJS's npm repository.

## Web Vulnerability Analysis Training

With the rise of security attacks to NodeJS-based web sites, being able to analyze potential vulnerabilities in JavaScript code has become increasingly important. As of today, there are hundreds of advisory reports documenting security vulnerabilities in npm packages. Unfortunately, many of such reports are ill-documented. To make matters worse, existing vulnerability scanners are quite limited, failing to automatically detect existing vulnerabilities, and overwhelming application developers with false positives. In light of this, it is crucial for a forensic analyst to gain hands-on training in the analysis of advisory reports: first, by being able to identify the root cause of a security vulnerability, and second, by critically assessing the quality of existing vulnerability scanners for JavaScript code.

In this exercise, each group will be given a set of 25 advisory reports to analyze. For each advisory report, you have to accomplish two main goals: (G1) identify the code snippet that causes the vulnerability described in the advisory report, and (G2) characterize the effectiveness of each vulnerability scanner – 9 tools in total – at correctly identifying each vulnerability. The final grade will be based on whether both these goals have been accomplished. The methodology to adopt for this task has been presented in the Tutorial II; the only difference is that you have to download an updated version of the analysis framework (`lab1-framework.zip`) from our website. You should constantly refer to that document to learn what to do to analyze a particular vulnerability. The set of advisory reports assigned to each group can be found in the accompanying document `lab1-reports.pdf`, also available on the course website.

For each report, your will have to provide an *examination file*, written in JSON, where your results for G1 and G2 will be included. In appendix, we provide further instructions of how to write such files. Note that you must provide an examination file for each advisory report assigned to your group. You will also have access to a script that will help you validate the JSON format of your examination files. Submitted files that fail to validate against our expected schema run the risk of not being accepted!

Each advisory report is ranked by an increasing level of difficulty, from L1 to L4. We suggest you to begin with L1 reports and then proceed by increasing levels of difficulty. For some reports, it might not be possible to determine the root cause of the vulnerability (snippet location), e.g., due to insufficient available information. In such cases, you must provide (in the examination file) a brief justification for why that was the case. You will not be penalized in your grade if your justification is compelling.

## Deliverables

The deadline for this work is October $30^{th}$. Until then, you must submit the following two deliverables:

- **Examination files**: A compressed zip file named `lab1-group`*XX*`.zip` (where *XX* is your group number) containing the examination files of the advisory reports assigned to your group. We expect one examination file, written in JSON and properly validated, per advisory report. Each examination file must be named as *ID*`.json`, where *ID* is the ID of the respective advisory report.
- **Questionnaire:** You will be asked to fill in a questionnaire about your experience in conducting this exercise. This questionnaire will be made available in due time on the course website. Every student must answer this questionnaire.

Good luck!

# A Structure of an Examination File

For each advisory report, you must generate an examination file. Its file name must follow the format *ReportID*`.json`, where *ReportID* is the ID of the respective advisory report. Internally, the file is structured in JSON, and it must include several fields. The example below shows the expected structure of an examination report that was produced for the advisory report number 1020, which was studied in detail in the Tutorial II. This example can also be downloaded from the course website (file `1020.json`).

```
{
    "advisory": {
        "id": 1020,
        "cwe": "CWE-78"
    },
    "correct_cwe": "CWE-78",
    "correct_package_link": "https://registry.npmjs.org/local-devices/-/local-
        devices-2.0.0.tgz",
    "vulnerability": {
        "vulnerability_location": [
            {
                "source": {
                    "file": "src/index.js",
                    "lineno": 16,
                    "code": "module.exports = function findLocalDevices (address) {"
                },
                "sink": {
                    "file": "src/index.js",
                    "lineno": 114,
                    "code": "return cp.exec('arp -n ' + address).then(parseOne)"
                }
            }
        ],
        "fail_reason": ""
    },
    "poc": [
        {
            "url": "https://github.com/DylanPiercey/local-devices/pull/16"
        }
    ],
    "patch": [
        {
            "url": "https://github.com/DylanPiercey/local-devices/pull/16/commits
                /7001c49249816be4ead9ec039573d49307b91487"
        }
    ],
    "tools": {
        "codeql": {
            "score": "A"
        },
        "njsscan": {
            "score": "E"
        },
        "graudit": {
            "score": "C"
        },
        "insidersec": {
            "score": "D"
        },
        "eslint": {
            "score": "C"
        },
        "appinspector": {
            "score": "C"
        },
```

```
        "msdevskim": {
            "score": "D"
        },
        "drek": {
            "score": "D"
        },
        "mosca": {
            "score": "C"
        }
    }
}
```

This JSON document contains several different properties. When writing your own examination files, it is very important that you follow the expected format that we have prescribed. To assist you in this task, we provide a script named `validate-json.sh` which takes as input your JSON file and checks its internal structure against a valid document schema. We recommend that you use the example shown above as a template for writing your own examination files. Next, we describe in detail each of the several properties that must be provided and where you can obtain that information.

**1. Advisory report identification (mandatory):** This parameter (*advisory*) is used to identify the advisory report characterized in this document. It contains two fields: *id* and *cwe*. These fields can be obtained directly from the advisory report's description.

```
"advisory": {
    "id": 1020,
    "cwe": "CWE-78"
}
```

**2. Metadata correction (mandatory):** We include two additional parameters (*correct_cwe* and *correct_package_link*) to help fix potential mistakes in the metadata available at the analysis framework. For example, you may find that an advisory initially reported as a code injection vulnerability should in fact be reported as an OS command injection, thus you report a *correct_cwe* field of *CWE-78* (Improper Neutralization of Special Elements used in an OS Command - "OS Command Injection"). You may also have to report a *correct_package_link* if you find out that the package link provided in the advisory metadata does not correspond to a vulnerable version.

```
"correct_cwe": "CWE-78",
"correct_package_link":
    "https://registry.npmjs.org/local-devices/-/local-devices-2.0.0.tgz",
```

Note that if both the *cwe* and *package_link* metadata fields are correct then you should still report these fields using the original parameters.

**3. Vulnerable code snippet (mandatory):** This parameter (*vulnerability*) is one of the most important parameters in this document as it will describe your findings for goal G1. It will be used for you to describe the code snippet that causes the vulnerability described in the advisory report. Section B will be exclusively dedicated to describe the format of this parameter.

**4. Proof-of-concept exploit (optional):** The proof-of-concept exploit (*poc*) is a parameter that links to a resource useful in reproducing the vulnerability. It can be either a small *code* example, the *path to a local file* you created or an *url* to an external resource. If it is a path to a local file, this file must be included in the final zip file to be submitted to Fenix. Although in the example below this parameter contains all three fields, this is just for demonstration purposes: oftentimes you only need to provide one of these fields (*url*, *code*, or *file_path*) to fully characterize the PoC.

```
"poc": [
    {
        "url": "https://github.com/DylanPiercey/local-devices/pull/16"
    },
    {
        "code": "var userInput = '127.0.0.1 | mkdir attacker';\nfind(userInput);"
    },
    {
        "file_path": "1020.poc.js",
    }
]
```

This parameter is in fact an array (i.e., you can add multiple sub-blocks) because you may find different PoC exploits worth registering. In some situations, the person that originally reported the vulnerability may have supplied a PoC exploit that you can use here. If you do not find any available exploits, you are not required to include this parameter.

If you intend to provide a PoC exploit as a file we suggest you to use the following naming convention *ReportID*.poc.*extension*, where *ReportID* is the ID of the respective advisory report and extension is the file extension of the language used to write the PoC exploit.

**5. Available code patch (optional):** A patch (*patch*) fixes a particular vulnerability. It can be either the *path to a local file* you created or an *url* to an external resource. If it is a path to a local file, this file must be included in the final zip file to be submitted to Fenix. Although in the example below this parameter contains all two fields, only one of these fields needs to be provided.

```
"patch": [
    {
        "url": "github.com/package/commit/commitid"
    },
    {
        "file_path": "1020.patch"
    }
]
```

This parameter is also an array because you may find different possible patches to a vulnerability. In some situations, the external references contained in the advisory report may point to resources containing a patch. If you do not find any information about available patches, you can leave this parameter unfilled.

If you intend to provide a patch as a file we suggest you to use the following naming convention *ReportID*.patch, where *ReportID* is the ID of the respective advisory report.

**6. Vulnerability scanner performance (mandatory):** This parameter (*tools*) is another very important parameter, because it will report your assessment of the most popular vulnerability scanning tools for JavaScript (i.e., it will describe your results for goal G2). Section C presents it more thoroughly.

# B Vulnerable Code Snippet

One of the most important parameters in this document is the parameter *vulnerability*, which identifies the code snippet responsible for the vulnerability described in the advisory report. For this purpose, it contains one field named *vulnerability_location* which can be filled in two possible ways depending on the pattern of how the vulnerability is triggered in the code: *source/sink* or *block*.

**1. Source/sink pattern:** Most vulnerabilities are characterized by a pair of *source* and *sink* code elements. Typically, these elements are either functions or variables that get input from the user or receive and handle user input incorrectly, respectively. For example, a function that returns a user supplied string

can be considered a *source*, whilst the JavaScript function *eval* is a dangerous *sink*, because when user-controlled data reaches this function it can trigger a vulnerability (in this case a JavaScript code injection vulnerability). The example below shows how this pattern could be expressed in the JSON document:

```
"vulnerability": {
        "vulnerability_location": [
            {
                "source": {
                    "file": "src/index.js",
                    "lineno": 16,
                    "code": "module.exports = function findLocalDevices (address) {"
                },
                "sink": {
                    "file": "src/index.js",
                    "lineno": 114,
                    "code": "return cp.exec('arp -n ' + address).then(parseOne)"
                }
            }
        ],
        "fail_reason": ""
    },
```

More specifically, this example tells that the source of the vulnerability is located in file `src/index.js`, line number 16, and the sink is placed in the same file, line number 114. The corresponding verbatim code is also provided in the *code* sub-fields of the *source* and *sink* fields.

Note that *vulnerability_location* is an array because there may be several possible flows (pairs of source and sink) for the same vulnerability. For example different *sources* may reach the same *sink*, or one *source* may reach different dangerous *sinks* depending on the program flow. In such cases, you need to provide multiple source/sink pairs to reflect the various different flows that can trigger that vulnerability.

**2. Block pattern:** Some other vulnerabilities may not be correctly characterized as a *source-sink* pair flow. In that case you should use the *block* property inside the *vulnerability_location* parameter:

```
"vulnerability": {
        "vulnerability_location": [
            {
                "block": {
                    "file": "index.js",
                    "start_lineno": 11,
                    "end_lineno": 12,
                    "code": "var downloader = require('unsafe-downloader');\
                        ndownloader.download('http://dadossensiveis.com');"
                }
            }
        ],
        "fail_reason": ""
    }
```

**Unable to determine the vulnerability location:** If you cannot identify the vulnerable snippet for a particular report then you should write a valid reason in the *fail_reason* property.

```
"vulnerability": {
        "fail_reason": "Insufficient information in Overview, External References or
            Tool Reports (...)"
    }
```

# C Vulnerability Scanner Performance

The parameter *tools* allows us to assess the effectiveness of automated vulnerability detection tools that can be useful in assisting analysts in the future. It is also how you will describe your results for goal G2. There are 9 properties corresponding to the name of each tool that was executed to scan vulnerabilities in the context of a given advisory report. For each tool, you will need to analyze its output, and assign a performance score for that particular code analysis. The listing below shows a classification example.

```
"tools": {
    "codeql": {
        "score": "A"
    },
    "njsscan": {
        "score": "E"
    },
    "graudit": {
        "score": "C"
    },
    "insidersec": {
        "score": "D"
    },
    "eslint": {
        "score": "C"
    },
    "appinspector": {
        "score": "C"
    },
    "msdevskim": {
        "score": "D"
    },
    "drek": {
        "score": "D"
    },
    "mosca": {
        "score": "C"
    }
}
```

The tool performance score varies from A (best) to E (worst), and it can be explained as follows:

- A: The tool correctly detects and classifies a vulnerability (e.g. "Command injection") and that vulnerability matches the problem reported in the advisory report;
- B: The tool detects an error / warning in the code snippet of the reported vulnerability, but it is unable to classify the detected error / warning as the vulnerability (i.e., the tool senses that an error exists, but cannot tell that it constitutes the security vulnerability described in the advisory report);
- C: The tool detects some other vulnerability / error / warning in the code which does not match the one described in the advisory report;
- D: The tool can parse and process the code, but no vulnerability / error / warning are detected.
- E: The tool output shows errors that occurred during the analysis (parsing errors, timeout errors, etc.) and that, consequently, lead to no results by that tool.

**Example 1:** As an example let's look at an excerpt of the *csv* output of the *codeql* tool for the advisory 1020:

```
"Unsafe shell command constructed from library input",
"Using externally controlled strings in a command line may allow a malicious user to change the meaning of the command.",
```

```
"error",
"[[""String concatenation""|""relative:///src/index.js:114:18:114:36""]] based on
    library input is later used in [[""shell command""|""relative:///src/index.js
    :114:10:114:37""]].",
"/src/index.js",
"114",
"18",
"114",
"36"
```

The output generated by codeql consists of a list of alerts, each alert being comprised of 9 properties[1]: alert name, description, severity, alert message, path, start line, start column, end line, and end column. The fragment of the results shown above, displays a single alert message and all its specific properties. We can see in this particular case that *codeql* is able to detect a vulnerability given by the alert name *Unsafe shell command constructed from library input*. This description hints to the existance of a possible command injection and, thus, we infer that codeql can correctly detect the *sink* present in line 114. For this reason this tool can be classified with the "A" score.

**Example 2:** As another example, the tool *mosca* detects a possible vulnerability, but it is unrelated to that of the current report. So it get a score of "C".

```
<report_mosca>
    <Path>
        /src/src/index.js
    </Path>
     <Module>
        /app/Mosca/eggs/javascript_common_fail.egg
    </Module>
     <Title>Possible code injection</Title>
     <Description>
     Command injection is an attack in which the goal is execution of arbitrary
         commands on the host operating system via a vulnerable application.
     </Description>
     <Level>High</Level>
     <Reference>
        https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId
            =2130132
    </Reference>
     <Match> eval\s?\(|setTimeout|setInterval</Match>
     <Result>
        Line: 62 -      socket.setTimeout(1000, close)
    </Result>
</report_mosca>
```

**Example 3:** A last example, the tool *EsLint Security Scanner Configs* shows 2 warnings, but none of those correspond to the vulnerability reported. As such, this tool also gets a "C" score.

```
/src/src/index.js
  63:5  warning  The function setTimeout can be unsafe  scanjs-rules/call_setTimeout
  64:5  warning  The function connect can be unsafe     scanjs-rules/call_connect

x 2 problems (0 errors, 2 warnings)
```

# D Framework Zip File

Lastly, we provide a brief description of the provided zip file (`lab1-framework.zip`). It contains an updated version of the web vulnerability analysis framework used in lab class Tutorial II. You can follow that lab guide for instructions on how to run the framework.

[1] https://help.semmle.com/codeql/codeql-cli/procedures/analyze-codeql-database.html#results

**General directory structure:** Inside the zip archive you will find one *docker-compose.yml* file and three directories. This file and both the *flask-app* and *mongo_entrypoint* directories are necessary to run the framework. The directory *schema-validator* contains information on how to validate your JSON files.

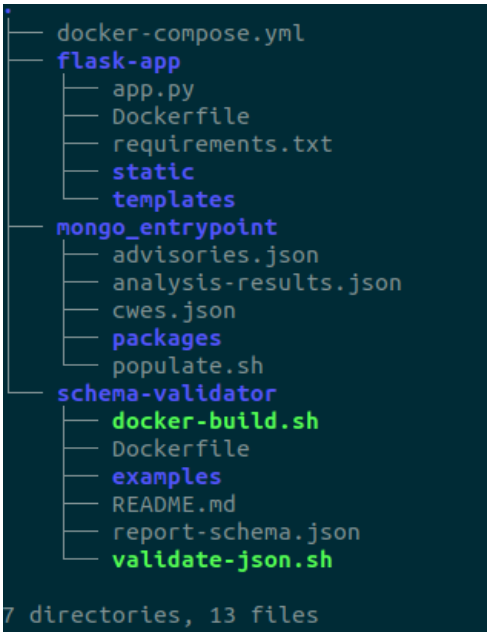


**Figure 1:** Directory structure of the framework zip file.

**Schema validator:** The *schema-validator* folder contains two scripts, one *Dockerfile*, a README file and the JSON schema used for validation. You also have examples of both valid and invalid JSON files inside the *examples* directory. The README contains instruction on how to run the validator script.

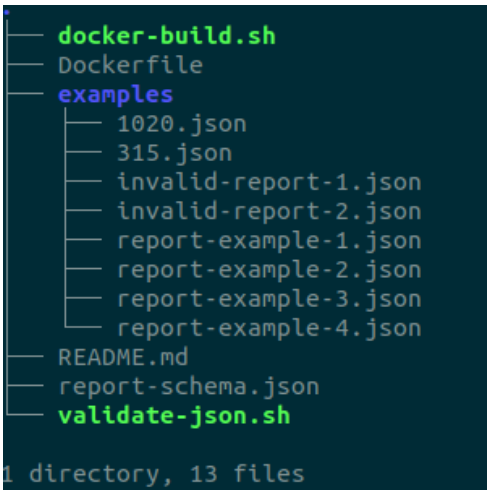


**Figure 2:** Directory structure of the *schema-validator* directory.