

TALLINN UNIVERSITY OF TECHNOLOGY  
Faculty of Information Technology  
Department of Computer Science

# **Analysis of Node.js platform web application security**

Master's thesis

Student:	Karl Dööna
Student code:	106851IVCM
Supervisor:	Andres Ojamaa

Tallinn  
2012

**Author's declaration**

Herewith I declare that this thesis is based on my own work. All works and major viewpoints of other authors, data from other sources of literature and elsewhere have been referenced. This thesis has not been submitted for any degree or examination in any other university.

(date)

(signature)

## Abstract

The security of a web application composes of not only the realisation of the application, but also the underlying platform and the various technologies that have been used. This thesis analyses the security aspects of one relatively new platform – Node.js. Although the thesis focuses on the security aspects of a web application, a large part of it can be expanded to all Node.js platform network applications.

Node.js platform has two important differences compared to traditional web application platforms. Firstly the platform is event-based and uses one main thread for its event loop, which forces developers to use asynchronous interfaces for I/O operations. This architecture tries to simplify the creation of big scalable web applications. The thesis investigates how the architectural choices of Node.js affect the security of applications running on it.

The second important difference is that Node.js applications are written in JavaScript. JavaScript is a dynamic programming language with functions as first class citizens and that, among other features, supports changing and expanding objects as well as defining functions at runtime. It is widely used on web pages, where it is run in a sandbox defined by the browser. Using this language on the server side, where the process lifetime is longer, must be secure and able to service many clients concurrently, challenges the developers and requires some change in current practises.

This thesis analyses the possible dangers and weaknesses of using Node.js platform and server side JavaScript. Example applications are also tested against common Denial of Service attacks. The final chapter of the thesis gives recommendations for writing and configuring secure and stable web applications on Node.js platform.

## Annotatsioon

Võrgurakenduste turvalisus ei sõltu üksnes rakenduse enda realiseerimisest, vaid ka platvormist, millel see töötab, ning tehnoloogiatest, mida rakenduse loomiseks on kasutatud. Käesolevas töös uuritakse ühe suhteliselt uue võrgurakenduste platvormi -- Node.js -- eripärasid turvalisuse seisukohalt. Kuigi töö keskendub veebirakenduste turvaküsimustele, laieneb suur osa sellest kõigile Node.js platvormi võrgurakendustele.

Node.js platvormil on traditsiooniliste veebirakenduste platvormidega võrreldes kaks olulist erinevust. Esiteks iseloomustab platvormi ühe pealõimega sündmuspõhine arhitektuur, mis sunnib rakenduste arendajad I/O-operatsioonide jaoks läbivalt kasutama asünkroonset liidest. Selle arhitektuuriga püütakse lihtsustada suure jõudlusega skaleeruvate võrgurakenduste realiseerimist. Käesolevas töös uuritakse, kuidas Node.js platvormi arhitektuurilised valikud mõjutavad rakenduste turvalisust.

Teine Node.js platvormi oluline eripära on, et rakendused kirjutatakse JavaScriptis. JavaScript on dünaamiline programmeerimiskeel, mis toetab muu hulgas kõrgemat järku funktsioone ning töö ajal objektide laiendamist ja funktsioonide defineerimist. JavaScript on laialdaselt kasutusel veebilehtedel, kus programmi jooksutatakse kliendi veebilehitsejas rangelt piiratud keskkonnas. Selle keele kasutamine serveri poolel, kus protsess töötab kaua, peab olema turvaline ning suutma teenindada korraga paljusid kliente, esitab arendajatele uusi väljakutseid ning nõuab seniste praktikate muutmist.

Selles töös analüüsitakse Node.js platvormi ning serveripoolse Javascripti kasutamise võimalikke ohte ja nõrkusi. Katsetatakse ka näidisrakenduse vastupidavust tuntud teenusetõkestusrünnete tehnikate suhtes. Töö viimases osas antakse soovitusi turvaliste ja stabiilsete Node.js platvormi veebirakenduste realiseerimiseks ning häälestamiseks.

## Contents

Introduction .....	7
1 JavaScript language peculiarities and their effect .....	9
1.1 History of JavaScript .....	9
1.2 JavaScript syntax .....	10
1.2.1 Function as a variable .....	10
1.2.2 Variable scope and global objects .....	11
1.2.3 JavaScript „with“ statement.....	12
1.2.4 Floating point numbers in JavaScript .....	14
1.2.5 Eval and other runtime interpreter functions .....	14
2 Security of a web server written on Node.js .....	16
2.1 What is Node.js? .....	16
2.2 Server main event loop .....	17
2.2.1 Run-time server poisoning .....	18
2.2.2 Denial of Service .....	21
2.3 Server configuration .....	26
2.3.1 Error handling in Node.js .....	27
2.3.2 Node.js memory usage .....	28
2.4 Rapid changes in Node.js.....	30
2.4.1 Staying behind the Node.js development .....	31
2.4.2 Node.js modules .....	34
3 Recommendations for improving application security .....	41
3.1 Know what to use it for.....	41
3.1.1 Calculations .....	41
3.2 Logging .....	44
3.3 Memory limitations.....	44
3.4 Programming .....	45
3.4.1 use strict.....	45
3.4.2 Try...catch.....	47
3.4.3 Object properties.....	48
3.5 Modules .....	49
3.6 Reliability of the application.....	50

Conclusion.....	52
References .....	53
Figures .....	55
Appendix 1 – The World of ECMAScript.....	57
Appendix 2 – Testing configuration.....	58
Host machine: .....	58
Test machines: .....	58
Software: .....	58
Appendix 3 – Test server code .....	59

## Introduction

The increasing technological capabilities and insatiable need for information of internet users has lead to the development of real-time web services. Real-time web services are services that allow users to receive information the moment it is published by the authors instead of requiring them to do periodic checks for updates. These include all manners of information feeds, but even more commonly web chat clients, which can be found on a variety of different sites.

Conventional web transactions were designed with regards to loading full documents with one request. Real time communication entails sending new information as soon as it becomes available, which usually resolves into sending lots of small document fragments. In order to avoid huge bandwidth overhead, associated with the creation of HTTP connections, long-term connections are preferred between the different parties. Popular web servers like Apache HTTP Server [1] were designed for serving full documents instead of holding on to long term connections.

The Apache HTTP Server, like many other servers, uses the threaded computational model. In this model for each request a new thread is created. Because threads consume system resources a set number of threads can be active simultaneously. Long-term connections hold those threads alive, so the number of clients that a server can handle is limited by the number of threads it can run. Evented computation models do not keep threads to handle requests, but instead react to various events, like a request, in the system. This, in general, allows those types of servers to handle more continuous requests at the same time.

One of the recently developed event-driven platforms that also allows the easy creation of web servers is Node.js. Its development began in early 2009 when Ryan Dahl decided to create an asynchronous networking server. It was first introduced to the world at JSConf 2009. Node.js provides a way to develop network applications with the widely used JavaScript language. It also came with its own package manager, which made adoption easier for new developers. [2]

Node.js provided solutions to several issues that had risen from the need for real-time communications and so it gained support from several large companies in the web industry

like Google [3] and Microsoft [4]. This gave a boost to the development of this platform. By now an active community has grown around Node.js and its development and it is expanding.

Although Node.js is being adopted into real-time web fast, little research has been done as to the security of sites running on Node.js. No thorough security analysis's can be found and many developers do not seem to be considering the security implications of migrating to this new platform.

This thesis provides an analysis of a web server running on Node.js. It focuses on the different security aspects of web services and gives an assessment of the risks for the possible attack vector based on the Confidentiality Integrity Availability (CIA) benchmark. Recommendations and possible solutions to the most common problems that can rise when developing applications on this technology are also provided.

This thesis is organised into three main chapters. Chapter 1 analyses the language in which Node.js applications are developed – JavaScript. After giving a brief introduction to the language history and state, it provides an overview of possible pitfalls and problems that are introduced to Node.js by it.

In Chapter 2 an analysis of a server built on Node.js from the standpoint of its application design and development is provided. Several different attack vectors and their effect on the server are also handled.

Chapter 3 provides recommendations and mitigation solutions for the previously shown problems. Following these propositions should make the application more secure and resilient to attacks.



# 1 JavaScript language peculiarities and their effect

In this chapter the language of Node.js applications – JavaScript, will be analysed. First a brief overview of the language will be given and then the possible pitfalls that the language introduces into Node.js will be outlined.

## 1.1 History of JavaScript

JavaScript was developed by Brendan Eich under the name Mocha as an enhancement to the Netscape browser and renamed to LiveScript at its beta release in 1995. It was later renamed to JavaScript as a marketing move and as such was officially released with Netscape 2.0 in March 1996. Microsoft implemented its own version of the language called JScript. The two languages were standardized under the name ECMA-262 (ECMAScript) when European Computer Manufacturers Association (ECMA) approved Netscape's application in 1996. Since then ECMA has published 5 versions of the specifications with the latest version - 5.1 being released in June 2011. [5]

While the language has an official standard in the name of ECMAScript it is still developed separately by Microsoft under the name of JScript. There are other implementations of the ECMAScript standard and different engines that run them. Of these JScript and JavaScript are often confused with one another and produce many problems on the browser side. Although outdated, the mapping of different JavaScript implementations and connections done in 2007 [Appendix 1] gives an idea of how varied the landscape is.

On the server side the first implementation of JavaScript was by Netscape in 1996. Netscape was planning to introduce a browser written only in Java and thus needed a Java implementation of JavaScript – project „Javagator“. Although the project was terminated, its subproject „Rhino“ remained [6]. Rhino passed on to mozilla.org project (later to become Mozilla Foundation) in 1998 and has been under its development since. It was not accepted by a wider user base and thus has had a slow progress. The current version conforms to ECMAScript 3, which is over a decade old. [7]

In 2006 Google was working on entering the browser market and needed a JavaScript engine to run in their browser. Development started on a JavaScript engine named V8 [8]. V8 compiles JavaScript into native machine code before executing it to increase the performance. The engine itself is developed to be embeddable, allowing it to be used in

any environment supporting C++. This appealed to Ryan Dahl who adopted V8 in his project to build a new application platform in the beginning of 2009. By the end of 2009 Node.js was in version 0.1.24 and still in its infancy. Over the next years it gained popularity and by 2012 May it is in version 0.6.16 and shows no signs of slowing down. [9]

JavaScript has had a chaotic development due to its multi-faced nature which held back the adoption of new features for quite some time. It started out as a simple scripting language meant to provide some interactivity to static HTML web pages. By now it has grown into a powerful programming language which ships on every desktop computer. This simple beginning accounts for the lack of built in security in JavaScript and causes some specific problems when developing applications on Node.js.

## 1.2 JavaScript syntax

JavaScript's first version was developed in 1995 with the goal of creating a simple scripting language. The aim was to complement Java with a lightweight interpreted language. Brendan Eich, who was the lead developer, took most of the syntax for the language from C. Eich built a simplified object model based on C structs, added patterns from SmallTalk and data and code symmetry from LISP. He also threw in an event model inspired by Hypercard. Early versions of the language also had simplified approaches to concurrency and memory management, because a typical webpage's life lasted up to a few minutes. [10]

### 1.2.1 Function as a variable

JavaScript is a language that treats functions as first-class citizens. This means that they can be declared and modified as any other variable. It is a useful method, since it allows the creation and passing around of anonymous functions. However it also allows to create and change functions at runtime - Figure 1.

```

// Define test function
var test = function(){
    alert('hello');
}

test(); // Runs test function - alerts hello

// Overwrite test function
test = function(){
    alert('goodbye');
}
test(); // Runs test function - alerts goodbye

```

Figure 1 - Changing functions at runtime in JavaScript

We will touch upon the security implications of this in later chapters.

### 1.2.2 Variable scope and global objects

Scope is a context within a program where a variable name can be used. Most higher level programming languages employ some form of scope and a lot of them force the scope to the function unless specifically overwritten. For example in PHP a variable has to be forced to be global using the `global` keyword as shown in - Example of scope in PHP Figure 2.

```

<?php
$a = 1; // Declared in the global scope

function test(){
    echo $a; // Will not print anything since
    // $a is not initialized in the function scope
}
function test2(){
    global $a; // Now you have access to global variable
    echo $a; // Prints 1
}
function test3(){
    global $b;
    $b = 2;
}
test();
test2();
test3();

echo $b; // Will print 2

?>

```

Figure 2 - Example of scope in PHP

JavaScript is also a function scoped language, however it is not a block scoped language like PHP. This means that `for` and `while` loops and `if` statements do not produce a scope in JavaScript [11]. JavaScript employs a large amount of callbacks and as such, the easy creation and modification of global variables is allowed. In JavaScript all that is needed to make a variable declaration global, is not to use the `var` keyword when declaring a variable - Figure 3.

```
var global = 1; //Create a global variable

function test(){
    var local = 2; // Create a local variable
    global = 2; // Overwrite the global variable
}

test();
alert(local == 2); // Will alert false since local does not exist in this scope
alert(global == 2); // Will alert alert true
```

Figure 3 - Example of scope in JavaScript

However this makes declaring and accessing global variables the default action, meaning that it is easy to do accidentally. This in turn can have adverse effects on the program flow. In Figure 4 an example code can be seen that shows the effect of accidentally modifying global variables in an authorization scheme.

```
accessLevel = 1; // Define base level

function canAccess(){
    accessLevel = accessLevel + 5; // Lets do some calculation with the level
    return accessLevel > 10; // Determine if we qualify
}
// Will not execute since the user is not qualified
if(canAccess()){
    alert('1');
}
// Would expect not to execute but since the global object is
// accidentally accessed then the changes accumulate and it will fire
if(canAccess()){
    alert('2');
}
```

Figure 4 - Example of authorization failure due to scope misuse

### 1.2.3 JavaScript „with“ statement

ECMAScript 3 introduced the `with` statement, which was meant to be a convenience function. It allowed making the code visually more appealing, by enabling the developer to specify which object was currently the focus and thus avoid long address chains – Figure 5.

```

one.two.three.four.five = 1;
one.two.three.four.six = 1;

with(one.two.three.four){ // Set the object to one.two.three.four
    five = 5; // Modify one.two.three.four.five
    six = 6; // Modify one.two.three.four.six
}

alert(one.two.three.four.five); // Will print 5
alert(one.two.three.four.six); // Will print 6

```

Figure 5 - Example of "with" statement usage

However the previously discussed open global scope of JavaScript makes the statement problematic. It makes it ambiguous as to what variable is actually being changed. If the variable does not exist under the specified object a global variable will be accessed instead, an example of which can be seen on Figure 6.

```

one.two.three.four.five = 1;
one.two.three.four.six = 1;

with(one.two.three.four){ // Set the object to one.two.three.four
    seven = 7; // Will create a global variable seven
}

alert(one.two.three.four.seven); // Will print undefined
alert(seven); // Will print 7

```

Figure 6 - Example of a "with" statement problem

Accidental changes into the global scope are a common problem in JavaScript because variables default to global. The effects of confusing local and global scope vary depending on the application and can affect all three security aspects of the application. However this issue most often affects the integrity of the application, because changes made into the global scope tend to accumulate over time thus modifying the server's behaviour.

It also makes the program flow hard to follow if there are some errors as to where the global scope is accessed. This problem is magnified by the fact that in JavaScript all functions are variables – meaning that the program can inadvertently delete or change functions. This is an issue that will be addressed in later chapters where its effects on the security of Node.js are discussed.

### 1.2.4 Floating point numbers in JavaScript

According to the ECMAScript standard all numbers in JavaScript are represented with double floating point numbers as specified by IEEE 754. Since some real numbers do not have a binary representation the IEEE Standard for Binary Floating-Point Arithmetic is used in the „round to nearest“ mode [12]. In most cases, especially on the client side where JavaScript is historically most often utilized it does not show any noticeable effects. This is why many people who program in JavaScript never even know of it. It also makes the results more unexpected when they do occur – an example is shown in Figure 7.

```
a = 0.1;
b = 0.2;

// Would expect this not to run since 0.3 should not be greater than 0.3
// However due to no exact representation of 0.1
// the sum will be 0.30000000000000004 which passes the validation
if(a+b > 0.3){
    alert('?');
}
```

Figure 7 - Example of floating point arithmetics in JavaScript

It shows that any application which relies on decimal math can become unstable and unpredictable unless thoroughly tested first. Dependant on its uses it can affect the confidentiality of the application with deviations in integrity and availability being less likely.

### 1.2.5 Eval and other runtime interpreter functions

JavaScript, like many other languages, allows dynamic creation of code strings, then compiling and executing them. As in various other programming languages this function is called `eval`. It is a powerful command since its uses are limited only by the language itself. However `eval` is also often misused by the developers.

The problem with `eval` comes from its most usual use case – creating and executing pieces of code based on user input. From the security standpoint it is a dangerous function because it creates many problems at the same time. The main issue is executing something constructed from user input which, without proper validation, is inherently insecure. Even with validation the attacker might find a way around it to execute arbitrary code.

JavaScript has three other functions that have the same effect as `eval` – `setTimeout`, `setInterval` and `Function`. All of these functions take in string based code and make it executable although with some variations on its executability. Figure 8 depicts the different functions that act like `eval`.

```
code = 'alert("hello");';

// Will alert hello
eval(code);

// Will alert hello after 2 seconds
setTimeout(code, 2000);

// Will alert hello every 5 seconds
setInterval(code, 5000);

// Will create a function sayHello() that will alert hello when ran
sayHello = new Function(code);
```

Figure 8 - Eval and its counterparts in JavaScript

JavaScript execution on the client side is usually sandboxed to the browser. This means that that historically `eval` has not produced a lot of security problems in JavaScript. However it became an issue when JavaScript moved from the client side to the server side, since execution of arbitrary code on the server is a major compromise of security all across the board. The bad habit of using `eval` in JavaScript is however difficult to break.

## 2 Security of a web server written on Node.js

In this chapter a web application that is realized on Node.js platform will be analysed. A look at the different parts of the working system and how they influence the overall security of the server will be taken.

### 2.1 What is Node.js?

Node.js is a wrapper written around Google's V8 JavaScript engine that allows JavaScript to be run outside the browser. It optimizes JavaScript for work outside the browser by providing it with binders, allowing the usage of various C libraries, and possibilities to manipulate binary data and access system functions, which is lacking in JavaScript. It also supplies JavaScript with request handling interpreters. This allows Node.js to be used as a server, which is also the most common usage of Node.js. [13]

Its development began in early 2009 when Ryan Dahl decided to create an evented networking server capable of asynchronous I/O operations. He chose the V8 JavaScript engine that Google had been developing for a few years because it was already optimized for running JavaScript in any UNIX environment. This made the adoption into Dahl's project easy and so Node.js was born. [9]

Node.js was first publicized on the JSConf of 2009 where Dahl introduced Node 0.1. It gained attention because it was novel and interesting and it also provided solutions for several issues that were present in modern web. Unlike other evented models that were already out there, Node.js supports the event model at the language level. This is why Node.js provides a good solution for providing real-time web services, which is a popular direction in modern web development. This coupled with the fact that developing Node.js applications essentially means writing JavaScript, which most web developers are familiar with, gave Node.js a good adoption rate.

The other reason Node.js became popular was the introduction of NPM (Node Package Manager), which gave developers an easy way to install and manage community developed modules. It is essentially a copy of the way package managers of different Linux distributions work. This provided Node.js a fast growing community codebase, which by now is nearing ten thousand packages. [14]



The Node.js project is being supported by some large players in the web development field like Google and there are also several prominent web services that are already running on servers written on the Node.js platform (LinkedIn mobile [15], Azure [16]). Node.js has achieved all of this before reaching a stable 1.0 version which is unusual and shows the demand for this type of server. So it appears that Node.js is here to stay and will gain more and more traction over time. [2]

## 2.2 Server main event loop

One of the most novel ideas that Node.js introduces is that the user is forced to write non-blocking code, since the built in functions are non-blocking. There are several other eventing systems out there, but most of them are libraries built on top of blocking platforms. Node.js supports the event model at the language level. Being asynchronous and non-blocking is in fact one of the core ideas behind the project and as such asynchronous interactions are forced upon the programmer [17]. Eventing is achieved by running a continuous loop thread that passes jobs to a thread pool and then handling callbacks as depicted on Figure 9.

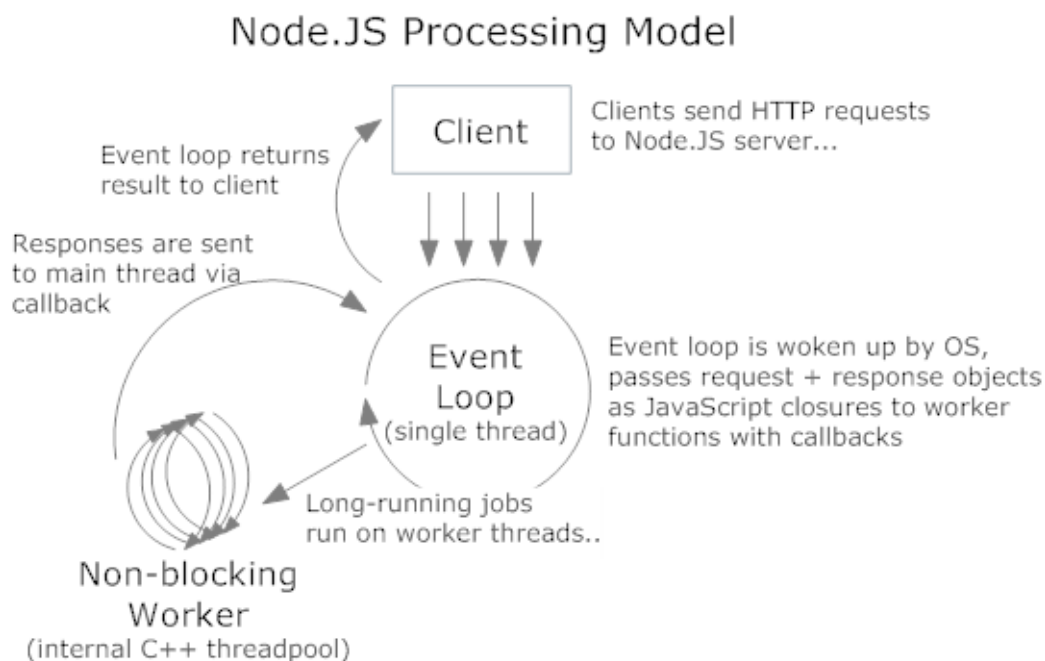


Figure 9 - Node.js processing model [18]

With this model a high concurrency rate can be achieved (unless wrongly programmed) since none of the connections actually block the event loop from answering subsequent

requests. The example web server application depicted in Figure 10 shows that the server itself is running as a non-blocking event loop where every request triggers a set callback.

```
var http = require('http'); // Require HTTP module

// Create a server and provide a callback in case of request event
http.createServer(function (req, res) {

    // Answer the request
    res.writeHead(200, {'Content-Type': 'text/plain'}); // Write the head

    // Write the end of the content after 2 seconds
    setTimeout(function() {
        res.end('world\n');
    }, 2000);

    // Write some of the content
    res.write('hello');

}).listen(1337); // Make the server listen on port 1337
```

Figure 10 - Non-blocking "Hello world" example server in Node.js [19]

It is a bit more complex example than a simple „hello world“ server, since it actually serves first „hello“, then after 2 second delay serves „world“. However during the two seconds it does not sleep and block the thread. Instead a marker is placed and after two seconds the callback is executed, so the single thread is able to answer thousands of requests without others waiting two seconds for the previous one to finish. [19]

This model is powerful in a sense that it allows for great non-blocking I/O to occur in a single thread, which makes the overhead of Node.js very small – no new threads are made. However it also provides a serious drawback – there is one main thread that houses the process. This in turn presents several security and reliability issues.

### 2.2.1 Run-time server poisoning

In most web servers every request spawns a new child process and everything happening within that process is terminated when the process ends. However since the whole Node.js server is running in a single thread then if this thread is corrupted then the behaviour of the server can be altered. These modifications will not last just for the duration of the corrupting request but for all subsequent requests.

To demonstrate this concept a corruptible server was set up. It prints out a form for the user and using an `eval` statement in the POST request handler as seen in Figure 11, returns the

sum. This provided a way to run arbitrary code on the server and demonstrate the server poisoning concept.

In the following examples we use the eval statement to create a corruptible handler, since it is the easiest and most evident. However this can be used with all of the previously addressed dynamic code generation functions or other security vulnerabilities that allow for the execution of arbitrary code. In these examples we employ the Express framework written for Node.js and exploit the fact that functions in JavaScript are actually variables. Specifications of the test environment are in Appendix 2 and full code and description can be found in Appendix 3.

```
// Show the form to user
app.get('/sum',function(req,res){
  res.send('<form method="POST">'+
    '<input name="first" />'+
    '<input name="second" />'+
    '<input type="submit" value="submit" /></form>');
});
// Process the form
app.post('/sum',function(req,res){
  // Sum the variables from POST request
  var sum = eval(req.body.first +'+'+req.body.second);
  // Send result to user
  res.send('the answer is '+sum);
});
```

Figure 11 - Example of a POST request handler using "eval" that can be corrupted

Upon sending a request to the server as depicted in Figure 12, the statement gets executed. This piece of code adds a path to the possible routes named */myurl*. When visiting that path a response of „corrupted“ is sent. This is one of the more benign ways of altering the server behaviour since adding another path to the list of paths is all that is done.

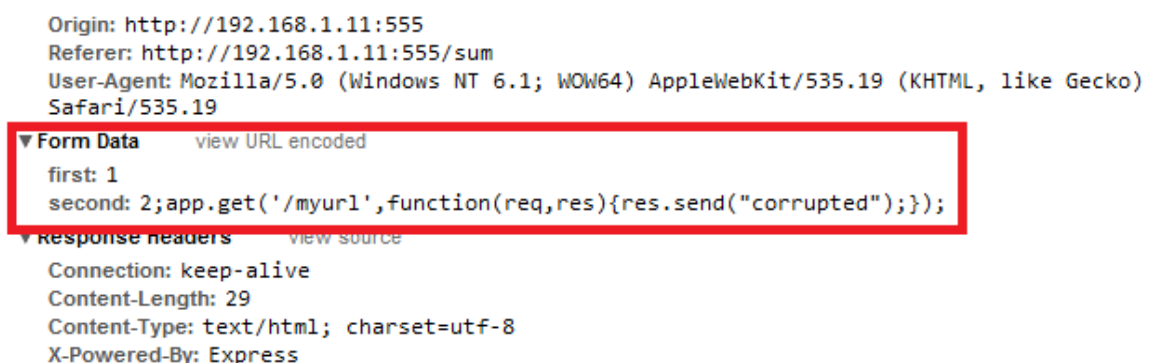


Figure 12 - Example excerpt of a corrupting request adding a path to the server

The following demonstration shows something more sinister that becomes possible because we can alter functions at run-time. After some exploration the POST request handler was found to be in `app.routes.routes.post[0].callbacks[0]` and sending the request depicted on Figure 13 actually changes how the POST request is handled.



Figure 13 - Example excerpt of a corrupting request changing the server function

Trying to sum anything after this will result in an answer of „1337“ instead. This demonstration shows the potential that this kind of attack has. Another handler that sends all the request information to the attacker could be added to the request. Or a more subtle change to how the request is handled could be implemented modifying the results slightly. The possible uses are limited by the attacker’s imagination and the intended use of the application.

There are several factors about this type of attack that make it scary:

1. It is stealthy
  - a. There are no log records generated except maybe of the original request.
  - b. If executed right then there will be no noticeable changes to the server.
2. It goes past HTTPS – since the attack is targeted at the server itself then no matter how secure the transaction to the server the information is still accessible.
3. It hard to protect against – In PHP for example the attacker would have to rewrite server files to achieve this behaviour and usually the server process is not allowed to modify those files. In Node.js or JavaScript there is no generic security restriction that would apply.

It allows the attacker to affect all three pillars of security. Confidentiality and Integrity are probably the main targets with the attacker changing the server behaviour to suite his agenda. Availability – although a possible target is a very crude use of this and as such

unlikely. Most likely attack on availability through server poisoning would be to reconfigure the server to limit availability to specific users.

### 2.2.2 Denial of Service

Denial of Service (DoS) attack is one of the simplest forms of network attacks. Instead of trying to steal or modify information the aim of this attack is to prevent access to the service or resource. This is usually achieved by flooding the server with requests which it is then unable to handle and will become unresponsive and may crash. [13]

Due to its architecture Node.js uses less system resources to handle a request and as such is resilient to conventional Denial of Service attacks such as bombarding the server with tons of requests. It is more likely to run into bandwidth problems then problems handling the request. Benchmarking tests were ran in order to test a basic Node.js server against request bombardment with the server shown in Figure 10.

The results were that ApacheBench [20] ran into problems with creating more concurrent requests before Node.js showed any signs of overloading. Results, shown in Figure 14 and Figure 15, show that the server can respond to the requests fast and it only uses a fraction of the resources that are available in the VM.

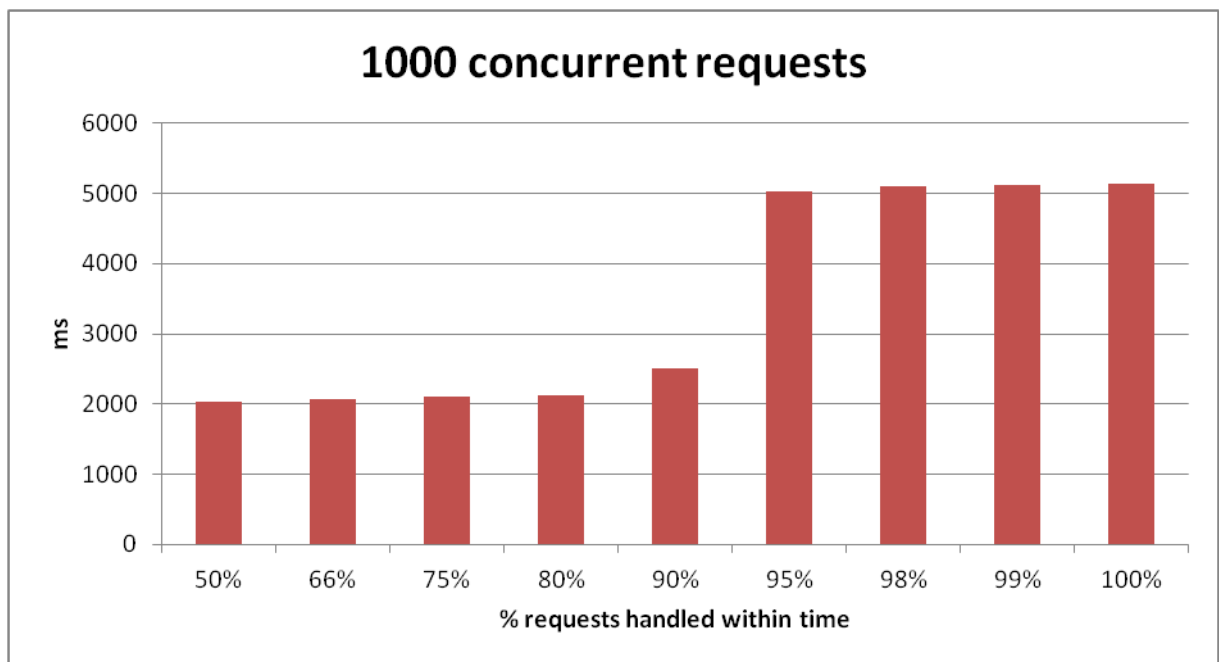


Figure 14 - 1000 concurrent requests on a "hello world" server

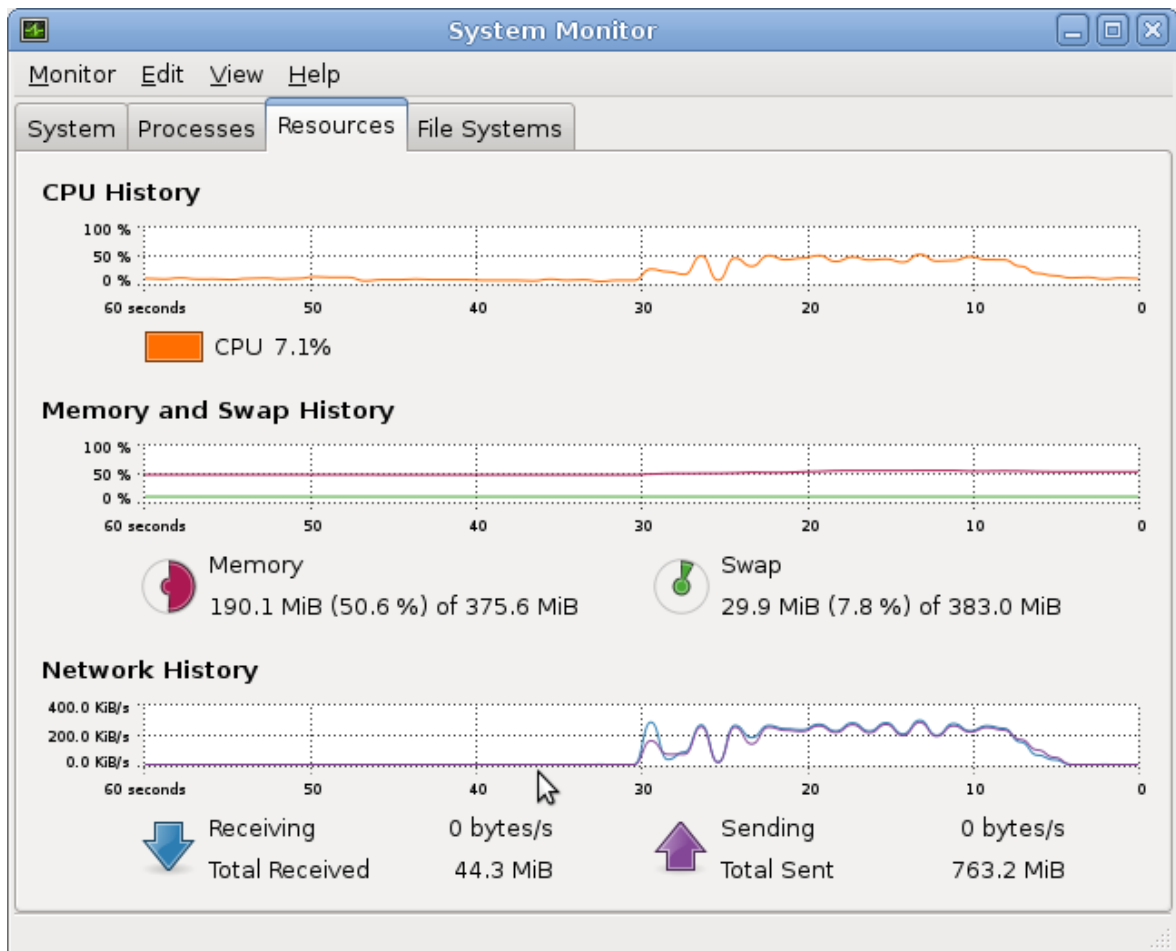


Figure 15 - System footprint on 1000 concurrent request test

Request flooding is one way to achieve Denial of Service. It can also be achieved by using flaws in the system or other methods to make the server unresponsive. This is where the single thread event loop starts to work against servers written on Node.js affecting the availability of web services written on it.

#### 2.2.2.1 Time consuming actions

As the previous test shows - Node.js has no problem handling thousands of requests on a single thread, however when the application is programmed incorrectly then it becomes problematic to even handle a few. A test example was constructed, shown in Figure 16, for calculating the nth Fibonacci number recursively. A recursive function was used because it is slow and allows the demonstration of the problems involved with doing intensive calculation.

```
// Define the route
app.get('/fibonacci/:num',function(req,res){
    var num = fibonacci(req.params.num);
    res.send('Fib is '+num);
});
// Function for calculating Fibonacci number recursively
function fibonacci(num){
    if(num<3) return 1;

    return fibonacci(num-1)+fibonacci(num-2);
}
```

Figure 16 - Example of a time-consuming process: Fibonacci calculation

Testing showed that calculating 40th Fibonacci number took about 5 seconds, which is quite fast compared against for example PHP. However, when more traffic was directed to the URL then the limitations of a single thread became evident. In threaded servers requests are handled in separate threads, so many of those can run in parallel when the hardware supports it. Node.js handles requests in a single main thread and as shown in Figure 17 it slows down the requests considerably.

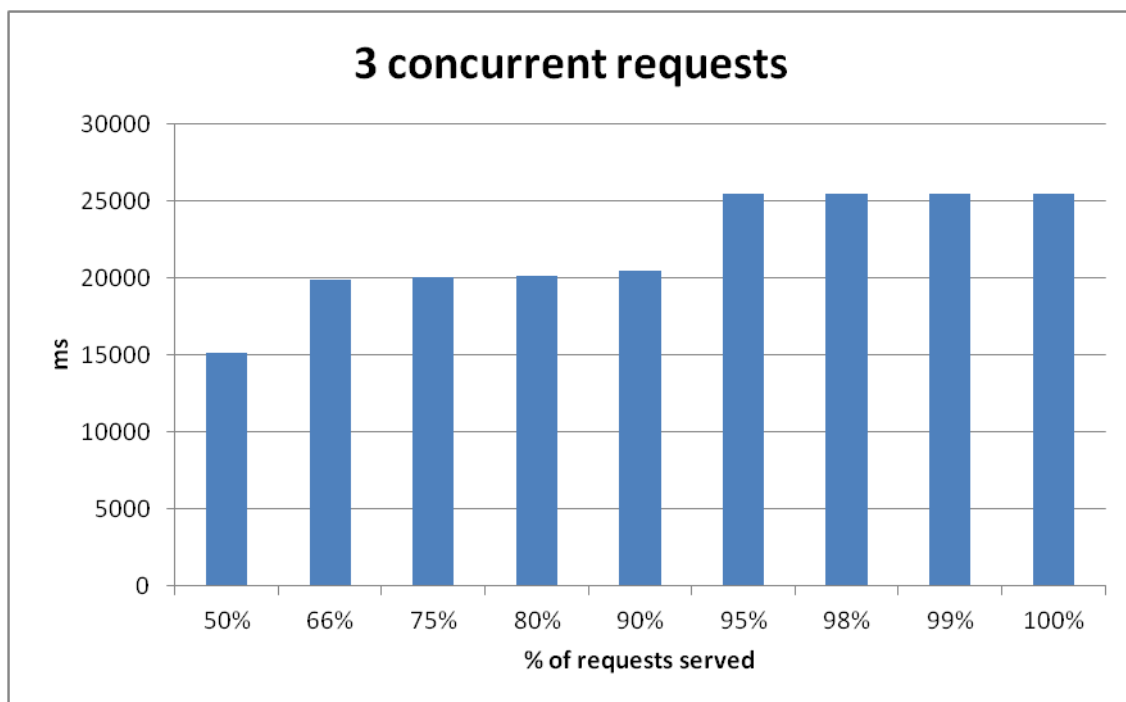


Figure 17 - Calculating 40th Fibonacci number with 3 concurrent requests

Not only do the calculations have to wait for the previous ones to finish, but all requests will be left hanging until a calculation is complete. Only then will another request be accepted by the server into the queue.

It shows that the developer is required to think a little differently when trying to create an application that does intensive calculations. Separation and queuing does not come naturally to Node.js in such cases and has to be brought in by the programmer.

#### 2.2.2.2 *Slowloris*

Slowloris [21] is an attack method that was conceived in 2007 by Adrian Ilarion Ciobanu. He suggested that taking down a server does not require flooding the server with requests to make it unresponsive – it is enough to open up lots of connections to the server and refuse to read the responses.

The attack is done by sending partial HTTP requests and then sending subsequent headers to keep the connection open. At some point the server reaches its limit of concurrent connections and the rest are rejected. This allows to make the server unresponsive with minimal bandwidth and without affecting other services and ports. [21]

This attack is mostly aimed at threaded servers like Apache HTTP and GoAhead, because they have a thread limit and thus start rejecting connections at certain point. Evented servers like Nginx [22] and lighttpd [23] are considered to be unaffected by it.

A test was set up to serve a simple index page using the Express framework [24] and Jade template engine [25]. The request handler can be seen in Figure 18 and all it does is use Jade template engine to render an index file – the contents of which was a string „Say lorem ipsum“.

```
// Define the index handler
app.get('/', function(req,res){
  // Render the index.jade file without a layout
  res.render('index',{layout:false});
});
```

Figure 18 - Route handler rendering index file with Jade template

Load tests were ran with ApacheBench doing 10000 requests with concurrency 1000. The results can be seen in Figure 19, where it is evident that most of the requests are being served under 4 seconds and a few requests were held up for almost 10 seconds. This is a good result that shows that in general the server can handle a high load. In Figure 20 it is shown that only a small spike on the CPU load and network utilization history charts is created by the test.



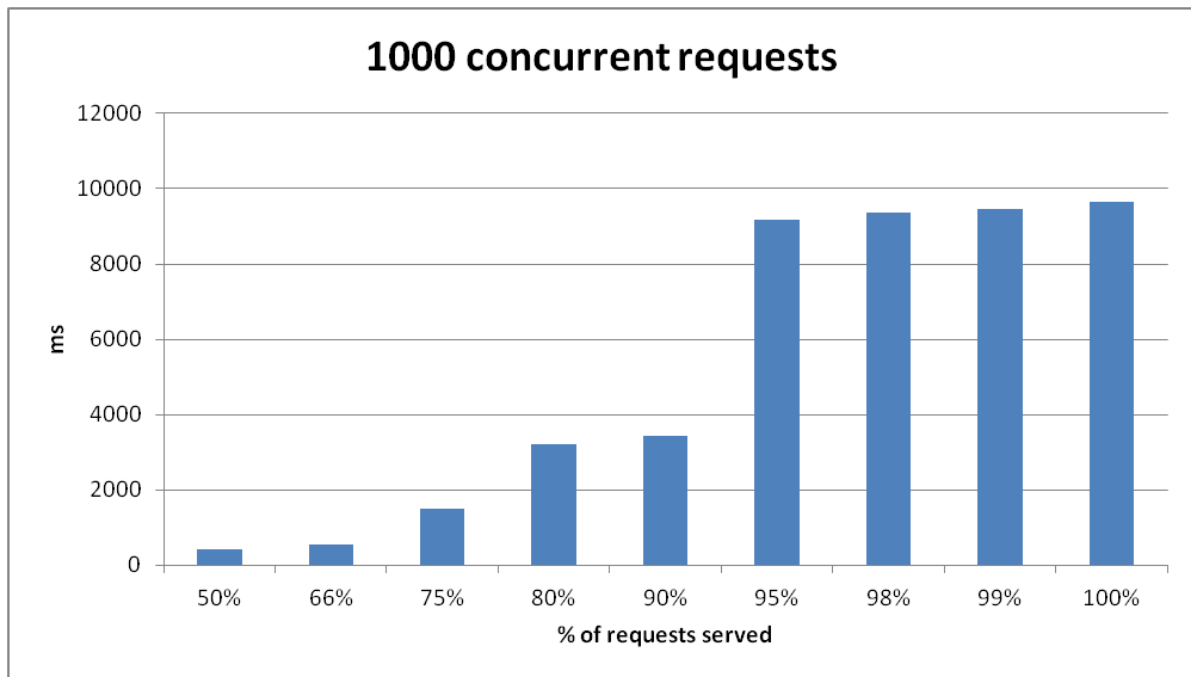


Figure 19 - Load test with 1000 concurrent connections rendering index file

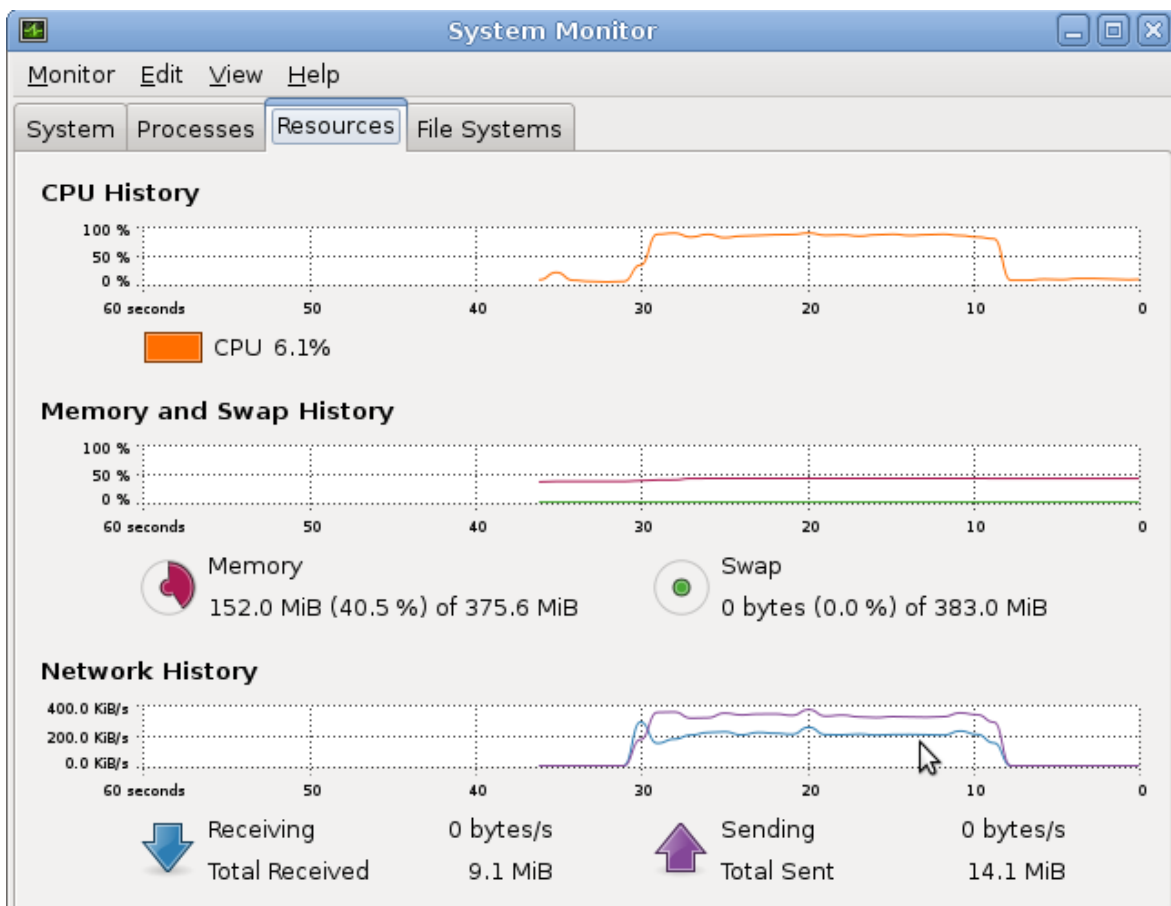


Figure 20 - System load with 1000 concurrent request rendering index from file

When attacking with Slowloris, then by increasing the number of open connections enough, a number can be reached, where the server will crash upon another request. The point where the server crashes is illustrated in Figure 21. Now what is interesting about this is that the server actually does not become unresponsive or will not reach its limit as to open connections. What happens is that the file descriptor limit is reached since a file is opened to render the answer and upon a subsequent request the main loop thread is terminated and because of it the whole server goes down.

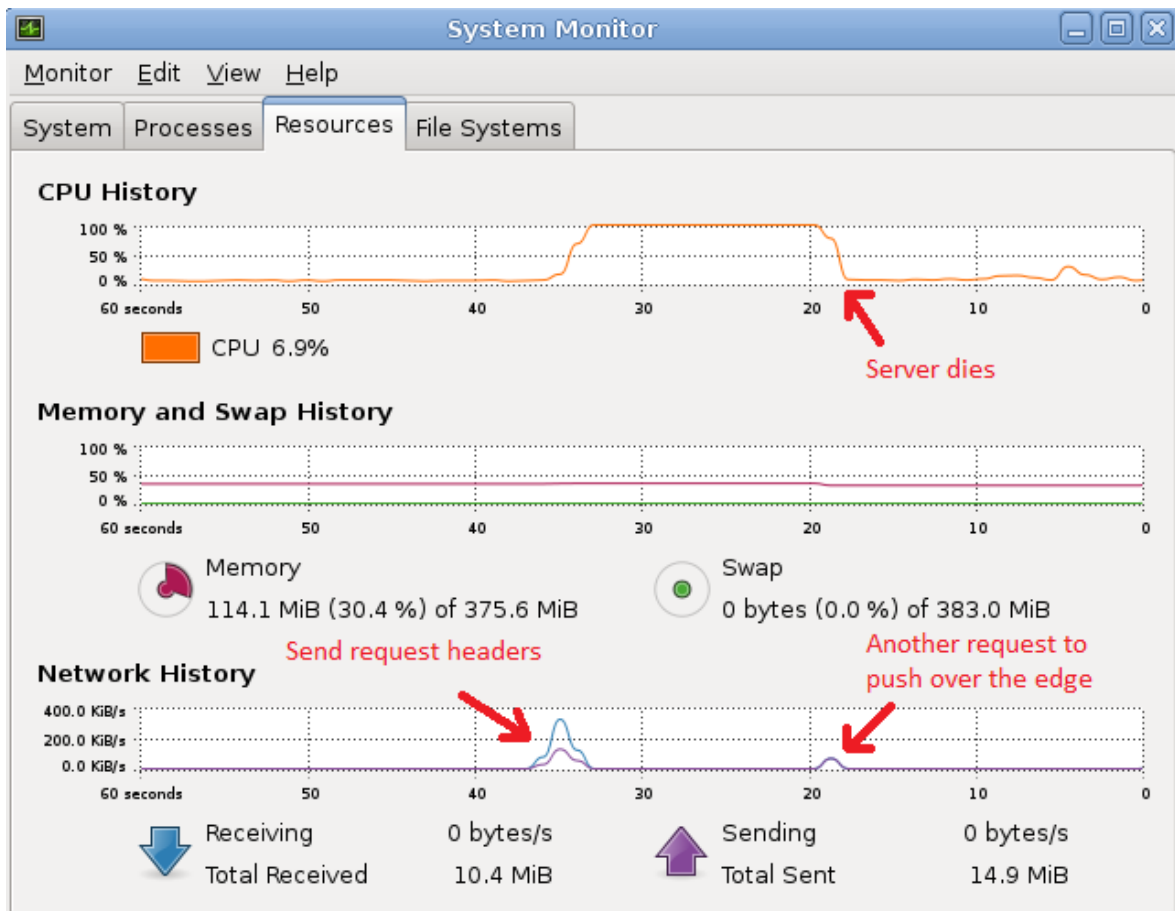


Figure 21 - Annotated illustration of load produced by Slowloris attack

This shows that if the server uses rendering from files, then it is still vulnerable to Slowloris, because it enables an attacker to tie up all the file descriptors. It means the server, more resilient than treaded servers like Apache for example, is still vulnerable.

## 2.3 Server configuration

One of the selling points that is often emphasized about Node.js is that it is lightweight compared to leading web servers. When building an application in Node.js it does exactly what it is told. It does not have a default configuration to serve some files or execute

others. This means that when a server as shown in Figure 22 is created, it will not do anything besides answer „Hello world“ on port 1337.

```
var http = require('http'); // Require HTTP module

// Create a server and provide a callback in case of request event
http.createServer(function (req, res) {

    // Here the request is answered
    res.writeHead(200, {'Content-Type': 'text/plain'}); // Write the head
    res.end('Hello World\n'); // Write the content

}).listen(1337); // Make the server listen on port 1337
```

Figure 22 - Example "Hello world" server

This means that Node.js servers, unlike other servers, have no default configuration that would provide unwanted behaviour of the server. It is a useful feature from the security side in the sense that the server will not perform actions not specifically programmed. This forces developers to actually think about all the different things they want their application to perform and thus increases the security. For example when a developer wants to serve files to the user then the server has to be specially programmed to do so. However the drawback is that some essential things have also been left to the hands of the user which are conventionally handled by the server.

For example – no requests or activities will be logged unless the developer specifically tells the server to do so. Since many developers do not think about such things it will make tracking erroneous events more difficult.

### 2.3.1 Error handling in Node.js

Error handling is one of those things that has been left completely to the hands of the user. More advanced Node.js libraries usually provide an optional error callback when invoking functions, however being optional makes it easy to overlook. Coming down to the basic levels there are currently no error handlers present. The programmer must be ever vigilant to avoid errors in the work process since this coupled with the fact that the main loop runs in a single thread makes the entire server fragile. Any unhandled error will basically shut down the server and all subsequent requests will fail.

A test example featuring a request handler for AJAX requests is shown in Figure 23. Now all the attacker has to do is send a request to that URL without JSON. That will result in an

error being thrown by the server and the process exiting, which essentially means that the server is down.

```
// Process request
app.post('/json',function(req,res){
  // Parse JSON to an object
  var o = JSON.parse(req.body.o);
  // Do something with o...
  o.ret.result = o.a + o.b;
  // Send the result to user
  res.send('the answer is '+o.ret.result);
});
```

Figure 23 - Example of a JSON request handler without error catching

This makes it easy to perform Denial of Service attacks against the server, since all the attacker has to do is induce a single unhandled error to take down the entire service, which is a major problem to the availability of the service.

### 2.3.2 Node.js memory usage

Usually the memory usage of a server process is limited by the server configuration by default. Node.js however has a main thread and as such it is difficult to limit how much memory a clients request can take, since setting a memory limit will affect the whole server not a single request. This leaves the server open to DOS attacks that rely on hogging the memory.

The common way of handling regular POST requests takes the request into memory and then processes it. Many examples can be found online and in Figure 24 an excerpt from the widely used Connect [26] middleware is shown.

```

/**
 * Parse application/x-www-form-urlencoded.
 */

exports.parse['application/x-www-form-urlencoded'] = function(req, options, fn){
  var buf = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ buf += chunk; });
  req.on('end', function(){
    try {
      req.body = buf.length
        ? qs.parse(buf)
        : {};
      fn();
    } catch (err){
      fn(err);
    }
  });
};

```



Figure 24 - POST request handler without limit

The request size is also not limited by Node.js which means that a large POST request can be sent to fill the whole memory. In Figure 25 an example code for creating a large POST request is shown.

```

// Dependencies
var http = require('http'),
    querystring = require('querystring');

// Specify request options
var options = {
  'host': '192.168.1.14',
  'port': '555',
  'path': '/memory',
  'method': 'POST',
  'headers': {
    'Content-type': 'application/x-www-form-urlencoded'
  }
};

// Create request
var post_req = http.request(options, function(res) {});

// Write a large request
for(i=0; i<2500000; i++){
  var arr = {};
  arr['file'+i] = 'hello'+i;
  post_req.write(querystring.stringify(arr));
}

// End the request
post_req.end();

```

Figure 25 - Script creating a large POST request

Figure 26 shows the servers attempt to handle a large POST request generated with the previous code. All the memory and swap is filled up and then the server becomes unresponsive. Even when the attack script is terminated the server will stay unresponsive for a long time. Depending on the server configuration the Node.js process might be terminated instead.

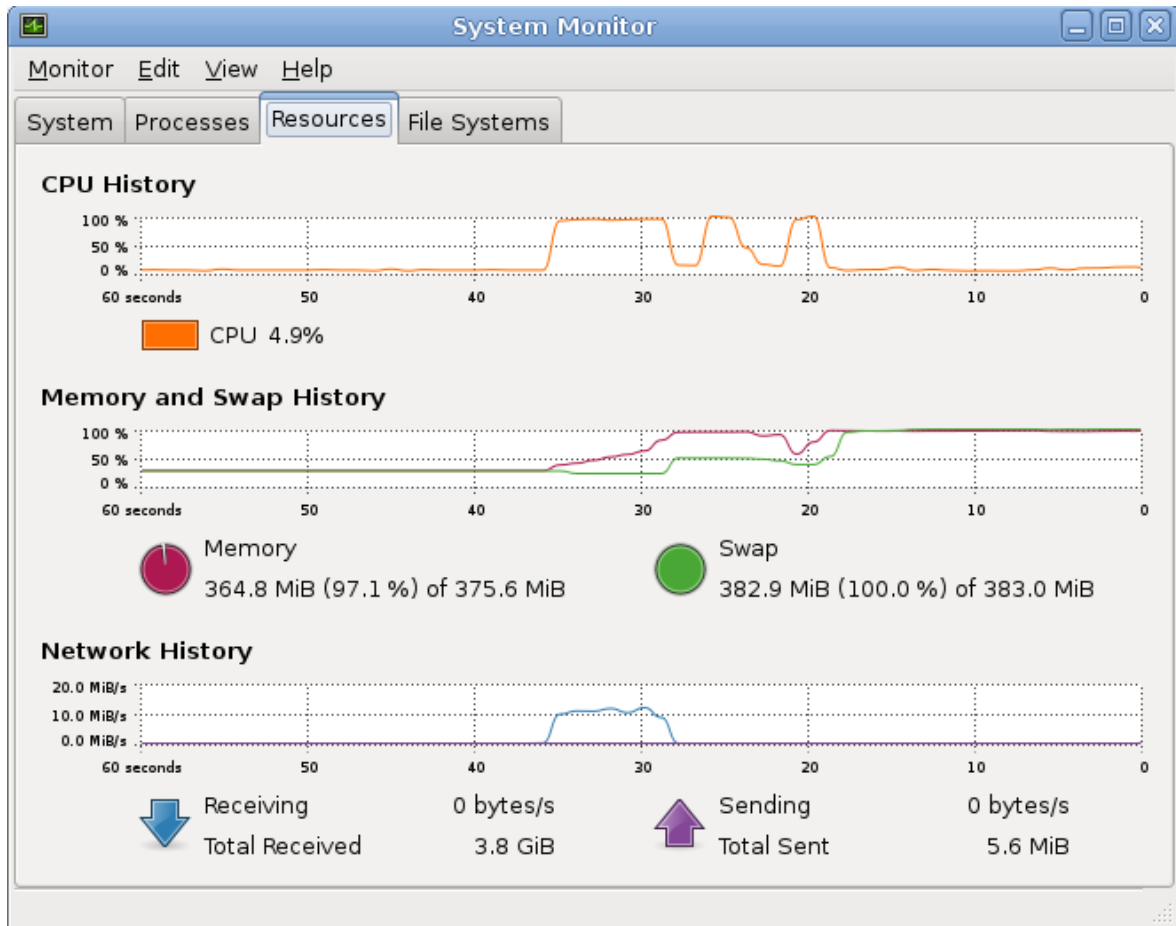


Figure 26 - Servers attempt to handle a large POST request

This attack can be performed on all web services that employ the popular Node.js framework Express, or other frameworks built on Connect, and have a POST request handler somewhere. It is simple attack against the availability of the service that is effective and easy to perform.

## 2.4 Rapid changes in Node.js

Node.js is a relatively new piece of technology that is just reaching 3 years in development. During this time it has gathered an active community and developer base and some large supporters for the project. This coupled with the fact that Node has yet to reach

1.0 milestone means that it is still in rapid development, which makes it an unstable platform for an application. Since reaching version 0.6 in autumn of 2011 it has released a new version on average every 11 days [9]. This produces a whole set of problems as to the security of the web application.

#### **2.4.1 Staying behind the Node.js development**

One way to develop in such a rapidly evolving environment is to freeze the version of Node.js that is in use. This will provide stability to the core and modules in use in the upcoming web application.

On the other hand it produces a problem since one of the main mantras of cyber security is keeping systems up to date, which is now quite difficult without a lot of work making sure everything works with the new version. However staying behind will leave systems vulnerable to all the possible errors that have been discovered, fixed and published in the mean time.

##### **2.4.1.1 Hash table collision**

During the 28C3 (28th Chaos Communication Congress: Behind Enemy Lines) Alexander ‘alech’ Klink and Julian | zeri presented a general attack against web services using hash table collision.

When creating an array in JavaScript for storing, e.g., HTTP request parameters, a hash table is usually created to store the key-value pairs. Hash tables in general are very efficient for inserting (see Figure 27) and looking up values. The best case time complexity of insert and lookup operations in a hash table is constant. However, if there are a lot of hash collisions in the table then operations with that hash table degrade into linear search. Hence, adding  $n$  elements to a hash table has the worst case time complexity of  $O(n^2)$  as illustrated in Figure 28. [27]

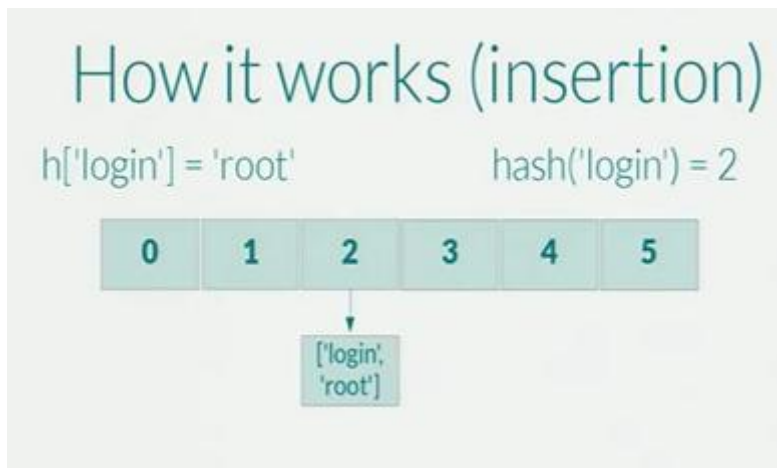


Figure 27 - Hash table working principle [27]

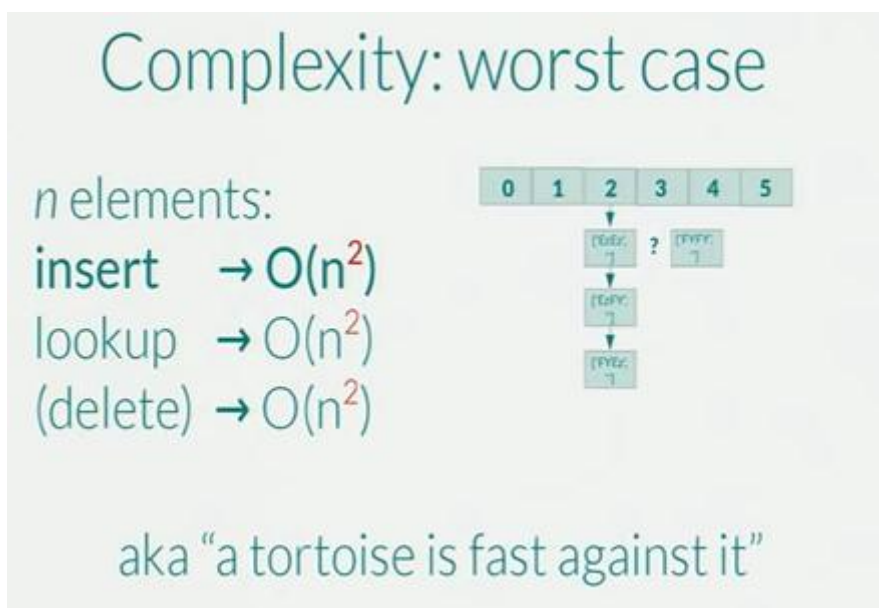


Figure 28 - Hash table worst case complexity [27]

Most hash functions used to create hashes for hash tables in different languages are a variation of DJBX33A function shown in Figure 29. It is not a complex function, allowing the calculation of different strings with colliding hashes. This in turn enables the construction of a HTTP request containing thousands of parameter names with colliding hash values. [27]





Figure 29 - DJBX33A hash function [27]

Sending such a request to a web application will essentially perform a Denial of Service attack, since most web servers automatically populate the POST and GET arrays. With a 2MB payload one 1GHz CPU could be kept busy for 40 seconds. This also means that most web servers are vulnerable to this attack regardless of the actual application. [27]

This problem has been known since at least 2003 when it was noticed in Perl. It influenced Perl and CRuby 1.9 to change their hash functions. It took another 8 years for this problem to reach a wider audience and gain attention.

The solution to this problem as offered by Alexander and Julian and already used by Perl is to randomize the hash function that is used when creating a hash table. This way the calculation of colliding hashes is not possible and the attack cannot be performed efficiently.

Node.js's rapid development has helped mitigate this issue fast. The underlying V8 engine fixed that vulnerability on the 5th of January 2012 and it made it into Node.js version 0.6.8 on the 19th of January, however all versions prior to 0.6.8 are still vulnerable to this type of Denial of Service attack.

### 2.4.2 Node.js modules

Node.js, being a lightweight C++ wrapper for V8, is a basic platform upon which to build applications. It comes bundled with some libraries for networking and file system operations, but everything else required to build a web application is missing.

This is where the Node Package Manager came into play. NPM is a simple package manager developed in parallel with Node.js by Isaac Z. Schlueter. While it is not officially affiliated or endorsed by the Node.js project it is bundled with the latest versions. It is designed similarly to the various Linux distributions package managers and allows users to install and modify packages using similar commands as in *apt* or *yum*. [28]

JavaScript has no native support for modules specified by ECMA standards, but in January 2009 an initiative named CommonJS began working on this problem among others. Their project gained a support from the community and they developed a module system for JavaScript. This system was adopted into Node.js allowing developers to group their code into more manageable modules. [29]

By providing the community with a simple to use tool for building and updating libraries the Node.js module network gained traction and by now there are almost 10000 entries in the NPM. Those modules vary greatly in their functionality, usability and quality. Most of those have been developed by a lone programmer who decided to make a solution to some problem public in the form of a module. This allows developers to reuse each other's code and speed up the development process.

A great concern however is the lack of a vetting process in adding new packages to the repository. Unfortunately this means that many packages that have been added are lacking in both code quality and support. It also means that any package on the list could either accidentally or maliciously contain security vulnerabilities and should therefore be thoroughly checked before active deployment. [28]

Using open source libraries and modules in developing a web application is a common approach, not just with Node.js, but also in other platforms and languages. However in Node.js many modules need to be used in an average project in order to make it viable. For example making a web application which uses MySQL as its database and choosing the popular setup of Jade and Express then the module list might look the following:

- express – Web framework
  - connect
    - qs
    - mime
    - formidable
- Jade – template engine
  - commander
  - mkdirp
- Sequelize – ORM for MySQL
  - mysql
    - hashish
      - traverse
  - underscore
  - underscore.string
  - lingo
  - validator
  - moment
  - commander
  - generic-pool

This setup is already using 18 different modules, which all have different development teams and as such vary greatly in quality and activity. This can cause further problems in an already volatile environment such as Node.js, since any significant updates to the core would mean that all of these modules need to be updated separately, which might take a long time. This ensures slow adoption of newer versions of Node.js simply because of the number of programmers that have to work on the different modules.

In the previous example those 18 modules were not installed separately, but some of those were included as dependencies. This again shows the similarities between the NPM and other package managers. The main difference is that the packages in NPM are in not validated. They are simply submitted to NPM using a JSON notation as seen on Figure 30.

```

{
  "name": "jade",
  "description": "Jade template engine",
  "version": "0.26.0",
  "author": "TJ Holowaychuk <tj@vision-media.ca>",
  "repository": "git://github.com/visionmedia/jade",
  "main": "./index.js",
  "bin": { "jade": "./bin/jade" },
  "man": "./jade.1",
  "dependencies": {
    "commander": "0.5.2",
    "mkdirp": "0.3.0"
  },
  "devDependencies": {
    "mocha": "*",
    "markdown": "*",
    "stylus": "*",
    "uubench": "*",
    "should": "*",
    "less": "*",
    "uglify-js": "*"
  },
  "component": {
    "scripts": {
      "jade": "runtime.js"
    }
  },
  "scripts": { "prepublish" : "npm prune" },
  "engines": { "node": ">= 0.1.98" }
}

```

Figure 30 - Node Package Manager package description in JSON

There are two important things to note about this system, which are both highlighted on Figure 31 - dependencies and scripts.

```

{
  "name": "jade",
  "description": "Jade template engine",
  "version": "0.26.0",
  "author": "TJ Holowaychuk <tj@vision-media.ca>",
  "repository": "git://github.com/visionmedia/jade",
  "main": "./index.js",
  "bin": { "jade": "./bin/jade" },
  "man": "./jade.1",
  "dependencies": {
    "commander": "0.5.2",
    "mkdirp": "0.3.0"
  },
  "devDependencies": {
    "mocha": "*",
    "markdown": "*",
    "stylus": "*",
    "uubench": "*",
    "should": "*",
    "less": "*",
    "uglify-js": "*"
  },
  "component": {
    "scripts": {
      "jade": "runtime.js"
    }
  },
  "scripts": { "prepublish" : "npm prune" },
  "engines": { "node": ">= 0.1.98" }
}

```

Figure 31 - Important parts of Node Package Manager package description in JSON

### 2.4.2.1 Dependencies

The dependencies property allows the owner of the package to define other packages it relies on which will be automatically installed alongside the original package. This is how in the previous example the 3 main modules that were selected expanded to 18 different modules that were actually installed. Such behaviour coupled with the number of packages presents a challenge to the security of the application, since all of those modules have to be checked for vulnerabilities.

A quick search within the modules installed in the example application reveals that there is at least one vulnerability in the *validator* module which is a dependency for *Sequelize*. This module is meant for checking a variable for different types or properties mostly through the use of regular expressions. The regular expression, seen on Figure 32, for checking the validity of a URL is long and complex.

```
(!this.str.match(/^((?:ht|f)tp(?:s?)\:\/\/|~|\/)?(?:\w+:\w+@)?((?:[-\w\d{1-3}]+\.)+)?(?:com|org|net|gov|mil|biz|info|mobi|name|aero|jobs|edu|co\.uk|ac\.uk|it|fr|tv|museum|asia|local|travel|[a-z]{2}))(?!(\b25[0-5]\b|\b2[0-4][0-9]\b|\b[0-1]?[0-9]?[0-9]\b)(\.(?!(\b25[0-5]\b|\b2[0-4][0-9]\b|\b[0-1]?[0-9]?[0-9]\b)){3}))(?:[\d]{1,5})?(?:((?:\/(?:[-\w~!$+|.,*:=]|%[a-f\d]{2})+)+|\/)+|\\)?(?:\#(?:\?[\w~!$+|.,*:=]|%[a-f\d]{2})+=?(?:[-\w~!$+|.,*:=]|%[a-f\d]{2})*(?:&(?:[-\w~!$+|.,*:=]|%[a-f\d]{2})+=?(?:[-\w~!$+|.,*:=]|%[a-f\d]{2})*)*(?:#(?:[-\w~!$ |\/.,*:=]|%[a-f\d]{2})*)?$/i) || this.str.length > 2083)
```

Figure 32 - Regular expression checking the validity of a URL in validator module

This expression has lots of grouping and back references – uses of previously captured groups, which in a Nondeterministic Finite Automaton engines such as V8’s regular expression engine can cause performance issues. It becomes an issue when „match 0 or more“ or „match 1 or more“ groups are referred to with „match 0 or more“ or „match 1 or more number“ signs as seen on Figure 33. In such cases then the complexity will rise exponentially with the length of the string if the string is not valid. [30]

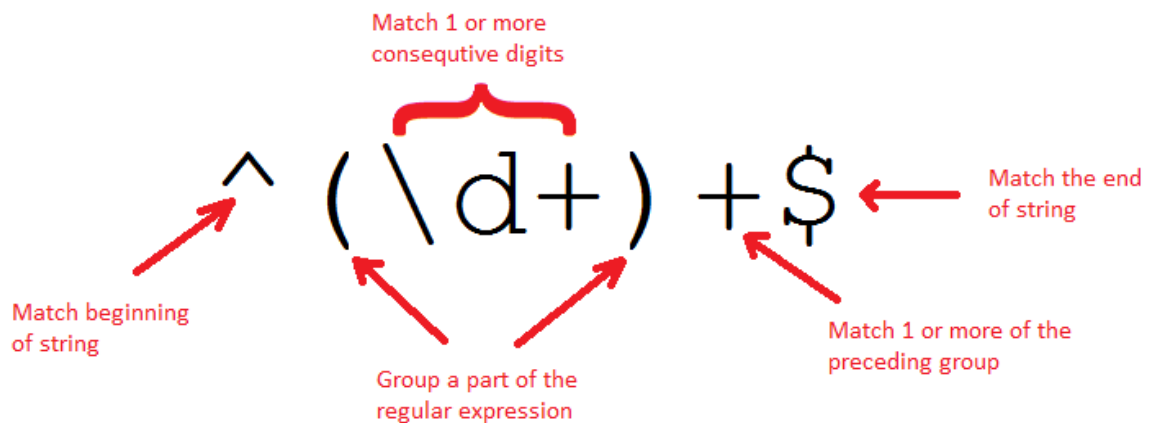


Figure 33 - Bad regular expression analysis

In Figure 34 the part of the regular expression that is similar to the previously shown pattern is underlined.



```

{
  "name": "malicious",
  "version": "0.1.0",
  "dependencies": {},
  // Run the script file before installation
  "scripts" : { "preinstall" : "node script.js" },
  "engines": { "node": ">= 0.1" }
}

```

Figure 35 - Malicious Node.js package JSON

```

// Require filesystem library
var fs = require('fs');

// Write a file in the root users directory
fs.writeFile("/root/test", "Malicious", function(err) {
  fs.unlink("script.js"); // Delete this script
});

```

Figure 36 - Malicious payload attached to test module

The test showed that no indication was given to the user about the script writing into privileged directory, which shows that in principle such a script could do anything with the system – create a backdoor, infect it or whatever the attacker wishes to accomplish.

This proves that those scripts are potentially very dangerous and coupled with the fact that arbitrary modules can be listed as dependencies the attacker can obfuscate the trail by hiding such a script somewhere in the dependency tree.

These attacks will most likely target the integrity and confidentiality factors of the service, since they occur prior to deployment or during updates.



### 3 Recommendations for improving application security

As it has been discussed and shown in previous chapters Node is still quite young and has many different issues coming both from the JavaScript language, Node.js application architecture or the NPM community. The minimalistic zero-configuration approach that the server is currently moving with means that the developer is left alone to tackle those problems. This means that without some planning and setting up standards for the development, writing a secure production ready web service is difficult. The most critical part to address is the availability of the service.

In this chapter some guidelines to help develop a secure application are presented.

#### 3.1 Know what to use it for

As with all servers, services, libraries and tools, they are usually intended for a specific purpose at which they excel at and this is also true for Node.js. Node.js is no magic server that does everything better than conventional servers. It was designed to do scalable non-blocking networking and that is where it is best. It is therefore best utilized as a simple web server or a network server.

##### 3.1.1 Calculations

Node.js was not designed to do heavy or very precise calculations. Realizing algorithms that require a great degree of precision are best implemented in languages other than JavaScript due to the previously discussed floating point number problem. When dealing with calculations using decimals or other numbers that cannot be expressed with floating point numbers, the problem does present itself.

When the required precision is not in the 10th decimal place then it is advisable to simply round calculation results before usage. Another possibility is to use some JavaScript library designed to work around floating point limitations. An example would be BigNumber [32], which transforms numbers into arrays of single digits and performs calculations on those. This method will introduce a performance loss, but it will retain the precision.

Implementing heavy calculations or long intense processing in a Node.js server should not be done in the main thread. Due to Node.js's single event loop this will clog the server and

adversely affect the availability of the service. This is why long calculations or processes in the main thread must be avoided.

When long processing or heavy calculations cannot be avoided by the logic of the web service then they should be separated into another thread. A generic way this can be achieved is by calling „`require('child_process').exec`“ function which allows running a command line command and will return the results that would have been printed to the terminal. This will not make the calculation faster on a single core machine, however it will ensure that other requests will not be left hanging because some calculation is taking place.

This can be demonstrated this by rewriting the Fibonacci calculation application discussed previously. The new request handler is shown in Figure 37 and the calculation function itself is in Figure 38.

```
// Routes
// Define the route
app.get('/fibonacci/:num',function(req,res){
  // Execute the separate script as a new thread
  cExec('node calcFibo.js '+req.params.num,{cwd:'/home/karl/masters'},function(error, stdout, stderr){
    // Send the result of the script
    res.send('Fib is ' +stdout);
  });
});
```

Figure 37 - Request handler invoking a Fibonacci calculation through the command line

```
// Get command line arguments
var args = process.argv.splice(2);

// Function for calculating Fibonacci number recursively
function fibonacci(num) {
  if(num<3) return 1;

  return fibonacci(num-1)+fibonacci(num-2);
}

// Print the result of the function
console.log(fibonacci(args[0]));
```

Figure 38 - Separate script for the calculation of Fibonacci numbers

The calculation itself takes around 5 seconds to complete then 3 concurrent calculations will take around 15 seconds because the CPU is now divided between the 3 processes. However new requests will not be left waiting for the previous calculations to finish and the server will be responsive during the calculation. A request on the same server that requires no intensive calculation will still receive a quick answer. It is important to note

that background processes should be limited otherwise the attacker could overwhelm the server with calculation requests.

A test of 20 requests, with 3 concurrent requests, was ran on the newly written code and the results are shown in Figure 39. Comparing these results to the ones on Figure 17 show that the result time has normalized and is around 15 seconds, which is exactly what was expected.

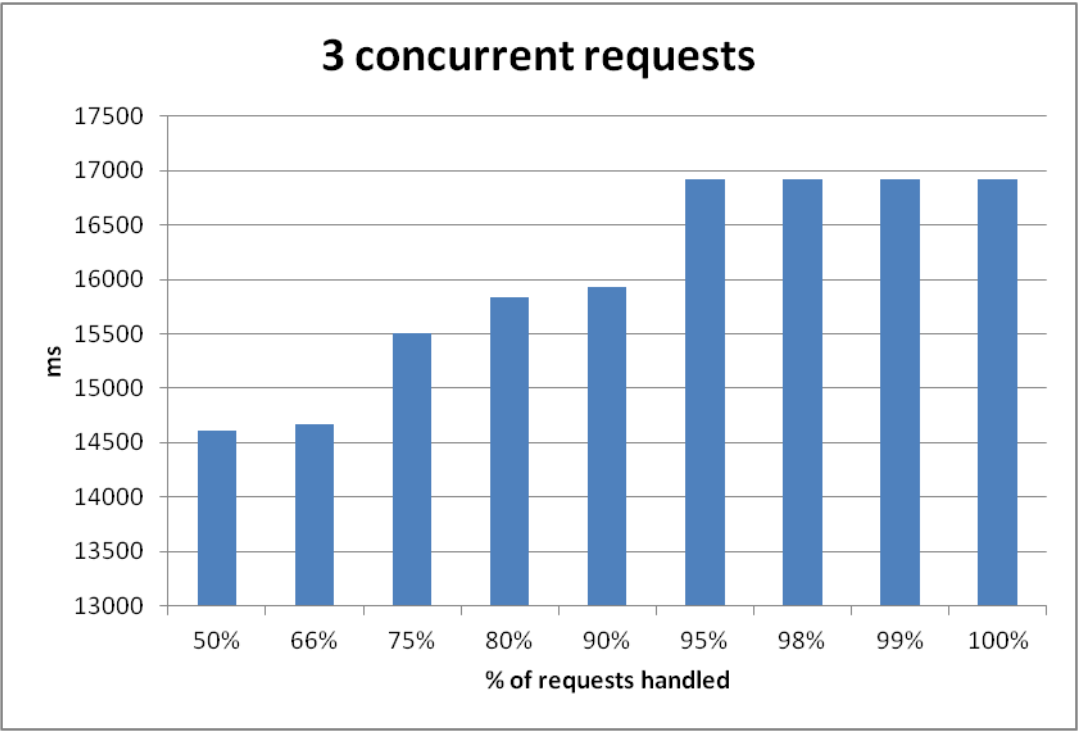


Figure 39 - 3 concurrent requests to calculate Fibonacci numbers in a separate thread

Another test was ran to demonstrate the speed of request handling at the same time as the Fibonacci number calculations are taking place from the previous test. The server was loaded with 3 concurrent requests to do Fibonacci calculations and at the same time a 100 concurrent requests were made to a route that printed the server's memory usage. As shown in Figure 40 – the requests were handled fast despite the background calculations.

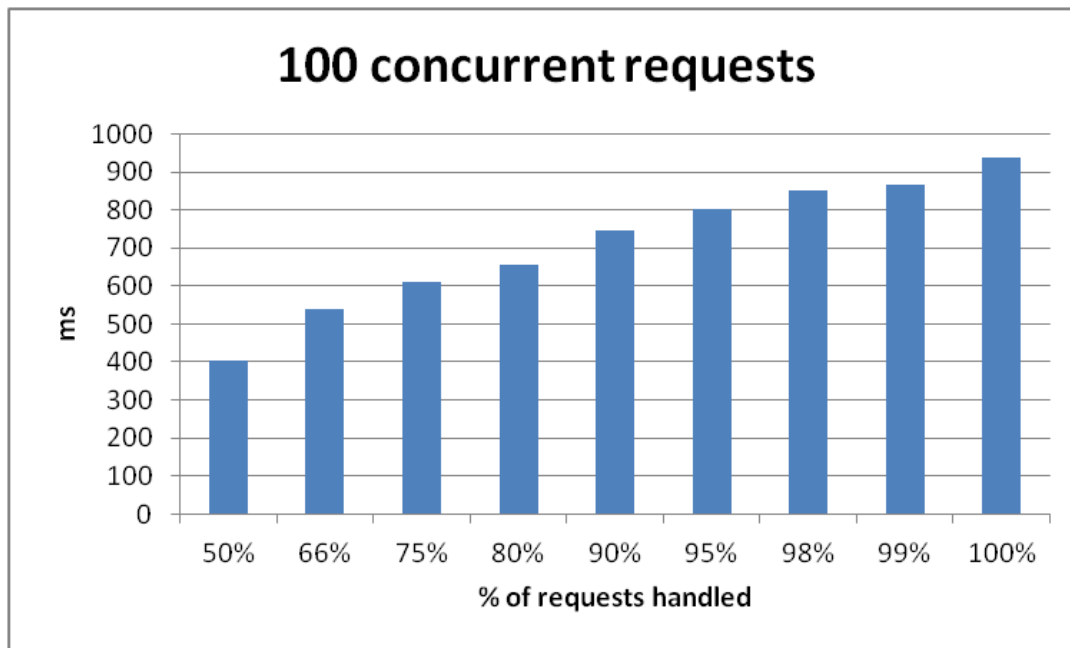


Figure 40 - 100 concurrent requests on an already calculating server

## 3.2 Logging

Node.js platform leaves logging activities completely up to the developer. As such it is important for the developer to log various events in the web application. Without logging all sorts of diagnostics and error detection may become very difficult.

Logging is made easier by the use of frameworks such as Connect or Express, which have their own logging systems built in and all the developer has to do is turn this functionality on and configure it. An example is shown in Figure 41.

```
// Configuration

app.configure(function() {
  app.use(express.logger({ format: ':method :url' }));
  /*
   * Other configuration
   */
});
```

Figure 41 - Example of Express logger configuration

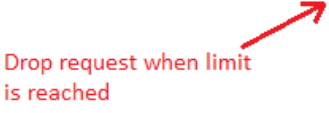
## 3.3 Memory limitations

Node.js provides no built in limitations as to the processes and requests done on the server. This means that it is up to the developer to impose those limits. There is no one fit for all solution for limiting the POST request size at the moment. It is advisable to check the function that is handling your POST requests and limit the size of the allowed request. In

Figure 42 an example is shown where the size of the URL encoded request data is limited to 1000000 characters.

```
/**
 * Parse application/x-www-form-urlencoded.
 */

exports.parse['application/x-www-form-urlencoded'] = function(req, options, fn){
  var buf = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ buf += chunk; if(buf.length > 1000000) req.destroy(); });
  req.on('end', function(){
    try {
      req.body = buf.length
        ? qs.parse(buf)
        : {};
      fn();
    } catch (err){
      fn(err);
    }
  });
};
```



Drop request when limit is reached

Figure 42 - POST request handler with data size limitation

## 3.4 Programming

Developing systems on Node.js platform means programming in JavaScript. It is a loosely typed language with confusing scopes, which get even worse when nested into layers upon layers of callbacks. As such it is important to follow a certain style of programming and follow some guidelines in order to make the program secure and readable.

### 3.4.1 use strict

To give help to developers in the error prone world of ECMAScript the Edition 5 of the standard lets any scope be executed in strict mode by writing „use strict“ in the beginning of it – Figure 43. Running JavaScript in strict mode makes the compiler throw more errors by deeming some practises possibly harmful and ignoring others, which in turn encourages the developer to write more secure and sustainable code.

```
// Non-strict code...
(function() {
  "use strict";

  // Define your library strictly...
})();

// Non-strict code...
```

Figure 43 - Using "strict mode" in JavaScript

There are several things that are addressed by the strict mode. The most important of them is that almost any usage of the name *'eval'* as seen on Figure 44 is not allowed and neither is assigning *'eval'* to a property or a variable. This means that `eval` function cannot be renamed or masked as something else which is important to keep the function visible.

```
// All generate errors...
obj.eval = '...';
obj.foo = eval;
var eval = '';
for ( var eval in obj ) {}
function eval(){}
function test(eval){}
function(eval){}
new Function("eval")
```

Figure 44 - Uses of the keyword "eval" that throw errors in "strict mode"

Introducing new variables through `eval` as seen on Figure 45, will also be blocked. This will prevent attackers from introducing new variables to the global scope, however it will not stop them from changing already existing variables.

```
eval("var a = false;");
print( typeof a ); // undefined
```

Figure 45 - Introducing a variable through "eval" in "strict mode"

Additionally to this limitation `eval` and its counterparts should be avoided as much as possible when dealing with any kind of user input, since they provide potentially dangerous security holes. When they cannot be avoided, then all precautions must be taken with validating the input.

The second important feature of strict mode is that trying to assign value to an undefined variable like `„foo = 'bar';“` will result in an exception. Previously a new global variable was created but strict mode prohibits variable assigning without the `„var“` keyword to prevent accidental access to global scope.

This is important since the ambiguity of JavaScript scopes has been a problem for beginning JavaScript developers. As such accessing the global scope is a common error, which causes unexpected behaviours that are hard to debug and foresee.

There are other effects besides the previously named properties of running code in strict mode. The misunderstood method „**with()**“ is all together not allowed. It also prohibits overwriting function arguments and some properties which are unsafe to change.

### 3.4.2 Try...catch

Servers running on Node.js are quite fragile, since any uncaught error might terminate the main process loop. This is why it is important to try and avoid as many errors as possible and provide error handlers where they can be supplied. However native functions in JavaScript usually do not have the possibility to provide error callbacks.

A possible solution is to use the `try...catch` statement. This statement tries to execute some code and upon error will be directed to the `catch` statement to handle the error. An example is shown in Figure 46.

```
// Try to do something
try{
    test(); // Will cause an error because test is undefined
}
// Catch the error
catch(err){
    console.log(err); // Print the error to console
}
```

Figure 46 - Example usage of „try...catch“ statement in JavaScript

This allows the execution of native functions with the ability to handle possible errors that can occur. It should be used in key places in the application where it is possible for the code to break. This is especially true around user input handling.

The `try...catch` statement can be used together with the `throw` method, which allows us to manually throw a custom error to be handled by the `catch` statement. An example is shown in Figure 47.

```

// Try to do something
try{
    // Defining some variables
    var a = 10,
        b = 7;
    // If the variables do not match a condition
    if((a+b)<20){
        // Throw an error
        throw "Not enough"
    }
}
// Catch the error
catch(err){
    console.log(err); // Print the error to console
}

```

Figure 47 - Example of throwing an error in JavaScript

### 3.4.3 Object properties

One of the most important changes introduced by ECMAScript 5 is that object properties have six new descriptors associated with them that are shown in Figure 48.

```

/*
{
    value:'Value', // Value of the variable
    get: function(){}, // Getter function
    set: function(){}, // Setter function
    writable:true, // Whether you can change the value
    configurable:true, // Whether you can change these properties
    enumerable:true // Whether this property will be iterated over
}
*/

```

Figure 48 - ECMAScript 5 object property descriptors

These descriptors allow the configuration of a property so that it cannot be modified later. This is helpful to prevent unwanted changes to the code – server poisoning. To define a property so that it cannot be modified later the new Object method „defineProperty“ or bulk method „defineProperties“ can be used as shown in Figure 49.



```
// Create object
var obj = {};

Object.defineProperty(obj, {
  "value": { // obj.value = true
    value: true,
    writable: true, // Can be changed
    configurable: false // Configuration cannot be changed
  },
  "name": { // obj.name = "John"
    value: "John",
    writable: false, // Cannot be changed
    configurable: false // Configuration cannot be changed
  }
});
```

Figure 49 - Defining properties in JavaScript using the `defineProperties` function

### 3.5 Modules

A typical Node.js application uses many modules to handle various different tasks that are not covered by the native libraries of Node.js. This is a common and useful behaviour that cuts down on the development time and cost.

Due to the immature and unstable nature of Node.js, however it becomes important to choose modules based on both their functionality and their support. More popular packages will more likely follow the core development faster and as such will not hold the development back in updating to the newest versions of Node.js. It is also important to inspect all used modules for possible security vulnerabilities and malicious scripts, since they are developed by the community and often not verified in any manner.

The NPM makes installing these packages convenient, but it also makes the server vulnerable to malicious tampering when installing packages without properly inspecting them first. It is also important to not run *npm* or *node* as root. This is a common mistake and should be avoided at all costs. It is advisable to create a special web user and run the server under the limited privileges of that.

It is important to follow the changes that are introduced to the core of the application and try to keep the application up to date. The complexity of the task of keeping up with Node.js development often comes down to the right choice of modules for the application.

### 3.6 Reliability of the application

One of the main issues with developing a production ready service is the applications tendency to break. We demonstrated that almost any uncaught error will result in the whole thread being killed, which effectively means that the whole service is offline until it has been restarted.

The first possibility is to debug and perfect the code so much that it is certain that it will not break. Considering how costly such action would be, it can be deemed unfeasible. The second choice is to run the application with „Forever“ [33] or some similar module.

*Forever* is a node library that allows the application to be run, as the name implies, forever. It will restart the node process any time it is killed for whatever reason. This eliminates many problems for production applications. It is no longer an issue that some random piece of code breaks the entire application until it is brought up again.

This also intermittently solves the Slowloris attack, since Slowloris works on Node.js mostly by breaking the main event loop. With Forever once the main loop is broken it is automatically restarted and normal traffic can continue. It does not fully fix the problem since now with continuous restarting of the server it is possible for Slowloris to send enough requests to keep a slower machine's event loop busy enough that normal traffic is mostly blocked. To mitigate the Slowloris attack, along with various other problems, a good firewall configuration is necessary. For Slowloris it is enough to limit the number of connections a single machine can have to anything below 1000. This will stop attacks with a single machine, but distributed attacks are still possible.

While *Forever* solves a many issues with errors tearing down the event loop, it does not solve the real issue behind it. When developing an application that for example deals with large uploads, then even though the thread is restarted upon a fatal crash, all uploads will be terminated with the crash itself.

This is a serious problem and at the moment there are no good solutions for it. The Node.js project team is working on a solution they call „domains“. Domains will allow collecting all the traffic and activity of one connection into a „domain“ and when something goes wrong then it can be killed without terminating the main event loop. This is planned for 1.0

release of Node [34]. A workaround at the moment is to create a new process for each upload so that they cannot destroy each other.

## Conclusion

This thesis set out to analyse the security of the Node.js platform, which is a project trying to change the way web developers think about I/O operations. It has developed an active community, gained some large supporters and is developing rapidly.

An analysis of the application language was given with various examples of possible pitfalls. It showed how the usage of JavaScript can produce some unique problems endangering various aspects of the web server's security.

Testing was done to outline various problem areas of a typical Node.js application. Those problems originate from both the underlying platform and various coding practises that have developed. The results showed that the biggest problem is the services availability – the main event loop is fragile due to the lack of proper error handling. Missing default configuration enforces this effect allowing various Denial of Service attacks against the system.

Another great concern is the integrity of the application. The main event loop can either by malicious tampering or accidental scope errors become corrupted. This can affect the applications behaviour in various ways, which deepen over time as changes accumulate.

Many of those problems can be avoided with security conscious programming standards and some specific programming methods. Possible solutions for many of those problems were shown demonstrating that most of the problems are induced by the developers who are unused to this new environment. However some problems are still rooted to the core of Node.js, which is understandable given the young age of the project.

In conclusion it can be said that Node.js, while providing some unique problems, is not inherently insecure and thus is capable of being used in production. Although mistakes are more common with Node.js applications it is because programmers lack the extensive experience of writing asynchronously on the server side. This issue along with the stability of the applications will improve over time as the project matures.

## References

- [1] “Welcome! - The Apache HTTP Server Project,” [WWW] <http://httpd.apache.org/>. (24.05.2012).
- [2] “Node.js,” [WWW] <http://nodejs.org/>. (22.05.2012).
- [3] “Google- Company,” [WWW] <http://www.google.com/about/company/>. (22.05.2012).
- [4] “About Microsoft: Your Potential. Our Passion.,” [WWW] <http://www.microsoft.com/about/en/us/default.aspx>. (22.05.2012).
- [5] A. White, JavaScript Programmer's Reference, John Wiley & Sons, 2010.
- [6] “Rhino History,” [WWW] <http://www.mozilla.org/rhino/history.html>. (22.05.2012).
- [7] “JavaScript Overview,” [WWW] <http://www.mozilla.org/rhino/overview.html>. (22.05.2012).
- [8] Google Inc, “v8 - V8 JavaScript engine,” [WWW] <http://code.google.com/p/v8/>. (22.05.2012).
- [9] “Node.js ChangeLog,” [WWW] <http://nodejs.org/changelog.html>. (22.05.2012).
- [10] C. Severance, “Java Script: Designing a Language in 10 Days,” *Computer*, pp. 7-8, February 2012.
- [11] J. Resig, “Chapter 2 - Object Oriented Javascript,” in *Pro JavaScript Techniques*, 2006, pp. 25-26.
- [12] ECMA International, “ECMA International,” [WWW] <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>. (22.05.2012).
- [13] M. W. Tom Hughes-Croucher, “A Very Brief Introduction to Node.js,” in *Node: Up and Running: Scalable Server-Side Code with JavaScript*, 2012, pp. 3-4.
- [14] “npm - Node Package Manager,” [WWW] <http://npmjs.org>. (22.05.2012).
- [15] “LinkedIn Mobile | LinkedIn,” [WWW] <http://www.linkedin.com/static?key=mobile>. (23.05.2012).
- [16] “Windows Azure: Cloud Computing | Cloud Services | Cloud Application Development,” [WWW] <http://www.windowsazure.com/en-us/>. (23.05.2012).
- [17] S. T. a. S. Vinoski, “Node.js: Using JavaScript to Build High-Performance Network Programs,” *Internet Computing*, vol. 14, no. 6, pp. 80-83, 2010.
- [18] “Aaronontheweb | Intro to Node.JS for .NET Developers,” [WWW]

- <http://www.aaronstannard.com/post/2011/12/14/Intro-to-NodeJS-for-NET-Developers.aspx>. (22.05.2012).
- [19] “Introduction to Node.js with Ryan Dahl - YouTube,” [WWW] [http://www.youtube.com/watch?v=jo\\_B4LTHi3I](http://www.youtube.com/watch?v=jo_B4LTHi3I). (22.05.2012).
- [20] “ab - Apache HTTP server benchmarking tool,” [WWW] <http://httpd.apache.org/docs/2.0/programs/ab.html>. (22.05.2012).
- [21] “Slowloris HTTP DoS,” [WWW] <http://ha.ckers.org/slowloris/>. (22.05.2012).
- [22] “Nginx - Main,” [WWW] <http://wiki.nginx.org/Main>. (23.05.2012).
- [23] “lighttpd fly light,” [WWW] <http://www.lighttpd.net/>. (23.05.2012).
- [24] “Express - node web framework,” [WWW] <http://expressjs.com/>. (22.05.2012).
- [25] “Jade - Template engine,” [WWW] <http://jade-lang.com/>. (22.05.2012).
- [26] “Connect - High quality middleware for node.js,” [WWW] <http://www.senchalabs.org/connect/>. (22.05.2012).
- [27] “28c3: Effective Denial of Service attacks against web application platforms,” [WWW] <http://www.youtube.com/watch?v=R2Cq3CLI6H8>. (23.05.2012).
- [28] “NPM - README,” [WWW] <http://npmjs.org/doc/README.html>. (23.05.2012).
- [29] “CommonJS effort sets JavaScript on path for world domination | Ars Technica,” [WWW] <http://arstechnica.com/business/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination/>. (23.05.2012).
- [30] B. Sullivan, “Regular Expression Denial of Service Attacks and Defenses,” *MSDN Magazine*, vol. 25, no. 5, pp. 82-85, 2010.
- [31] “NPM - scripts,” [WWW] <http://npmjs.org/doc/scripts.html>. (23.05.2012).
- [32] “Classes.Big Number - JSFromHell.com: JavaScript Repository,” [WWW] <http://jsfromhell.com/classes/bignumber>. (23.05.2012).
- [33] “nodejitsu/forever,” [WWW] <https://github.com/nodejitsu/forever/>. (23.05.2012).
- [34] “Ryan Dahl - History of Node.js - YouTube,” [WWW] <http://www.youtube.com/watch?v=SAc0vQCC6UQ#!>. (23.05.2012).

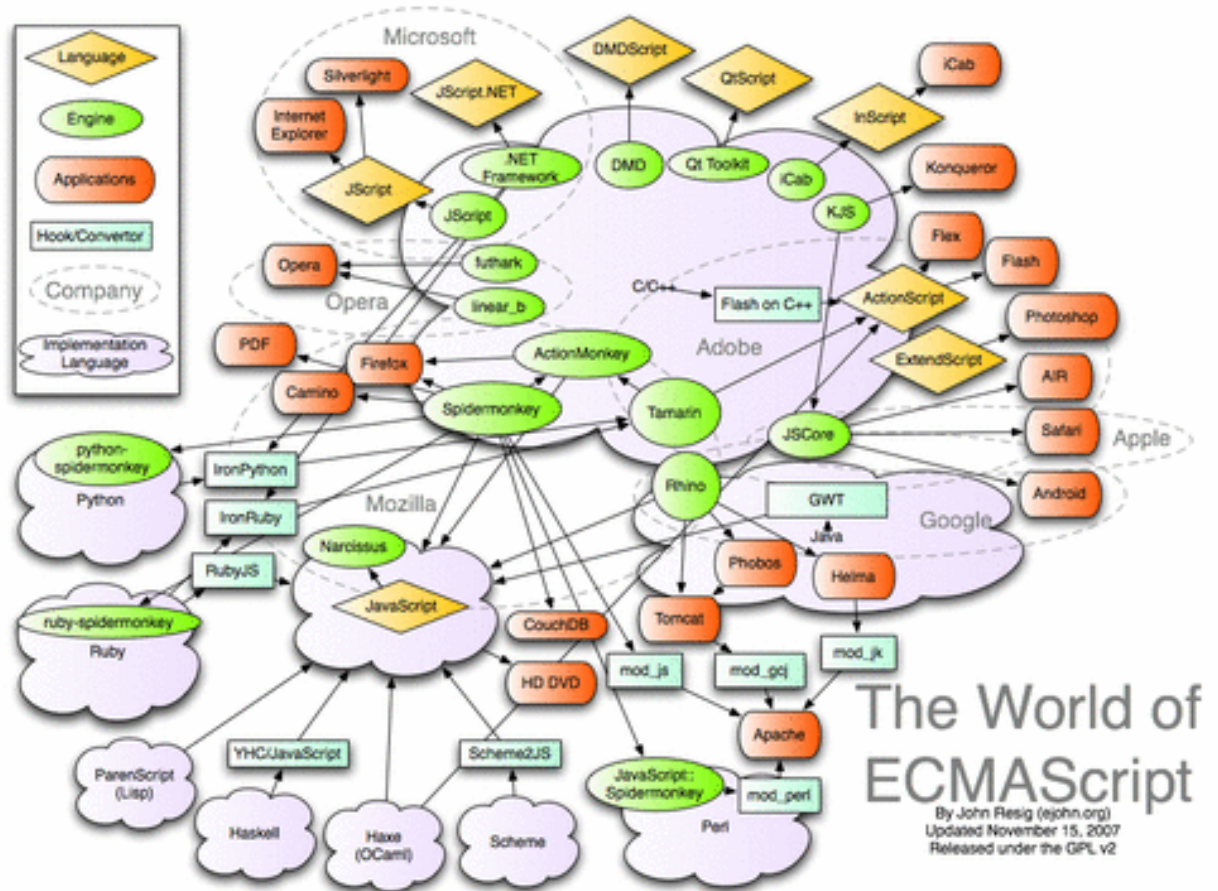
## Figures

Figure 1 - Changing functions at runtime in JavaScript.....	11
Figure 2 - Example of scope in PHP .....	11
Figure 3 - Example of scope in JavaScript.....	12
Figure 4 - Example of authorization failure due to scope misuse .....	12
Figure 5 - Example of "with" statement usage .....	13
Figure 6 - Example of a "with" statement problem .....	13
Figure 7 - Example of floating point arithmetics in JavaScript.....	14
Figure 8 - Eval and its counterparts in JavaScript .....	15
Figure 9 - Node.js processing model [18] .....	17
Figure 10 - Non-blocking "Hello world" example server in Node.js [19] .....	18
Figure 11 - Example of a POST request handler using "eval" that can be corrupted .....	19
Figure 12 - Example excerpt of a corrupting request adding a path to the server.....	19
Figure 13 - Example excerpt of a corrupting request changing the server function .....	20
Figure 14 - 1000 concurrent requests on a "hello world" server .....	21
Figure 15 - System footprint on 1000 concurrent request test .....	22
Figure 16 - Example of a time-consuming process: Fibonaccy calculation.....	23
Figure 17 - Calculating 40th Fibonaccy number with 3 concurrent requests.....	23
Figure 18 - Route handler rendering index file with Jade template .....	24
Figure 19 - Load test with 1000 concurrent connections rendering index file.....	25
Figure 20 - System load with 1000 concurrent request rendering index from file.....	25
Figure 21 - Annotated illustration of load produced by Slowloris attack .....	26
Figure 22 - Example "Hello world" server .....	27
Figure 23 - Example of a JSON request handler without error catching .....	28
Figure 24 - POST request handler without limit .....	29
Figure 25 - Script creating a large POST request.....	29
Figure 26 - Servers attempt to handle a large POST request.....	30
Figure 27 - Hash table working principle [27] .....	32
Figure 28 - Hash table worst case complexity [27] .....	32
Figure 29 - DJBX33A hash function [27] .....	33
Figure 30 - Node Package Manager package description in JSON.....	36
Figure 31 - Important parts of Node Package Manager package description in JSON.....	37
Figure 32 - Regular expression checking the validity of a URL in validator module.....	38
Figure 33 - Bad regular expression analysis.....	38
Figure 34 - Problematic area within a complex regular expression .....	39
Figure 35 - Malicious Node.js package JSON .....	40
Figure 36 - Malicious payload attached to test module.....	40
Figure 37 - Request handler invoking a Fibonaccy calculation through the command line.....	42
Figure 38 - Separate script for the calculation of Fibonaccy numbers.....	42
Figure 39 - 3 concurrent requests to calculate Fibonaccy numbers in a separate thread ....	43
Figure 40 - 100 concurrent requests on an already calculating server .....	44
Figure 41 - Example of Express logger configuration .....	44
Figure 42 - POST request handler with data size limitation.....	45
Figure 43 - Using "strict mode" in JavaScript.....	45

Figure 44 - Uses of the keyword "eval" that throw errors in "strict mode" .....	46
Figure 45 - Introducing a variable through "eval" in "strict mode" .....	46
Figure 46 - Example usage of „try...catch“ statement in JavaScript .....	47
Figure 47 - Example of throwing an error in JavaScript .....	48
Figure 48 - ECMAScript 5 object property descriptors .....	48
Figure 49 - Defining properties in JavaScript using the defineProperties function .....	49



## Appendix 1 – The World of ECMAScript



## Appendix 2 – Testing configuration

Our test server ran as a virtual machine in VirtualBox environment.

### Host machine:

**OS:** 64bit Windows 7 Professional SP1

**CPU:** Intel(R) Core(TM) i3 CPU M 370 @ 2.40GHz

**RAM:** 4GB

### Test machines:

**OS:** Debian GNU/Linux 6.0 Kernel 2.6.32-5-686

**Base memory:** 384MB

**CPUs:** 1 CPU without execution cap

**HDD:** 8GB

**Network:** Bridged adapter Intel(R) Centrino(R) Advanced-N 6200 AGN

### Software:

**Node** version 0.6.14

**Express** version 2.5.9

**Jade** version 0.22.1

**Sequelize** version 1.4.0

**Slowloris** version 0.7 - <http://hackers.org/slowloris/slowloris.pl>

## Appendix 3 – Test server code

```
/**
 * Module dependencies.
 */

var express = require('express'),
    app = express.createServer();

// Configuration

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

var cExec = require('child_process').exec;
// Routes

/* CORRUPTABLE */
// Show the form to client
app.get('/sum',function(req,res){
  res.send('<form method="POST">'+
    '<input name="first" />'+
    '<input name="second" />'+
    '<input type="submit" value="submit" /></form>');
});
// Process the form
app.post('/sum',function(req,res){
  var sum = eval(req.body.first +' '+ req.body.second);
  res.send('the answer is '+sum);
});

/* LONG PROCESS */
// Define the route
app.get('/fibonacciLong/:num',function(req,res){
  var num = fibonacci(req.params.num);
  res.send('Fib is '+num);
});
// Function for calculating fibonacci number recursively
function fibonacci(num){
  if(num<3) return 1;

  return fibonacci(num-1)+fibonacci(num-2);
}

/* SEPARATE PROCESS */
// Define the route
```

```

app.get('/fibonaccy/:num',function(req,res){
  // Execute the separate script as a new process
  cExec('node calcFibo.js '+req.params.num,{cwd:'/home/karl/masters'},function(error, stdout,
  stderr){
    // Send the result of the script
    res.send('Fib is ' +stdout);
  });
});

/* ERRORNOUS */
// Process request
app.post('/json',function(req,res){
  var o = JSON.parse(req.body.o);

  o.ret.result = o.a + o.b; // Do something with o...

  res.send('the answer is '+o.ret.result);
});

// Create a form to send request
app.get('/json',function(req,res){
  res.send('<form method="POST"><input name="o" />'+
    '<input type="submit" value="submit" /></form>');
});

/* SLOWLORIS */
// Define the index handler
app.get('/', function(req,res){
  res.render('index',{layout:false});
});

app.listen(555, function(){
  console.log("Express server listening on port %d in %s mode", app.address().port, app.settings.env);
});

```