Community Experience Distilled

Node Security

Take a deep dive into the world of securing your Node applications with Node Security

Dominic Barnes

[PACKT] PUBLISHING open source*

Table of Contents

Node Security
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Introduction to Node.js
    History of Node.js
    How Node.js differs?
    Securing Node.js applications
    Summary
2. General Considerations
    JavaScript security
    ES5 features
        Strict mode
        Object property descriptors
    Static program analysis
    Considerations for Node.js
        Callback errors
        EventEmitter error handling
        Uncaught exceptions
        Domains
        Process monitoring

- npm modules (third-party code)
- Summary

3. Application Considerations
- Introduction to Express
- Authentication
  - HTTP Basic Authentication
  - HTTP Digest Authentication
  - Introducing Passport.js
  - OpenID
  - OAuth
- Authorization
- Security logging
- Error handling
- Summary

4. Request Layer Considerations
- Limiting the request size
  - Using streams instead of buffering
- Monitoring the event loop's responsiveness
- Cross-site Request Forgery
- Input validation
- Summary

5. Response Layer Vulnerabilities
- Cross-site Scripting (XSS)
- Denial of Service
- Security-related HTTP headers
  - Content security policy
  - HTTP Strict Transport Security (HSTS)
  - X-Frame-Options
  - X-XSS-Protection
  - X-Content-Type-Options
  - Cache-Control
- Summary

Index

Node Security

Node Security

First published: October 2013

Production Reference: 1211013

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-149-7

www.packtpub.com

Cover Image by Prashant Timappa Shetty
(<sparkling.spectrum.123@gmail.com>)

Credits

Author

Dominic Barnes

Reviewers

Johannes Boyne

Dan Palmer

Acquisition Editors

Antony Lowe

Grant Mizen

Commissioning Editor

Mohammed Fahad

Technical Editor

Shashank Desai

Project Coordinator

Romal Karani

Proofreader

Hardip Sidhu

Indexer

Rekha Nair

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Dominic Barnes is a web developer as a hobbyist and by profession. Since writing HTML with Microsoft Notepad back in high school, he has grown in skill through the many opportunities he has had. With experiences in ColdFusion, ASP.NET, PHP, and now Node.js, his passion is to create applications that people find useful. To him, the user experience is paramount and requires writing secure and high-performance code, no matter what platform is being used.

I want to thank Jesus Christ above all, for blessing me with the opportunities to serve people through my work with technology. Without Him, I would not be where I am today and I could not do what I do without His work in my life. He has also blessed me richly through my lovely wife, Joanie, who is the best friend I could ever ask for. She has supported and encouraged me through this entire process, and she helps me work hard and put forth excellence in everything I do. I love her very much, and cannot picture my life without her.

About the Reviewers

Johannes Boyne is the technical project lead for VIRTUAL TWINS®, an indoor-navigation and information system by Archkomm GmbH.

His work with Node.js begun with Version 0.4 and since then he has supported the Node.js community, and recently he joined the Node Security Project as an auditor.

He started as a rich Internet application developer and did consulting work later on till he joined Archkomm for the VIRTUAL TWINS® project. He is interested in new technologies such as NoSQL, high-performance and highly scalable systems, as well as cloud computing. Besides work he loves sports, reading about new scientific researches, watching movies, and travelling.

He also worked on the books *Rich-Internet-Applications with Adobe Flex 3* and *Adobe Flex 4* both by the author, *Simon Widjaja*.

Dan Palmer is a Computer Science Master's student at the University of Southampton, UK, and has worked at as a software developer at a range of companies during his education. He always had a keen interest in security, and has recently completed a placement at MWR InfoSecurity as a security tools developer and penetration tester. He has also worked in the past as a Node.js web developer and Mac OS software developer, making software and services for end users.

I'd like to thank all those I have worked with over the past few years, who have helped me develop my software development skills, and also my appreciation for security in many contexts, and the impact it has across our industry. Thanks *Keith*, *Gerhard*, *Geoff*, *Dan*, *Mike*, *Martin*, *Dave*, and everyone else. I really appreciate the help and advice you've all given me.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content

- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Node.js is a fast-growing platform for building server applications using JavaScript. Now that it is being used more widely in production settings, Node.js applications will begin to be specifically targeted for security vulnerabilities. Protecting your users will require the understanding of attack vectors that are unique to Node.js as well as those shared with other web application platforms.

What this book covers

Chapter 1, *Introduction to Node.js*, introduces Node.js and explains how it differs from other development platforms.

Chapter 2, *General Considerations*, goes over the general security considerations, particularly within JavaScript itself as well as Node.js applications in general.

Chapter 3, *Application Considerations*, addresses the security issues related to the applications in general, including authentication, authorization, and error handling.

Chapter 4, *Request Layer Considerations*, covers vulnerabilities that are specific to request handling, such as **Cross-site Request Forgery** (**CSRF**).

Chapter 5, *Response Layer Vulnerabilities*, deals with the issues that arise during or after the response is processed, such as **Cross-site scripting** (**XSS**).

To get the most from this book, you should have Node.js installed on your system. Instructions are available for many platforms at http://nodejs.org/. Be familiar with npm and its command-line usage. It is bundled with Node.js, so no additional installation is required.

Who this book is for

This book is intended to help the developers to secure their Node.js applications, whether they are already using it in production, or considering it for their next project. Understanding of JavaScript is a prerequisite, and some experience with Node.js is recommended, but not required.

Conventions

In this book, you will find a number of styles of text that distinguishes between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "It should be noted that the `EventEmitter` object has a very specific behavior regarding the error event."

A block of code is set as follows:

```
function sayHello(name) {
       "use strict"; // enables strict mode for this
function scope
      console.log("hello", name);
}
```

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Introduction to Node.js

Node.js has ushered in the age of server-side JavaScript, the next logical step from the renaissance that client-side JavaScript has experienced over the last few years. While Node.js is not the first server-side JavaScript implementation, it has certainly become the most popular. By leveraging the best features of JavaScript as a language and nurturing a vibrant community, Node.js has become a tremendously popular platform and framework, with no signs of slowing down. A great description of what Node is can be found at http://nodejs.org/:

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

History of Node.js

The project began as the brain-child of Ryan Dahl back in 2009. At JSConf.eu (a conference held annually in Europe) that year, he made his presentation and changed the face of JavaScript development. His speech included an impressive demonstration of a complete IRC server that had been written in roughly 400 lines of JavaScript. During his presentation, he outlined why he started the project, why JavaScript became such an integral part of it, and what goals he sought to accomplish along the way in the field of server programming – particularly with regards to how we deal with input and output (I/O).

Later that year, the **npm** project began, with the goal of managing packages for Node.js applications, as well as creating a publicly available registry for sharing code between Node.js developers. As of version 0.6.3 of Node.js, npm is deployed and installed alongside Node.js, making it the de facto package manager.

How Node.js differs?

What makes Node.js different from other platforms is in how it approaches I/O. It uses an event-loop in conjunction with asynchronous I/O, which allows it to achieve a high level of concurrency with a light footprint.

Typically, when a program needs some sort of external input, it does so in a synchronous fashion. The following line of code should be very familiar to any programmer:

```
var results = db.query("SELECT * FROM users");
print(results[0].username);
```

All we are doing here is querying a SQL database for a list of all users, and then we are printing out the first user's name. When querying a database like this, there are many intermediary steps that need to be taken, such as:

1. Opening a connection to the database server.
2. Transmitting the request over the network to that server.
3. The server itself needs to process the request after receiving it.
4. The server must transmit the response back over the network to our application.

This list does not cover all the specifics, as there are many more factors than are necessary for the point to be made. By looking at our source code, this is treated as an instantaneous action, but we know better. We often neglect this wasted time because it is so fast that we don't notice it happening. Consider the following table:

The Cost of I/O	
L1-cache	3 cycles

L2-cache	14 cycles
RAM	250 cycles
Disk	41,000,000 cycles
Network	240,000,000 cycles

Each I/O operation has a cost, which is paid directly in a program that uses synchronous I/O. There could easily be millions and millions of clock cycles that occur before the program can progress.

When writing an application server, a program like this can only serve one user at a time, and the next user cannot be served until all the I/O and processing is complete for the previous user. This is unacceptable of course, so the easiest solution is to create a new thread for each incoming request, so they can run in parallel.

This is how the **Apache** web server works, and it is not difficult to implement. However, as the number of simultaneous users increase, the amount of memory used also increases. Each of those threads requires overhead at the operating system level, and it adds up pretty quickly. In addition, the overhead of context switching between those threads is more time consuming than desired, further compounding the problem.

The **nginx** web server uses an event loop at its core to handle processes. By doing so, it is able to handle more simultaneous users at once, with fewer resources. An event loop requires that the bits of processing be broken up into small pieces, and run in a single queue. This removes the high cost of creating threads, switching back and forth between those threads, and requires less demand of the overall system. At the same time, it fills in the processing gaps, particularly those that occur during the wait for I/O to complete.

Node.js takes the event-driven model that nginx uses to such great success, and it exposes that same capability for many types of applications. In Node.js, all I/O is entirely asynchronous and does not block the rest of the application thread. The Node.js API accepts function parameters (usually known as a "callback function") for all I/O operations. Node.js then fires off that I/O operation, and lets another thread outside the application do the processing. After that, the application is free to continue handling other requests. Once the requested operation is complete, the event-loop is notified, and the callback function is invoked with the results.

As it turns out, waiting for I/O to complete is the most expensive part of many applications in terms of raw processing time. With Node.js, the time spent waiting for I/O is completely detached from the rest of your application's processing time. Your application just uses callback functions to process results as simple events, and JavaScript's ability to use closure retains the function's context, despite being executed asynchronously.

If you were to take up the task of writing a multi-threaded application, you would have to concern yourself with concurrency problems like deadlocks, which are very difficult (if not impossible) to reproduce and debug in real-world applications. With Node.js, your primary application logic runs on a single thread, free of such concurrency problems, while the time-consuming I/O is handled on your behalf by Node.js.

Like any other platform, Node.js has an API developers can use to write their applications. JavaScript itself lacks a standard library, particularly for performing I/O. This actually turned out to be one of the reasons that Ryan Dahl chose JavaScript. As the core API can be built from the ground up, without needing to worry about creating conflicts with a standard library, in case it is done wrong (given JavaScript's history, this is not an unreasonable assumption).

That core library is minimalistic, but it does include modules for the essentials. This includes, but is not limited to: filesystem access, network communication, events, binary data structures, and streams. Many of these APIs, while not difficult to use, are very low-level in implementation. Consider this "Hello World" demonstration straight from the Node.js

website (with comments added):

```
// one of the core modules
var http = require('http');
// creates an http server, this function is called for
each request
http.createServer(function (req, res) {
  // these parameters represent the request and
response objects
  // the response is going to use a HTTP status code
200 (OK)
  // the content-type HTTP header is set as well
  res.writeHead(200, {'Content-Type': 'text/plain'});
  // lastly, the response is concluded with simple
text
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at
http://127.0.0.1:1337/');
```

This server uses the **http** core module to set up a web server that simply sends "Hello World" to anyone who makes a request of it. This is a simple example, but without comments, this consists of only six lines of code in all.

The Node.js team has opted to keep the core library limited in scope, leaving the community of developers to create the modules they need for everything else, such as database drivers, unit-testing, templating, and abstractions for the core API. To aid in this process, Node.js has a package manager called npm.

npm is the tool that handles installing dependencies for Node.js applications. It opts for locally bundled dependencies, rather than using a single global namespace. This allows different projects to have their own dependencies, even if the version varies between those projects.

Tip

Downloading the example code

You can download the example code files for all Packt books you

have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/supportand register to have the files e-mailed directly to you.

In addition to allowing for the use of third-party modules, npm also makes contributing to the registry a public affair. Adding a module to the registry is as simple as a single command, making the barrier to enter extremely low. Today, the npm registry has over 42,000 packages listed and is growing faster by the day.

With the registry growing so fast, it's obvious there is a vibrant ecosystem behind it. I can personally attest to the fact that the Node.js developer community is very friendly, extremely prolific, and has an enormous amount of enthusiasm.

Securing Node.js applications

When it comes to securing your application, there are many factors to consider. We will start by examining JavaScript itself, then analyze Node.js as a platform, and reveal some of the internals that are relevant to the discussion. After that, we will investigate considerations and patterns for your applications as a whole. Last, we will survey vulnerabilities at the request and response level of your applications. By the end of this book, you should have enough understanding of the internals of Node.js to not only address what we are discussing here, but also to grasp any future vulnerability that may appear for your applications.

Summary

In this chapter, we explored the history of the Node.js project itself, and gave some background to the development environment and community. In the next chapter, we will start by looking at security features present within the JavaScript language itself.

Chapter 2. General Considerations

Building secure Node.js applications will require an understanding of the many different layers that it is built upon. Starting from the bottom, we have the language specification that defines what JavaScript consists of. Next, the virtual machine executes your code and may have differences from the specification. Following that, the Node.js platform and its API have details in their operation that affect your applications. Lastly, third-party modules interact with our own code and need to be audited for secure programming practices.

First, JavaScript's official name is ECMAScript. The international **European Computer Manufacturers Association** (**ECMA)** first standardized the language as **ECMAScript** in 1997. This ECMA-262 specification defines what comprises JavaScript as a language, including its features, and even some of its bugs. Even some of its general quirkiness has remained unchanged in the specification to maintain backward compatibility. While I won't say the specification itself is required reading, I will say that it is worth considering.

Second, Node.js uses Google's **V8** virtual machine to interpret and execute your source code. While developing for the browser, you have to consider all the other virtual machines (not to mention versions), when it comes to available features. In a Node.js application, your code only runs on the server, so you have much more freedom, and you can use all the features available to you in V8. Additionally, you can also optimize for the V8 engine exclusively.

Next, Node.js handles setting up the event loop, and it takes your code to register callbacks for events and executes them accordingly. There are some important details regarding how Node.js responds to exceptions and other errors that you will need to be aware of while developing your applications.

Atop Node.js is the developer API. This API is written mostly in JavaScript which allows you, as a JavaScript developer, to read it for

yourself, and understand how it works. There are many provided modules that you will likely end up using, and it's important for you to know how they work, so you can code defensively.

Last, but not least, the third-party modules that npm gives you access to, are in great abundance, which can be a double-edged sword. On one hand, you have many options to pick from that suit your needs. On the other hand, having a third-party code is a potential security liability, as you will be expected to support and audit each of these modules (in addition to their own dependencies) for security vulnerabilities.

JavaScript security

One of the biggest security risks in JavaScript itself, both on the client and now on the server, is the use of the `eval()` function. This function, and others like it, takes a string argument, which can represent an expression, statement, or a series of statements, and it is executed as any other JavaScript source code. This is demonstrated in the following code:

```
// these variables are available to eval()'d code
// assume these variables are user input from a POST
request
var a = req.body.a; // => 1
var b = req.body.b; // => 2
var sum = eval(a + "+" + b); // same as '1 + 2'
```

This code has full access to the current scope, and can even affect the global object, giving it an alarming amount of control. Let's look at the same code, but imagine if someone malicious sent arbitrary JavaScript code instead of a simple number. The result is shown in the following code:

```
var a = req.body.a; // => 1
var b = req.body.b; // => 2; console.log("corrupted");
var sum = eval(a + "+" + b); // same as '1 + 2;
console.log("corrupted");
```

Due to how `eval()` is exploited here, we are witnessing a "remote code

execution" attack! When executed directly on the server, an attacker could gain access to server files and databases. There are a few cases where `eval()` can be useful, but if the user input is involved in any step of the process, it should likely be avoided at all costs!

There are other features of JavaScript that are functional equivalents to `eval()`, and should likewise be avoided unless absolutely necessary. First is the `Function` constructor that allows you to create a callable function from strings, as shown in the following code:

```
// creates a function that returns the sum of 2
arguments
var adder = new Function("a", "b", "return a + b");
adder(1, 2); // => 3
```

While very similar to the `eval()` function, it is not exactly the same. This is because it does not have access to the current scope. However, it does still have access to the global object, and should be avoided whenever a user input is involved.

If you find yourself in a situation where there is an absolute need to execute an arbitrary code that involves user input, you do have one secure option. Node.js platform's API includes a **vm** module that is meant to give you the ability to compile and run code in a sandbox, preventing manipulation of the global object and even the current scope.

It should be noted that the vm module has many known issues and edge cases. You should read the documentation, and understand all the implications of what you are doing to make sure you don't get caught off-guard.

ES5 features

ECMAScript5 included an extensive batch of changes to JavaScript, including the following changes:

1. Strict mode for removing unsafe features from the language.
2. Property descriptors that give you control over object and property access.
3. Functions for changing object mutability.

Strict mode

Strict mode changes the way JavaScript code runs in select cases. First, it causes errors to be thrown in cases that were silent before. Second, it removes and/or change features that made optimizations for JavaScript engines either difficult or impossible. Lastly, it prohibits some syntax that is likely to show up in future versions of JavaScript.

Additionally, strict mode is opt-in only, and can be applied either globally or for an individual function scope. For Node.js applications, to enable strict mode globally, add the `--use_strict` command line flag, while executing your program.

Tip

While dealing with third-party modules that may or may not be using strict mode, this can potentially have negative side effects on your overall application. With that said, you could potentially make strict mode compliance a requirement for any audits on third-party modules.

Strict mode can be enabled by adding the `"use strict"` pragma at the beginning of a function, before any other expressions as shown in the following code:

```
function sayHello(name) {
    "use strict"; // enables strict mode for this
function scope
    console.log("hello", name);
}
```

In Node.js, all the required files are wrapped with a function expression that handles the `CommonJS` module API. As a result, you can enable strict mode for an entire file, by simply putting the directive at the top of the file. This will not enable strict mode globally, as it would in an environment like the browser.

Strict mode makes many changes to the syntax and runtime behavior, but for the sake of brevity we will only discuss changes relevant to application security.

First, scripts run via `eval()` in strict mode cannot introduce new variables to the enclosing scope. This prevents leaking new and possibly conflicting variables into your code, when you run `eval()` as shown in the following code:

```
"use strict";
eval("var a = true");
console.log(a); // ReferenceError thrown – a does not
exist
```

In addition, the code run via `eval()` is not given access to the global object through its context. This is similar, if not related, to other changes for function scope, which will be explained shortly, but this is specifically important for `eval()`, as it can no longer use the global object to perform additional black magic.

It turns out that the `eval()` function is able to be overridden in JavaScript. It can be accomplished by creating a new global variable called `eval`, and assigning something else to it, which could be malicious. Strict mode prohibits this type of operation. It is treated more like a language keyword than a variable, and attempting to modify it will result in a syntax error as shown in the following code:

```
// all of the examples below are syntax errors
```

```
"use strict";
eval = 1;
++eval;
var eval;
function eval() { }
```

Next, the function objects are more tightly secured. Some common extensions to ECMAScript add the `function.caller` and `function.arguments` references to each function, after it is invoked. Effectively, you can "walk" the call stack for a specific function by traversing these special references. This potentially exposes information that would normally appear to be out of scope. Strict mode simply makes these properties throw a `TypeError` remark, while attempting to read or write them, as shown in the following code:

```
"use strict";
function restricted() {
    restricted.caller;    // TypeError thrown
    restricted.arguments; // TypeError thrown
}
```

Next, `arguments.callee` is removed in strict mode (such as `function.caller` and `function.arguments` shown previously). Normally, `arguments.callee` refers to the current function, but this magic reference also exposes a way to "walk" the call stack, and possibly reveal information that previously would have been hidden or out of scope. In addition, this object makes certain optimizations difficult or impossible for JavaScript engines. Thus, it also throws a `TypeError` exception, when an access is attempted, as shown in the following code:

```
"use strict";
function fun() {
    arguments.callee; // TypeError thrown
}
```

Lastly, functions executed with `null` or `undefined` as the context no longer coerce the global object as the context. This applies to `eval()` as we saw earlier, but goes further to prevent arbitrary access to the global object in other function invocations, as shown in the following code:

```
"use strict";
```

```
(function () {
    console.log(this); // => null
}).call(null);
```

Strict mode can help make the code far more secure than before, but ECMAScript 5 also includes access control through the property descriptor APIs. A JavaScript engine has always had the capability to define property access, but ES5 includes these APIs to give that same power to application developers.

Object property descriptors

Object properties have the following three hidden attributes that determine what mutations can occur to them:

- `writable`: If this is `false` means the property value cannot be changed (in other words, read only)
- `enumerable`: If this is `false` means the property will not come up during for in loops
- `configurable`: If this is `false` means the property cannot be deleted

While defining an object property with an object literal or through assignment, which is the most common method, each of these three hidden properties defaults to `true`. This makes the property completely open to modification in every respect. However, there are a few new functions that allow application developers to set these property attributes on their own, restricting access to certain object properties. The property descriptor API is completely opt-in, and the default behavior of object properties does not change, even in ES5.

First, the `Object.defineProperty()` function allows you to specify a single property and its accessor descriptor on a specified object. It takes three arguments: the target object, the name of the new property, and the descriptor object mentioned earlier. An accessor descriptor is just an object that contains specified properties corresponding to the attributes listed earlier.

Tip

The accessor descriptor tells JavaScript engine, the access level to give to our new property. While using `Object.defineProperty()` and its related functions, it is important to note that all the descriptor attributes value are by default set to `false`. This is the opposite effect when compared to basic assignment.

```
var o = {};

// the next 2 statements are completely identical in
result

o.a = "A";

Object.defineProperty(o, "a", {
    writable: true,
    enumerable: true,
    configurable: true,
    value: "A"
});
```

Both of these statements have the same result, and the latter is much more verbose. However, traditional assignment cannot affect any of the descriptors, unlike the latter. Let's see what it takes to create a "locked down" property:

```
var o = {};

Object.defineProperty(o, "a", {
    value: "A"
});
```

What we have just done is created a property that cannot be written, deleted, or enumerated, making it immutable. This allows application developers to control data access, even while sharing objects across various code boundaries.

One last capability afforded by accessor descriptors is to allow developers to create getter and setter functions for specific properties. A getter is a function that returns the data when a property is accessed, and a setter stores the data that is sent via an assignment. This is illustrated in the following code:

```
var person = {
    firstName: "Dominic",
    lastName: "Barnes"
};

Object.defineProperty(person, "name", {
    enumerable: true,
    get: function () {
        return this.firstName + " " + this.lastName;
    },
    set: function (input) {
        var names = input.split(" ");
        this.firstName = names[0];
        this.lastName = names[1];
    }
});

console.log(person.name); // => "Dominic Barnes"
```

This code creates a property that contains data from two other properties on the same object, and it is computed dynamically. The same could be accomplished with a function in many cases, but this enables more separation between the two operations, without needing two separate functions on the object itself.

The next function, `Object.defineProperties()`, is similar. This one, however, only takes two arguments, the host object and another object that is a hash of multiple properties, where the property values are all accessor descriptors. This is seen in the following code:

```
var letters = {};

Object.defineProperties(letters, {
    a: {
        enumerable: true,
        value: "A"
    },
    b: {
        enumerable: true,
        value: "B"
    }
});

console.log(letters.a); // => "A"
console.log(letters.b); // => "B"
```

This allows us to condense multiple property definitions into a single function call, which is more about convenience than anything else. Next up is the most powerful of them all: the `Object.create()` function. This function creates a completely new object from scratch, and also assigns it a prototype. This is reflective of the prototypal nature of JavaScript, and we will not take time to discuss that further, as it is not particularly relevant to this discussion.

This function only takes two arguments, the prototype for the new object (or `null` to assign no inheritance at all), and a properties object just like we use in `Object.defineProperties()`, as shown in the following code:

```
var constants = Object.create(null, {
    PI: {
        enumerable: true,
        value: 3.14
    },
    e: {
        enumerable: true,
        value: 2.72
    }
});
```

By setting the prototype as `null`, instead of some other object, we have created a completely plain object that inherits nothing, not even from the `Object.prototype` object. This is desirable as even modifications to `Object.prototype` (which is a bad idea anyway) will not adversely affect objects created with this method.

There are a few other special functions for changing an object's accessibility. First is the `Object.preventExtensions()` function, which prevents new properties from being added to the object specified, as shown in the following code:

```
var o = {
    a: "A",
    b: "B",
    c: "C"
};

o.d = "D"; // works as expected
```

```
Object.preventExtensions(o);

o.e = "E"; // will not work
```

As you can see, this allows you to configure an object so that nobody else can create additional properties on your object. If you include strict mode in the mix, the last assignment will throw an error rather than failing silently. Also, it should be noted that this operation cannot be reversed once it has occurred.

Next is the `Object.seal()` function which takes an object, and prevents properties from being deleted, in addition to the effects of the `Object.preventExtensions()` function. In other words, this takes all the existing properties and sets their configurable property attributes to `false`.

```
var o = {
    a: "A",
    b: "B",
    c: "C"
};

delete o.c; // works as expected

Object.seal(o);

delete o.b; // will not work
```

This is powerful because we can preserve the structure of an object, but still allow property values to change. Like before, this operation cannot be reversed. In addition, adding the strict mode causes an exception to be thrown, rather than allowing the operation to fail silently.

Last up is the most powerful of them all, the `Object.freeze()` function. This function applies all the same effects as `Object.seal()`, and also completely locks down all the properties. No values can be changed (that is, all writable attributes are set to `false`), and the property descriptors are all unmodifiable. This makes the object effectively immutable, and prevents all other attempts to change anything about the object, as shown in the following code:

```
var o = {
    a: "A",
```

```
    b: "B",
    c: "C"
};

// works as expected
o.a = 1;
delete o.c;

Object.freeze(o);

// will not work
o.a = "A";
delete o.b;
```

Freezing an object is, like the other operations, irreversible. In strict mode, errors will be thrown during any attempt to write or change the object.

Static program analysis

Keeping a track of all the things we have discussed here can be overwhelming. The problem is compounded, when a team of people are working on the same project. Tools that perform static analysis take your source code (without executing it), and check for specific code patterns that you can configure.

For example, you can configure **JSHint** to forbid the use of `eval()` and require strict mode for all your functions. By letting it examine your source code, it will alert you when these rules are violated. This can be used in conjunction with version control to prevent insecure code from being added to your project's repository. In addition, it can also be used prior to releases to ensure that all the code is secured before heading out to production.

JSHint is a community-driven fork of the **JSLint** project. JSLint is opinionated and it is not as configurable as many desired, so JSHint was created to fill that gap. Both are great tools, and I highly recommend you adopt either one for your JS projects. While static analysis will not catch everything, it will help ensure a higher quality of code through automation.

Considerations for Node.js

JavaScript has exceptions built into the language as an error-handling construct. When an exception is thrown, there needs to be some code to detect that error and handle it appropriately. However, if an exception remains uncaught, it will trigger a show-stopping error.

In the browser, an uncaught exception immediately halts any execution that takes place. This will not cause your web page to crash, but it has the potential to leave your application in an unstable place.

In Node.js, an uncaught exception will terminate the application thread. This is very different from other server-side programming languages like PHP, where a similar error only causes a single request to fail. Now, you must contend with your entire server and application being abruptly halted.

Callback errors

The first step you can take is to make sure you throw errors in an expected and predicable way, so they can be effectively caught later. In Node.js, the convention for asynchronous actions that use a callback is to send an `Error` object, to that callback function, as the first argument. This is the standard convention used in Node.js core, and it has been widely adopted by the community.

```
var fs = require("fs");

fs.readFile("/some/file", "utf8", function (err,
contents) {
    // err will be...
    // null if no error has occurred … or
    // an Error object with information about the
error
});
```

The preceding code simply reads a file into a string. This operation has a callback that takes two arguments. The first is an `Error` object, but only if an error occurred during this I/O operation, such as the file not existing.

By simply passing the error object as a function argument, this does not technically "throw" an exception. Your application should still handle these errors, by correcting them, if possible. If an unexpected error occurs, or if it cannot be corrected directly, you should throw that error yourself, rather than swallowing errors quietly, and creating hard-to-debug scenarios for yourself later.

EventEmitter error handling

The Node.js core has a widely used utility object called the `EventEmitter`. This is an object that can be instantiated or inherited from that allows for binding to and emitting events for asynchronous actions. When an error is encountered by an `EventEmitter` object, the convention is to emit an error event with an `Error` object as a parameter.

```
var http = require("http");

http.get("http://nodejs.org/", function (res) {
    // res is an EventEmitter that represents the HTTP
response

    res.on("data", function (chunk) {
        // this event occurs many times
        // each with a small chunk of the response
data
    });

    res.on("error", function (err) {
        // this event occurs if an error occurs during
transmission
    });
});
```

The preceding code simply makes an HTTP request to http://nodejs.org/. The resulting object is an `EventEmitter` object that represents the HTTP response. It emits multiple data events, as it receives data from the server, and if an error occurs during transmission (similar to a network disconnection) then an `error` event is emitted.

It should be noted that the `EventEmitter` object has a very specific behavior regarding the `error` event. If you have an `EventEmitter` object that emits an `error` event, but has no attached listeners to respond to the

event, then the corresponding `Error` object is thrown, and will likely become an uncaught exception. This means that any unhandled error events will crash your application, so always bind an `error` event handler, while using the `EventEmitter` object.

Uncaught exceptions

When an uncaught exception does occur, Node.js will print the current stack trace, and then terminate the thread. There is a global object available to all Node.js applications called `process`. It is an `EventEmitter` object with a special event called `"uncaughtException"` that gets emitted, when an uncaught exception is brought up to the main event loop. By binding to this event you can set up custom behavior, such as sending an email, or writing to a special log file. This can be seen in the following code:

```
process.on("uncaughtException", function (err) {
    // we're just executing the default behavior
    // but you can implement your own custom logic
here instead
    console.error(err);
    console.trace();
    process.exit();
});
```

In the preceding code, I've simply done what Node.js does by default. As I mentioned before, you can implement your own error-logging procedures. You need to make sure to terminate the process yourself via the `process.exit()` function, if you are using a custom handler.

While it is possible to continue the application after an uncaught exception, it is not recommended! By definition, an uncaught exception has interrupted the normal flow of your application, leaving it in an unstable and unreliable state. If you simply swallow the error, and continue processing, then you are wandering into a dangerous territory. The Node.js documentation equates this with unplugging a computer to shut it down. You can get away with it a few times, but if it keeps happening repeatedly, the system will become increasingly unstable and unpredictable.

Domains

While the `uncaughtException` event allows us to handle errors, it is still rather crude. You lose much of the original context from where the error originates, which makes it a bit more difficult to debug later. As of Node.js v0.8, there is a new error-handling mechanism available, called **Domains**. They are a way to group different I/O operations together so that in the event of an error, the domain object is notified instead of the process object via the `uncaughtException` event. This allows you to preserve the context of the error itself, and helps you to prepare for and correct the error in future.

In addition to preserving context, domains also allows you to gracefully shut down related services in the event of an error. If you have an HTTP server running, and an error occurs for one of your users, simply shutting down the server will immediately interrupt any other users that are currently using the server at the same time. This isn't fair to those users, so we need to be able to shut down our server more gracefully. We should stop the server from accepting new connections, and let the current requests be fulfilled before shutting down the server.

```
var http = require("http"),
    domain = require("domain"),
    server = http.createServer(),
    counter = 0;
server.on("request", function (req, res) {
    // this domain will cover this entire
request/response cycle
    var d = domain.create();
    d.on("error", function (err) {
        // outputs all relevant context for this error
        console.error("Error:", err);

        res.writeHead(500, { "content-type":
"text/plain" });
        res.end(err.message);

        // stops the server from accepting new
connections/requests
        console.warn("closing server to new
connections");
        server.close(function () {
            console.warn("terminating process");
```

```
            process.exit(1);
        });
    });

    // adding the req and res objects to the domain
allows
    // errors they encounter to be handled by the
domain
    // automatically
    d.add(req);
    d.add(res);

    d.run(function () {
        if (++counter === 4) {
            throw new Error("Unexpected Error");
        }

        res.writeHead(200, { "content-type":
"text/plain" });
        res.end("Hello World\n");
    });
});

server.listen(3000);
```

The preceding code sets up a simple HTTP server that will respond four times before an error occurs. For each request, a domain is created, which can be passed around to all the various pieces of our request handler, and any asynchronous operations can be run in the domain's context. On request number `4`, we will throw an `Error` object. The domain has an error event handler that outputs the error information, a stack trace, and then proceeds to shut down the server. First, it sends the current request an error message, then it stops accepting new requests, and finishes serving all the current requests it has in its queue. Once this is completed, the process itself is terminated.

We could technically implement what I demonstrated here with the `uncaughtException` event. However, if you are running multiple servers (for example, an HTTP server and a WebSocket server) side by side in your application (or even running multiple processes with the cluster module), that event handler won't necessarily give you the context you need to handle those errors specific to the server that encountered the error. In fact, you won't even be able to distinguish between different requests with the `uncaughtException` event, as that context is lost as well.

With domains, you can handle errors more gracefully, without losing context.

Node.js has a module called **cluster**, which allows you to take advantage of multiple core environments. It does this by spawning multiple worker processes that share the same server port, and the `cluster` module handles message passing between those processes for you. If an error happens in one of those workers, a domain would allow you to easily shut down only that single server and worker process, while letting the others continue operating normally. Once that process finishes cleaning up and exiting, you can spawn a brand new one to take its place, and your application will experience zero downtime as a result.

Process monitoring

With that said, stuff is going to go wrong. You shouldn't ignore uncaught exceptions as your application will be unstable, and will leak references and memory. The only safe way to deal with uncaught exceptions is to stop that process. The implication here is that your server will be unavailable to other users. This means that if a malicious user can figure out a way to trigger an uncaught exception on your server, they are effectively executing a denial of service attack against your other users.

The solution is to have a process monitor that can watch your application process and automatically restart it whenever it is stopped. There are many options out there, including ones that are platform-specific. Some available process monitors include forever, mon, and upstart. The point is you should implement some sort of process monitoring, so you do not have to manually restart your applications, if something goes wrong.

Once you have a process monitor in place, be sure to configure it to log errors somewhere so that you can keep a track, in order to correct harmful and fatal errors in your application. It is also wise to monitor how often your application crashes, and correct errors as fast as possible.

npm modules (third-party code)

As mentioned before, one of the biggest features of Node.js is its vibrant community and fast-growing registry of modules. Because the Node.js core API is intentionally small and focused, you are likely to incorporate other modules, so you don't have to write a lot from scratch.

Just as you will take efforts to audit your code for security practices, you should also take an active role in monitoring the npm modules; you end up including in your project. Many projects out there on npm are completely open source, and often available on GitHub or other similar online resources. This makes it easy to look through the source manually for things that stand out. As a last resort, you can inspect the local packages that npm downloads while installing dependencies, although you are not guaranteed to get everything that is part of the package's development environment.

While picking out modules to adopt, look for the ones that include a test suite of some sort. If they have running tests, it will be easier for you to know for certain that functionality is working as designed. Second, look for projects that incorporate some static analysis, which will usually come in the form of JSHint or JSLint. Look at their style guide or static analysis configuration to understand what rules they abide by. Many projects of this type have some sort of build process, which likely will include a way to run automated tests, static analysis, and other related tools.

One of the focuses that Node.js developers place into their modules is to make them small, highly-focused, and composable (that is, they are easily interoperable with other modules). As such, they usually are very small in terms of lines of code and complexity, making it much easier to write securely and in a testable fashion. This works greatly to Node.js platform's advantage when it comes to application security.

There is an up-and-coming undertaking called the **Node Security Project**, which can be found at http://nodesecurity.io/. Their goal is to audit every single npm module for security vulnerabilities. They are in need of Node.js developers and security researchers to assist them, as they have a monumental task ahead of them. If you are interested in

securing your own applications already, you can likely contribute the time you spend auditing modules that you end up using to this team for their own registry. This is a great way to accomplish your own goals, as well as contributing to the Node.js community as a whole.

Summary

In this chapter, we examined the security features that applied generally to the language of JavaScript itself, including how to use static code analysis to check for many of the aforementioned pitfalls. Also, we looked at some of the inner workings of a Node.js application, and how it differs from typical browser development, when it comes to security. Lastly, we briefly discussed the npm module ecosystem, and the Node Security Project, which aims to audit each and every module for security purposes. In the next chapter, we will look at security considerations for applications in general.

Chapter 3. Application Considerations

Now it's time to deal with real-world applications! As mentioned before, one of Node.js platform's killer features is the wealth of modules and rapidly moving community. It is still important to audit every module that you use for security, but using modules is likely going to become an indispensable part of your workflow.

Because of its immense popularity, I will be writing my code examples to specifically targeting **Express** applications. This should cover the vast majority of Node.js applications out there today, but many of the concepts we will cover apply to any platform.

Introduction to Express

Express is a minimal web development framework for Node.js that focuses on remaining small, yet robust. It is built on top of another framework called **Connect**, which is a platform for writing HTTP servers with small plugins known as middleware.

The architecture of Connect and Express allows you to use only what you require, and nothing else. This works very nicely into the security discussion, as you aren't incorporating lots of functionality that you don't use, which leaves the doors open for security vulnerabilities that may go unchecked.

Connect is bundled with over 20 commonly used middleware, adding capabilities, such as logging, sessions, cookie parsing, request body parsing, and more. While defining a Connect or Express app, you simply add the middleware that you intend to use as shown in the following code:

```
var connect = require("connect"),
    app = connect(); // create the application
```

```
// add a favicon for browsers
app.use(connect.favicon());

// require a simple username/password to access
app.use(connect.basicAuth("username", "password"));

// this middleware simply responds with "Hello World"
to every request
// that isn't responded to by previous middleware
(i.e. favicon)
app.use(function (req, res) {
    res.end("Hello World");
});

// app is a thin wrapper around Node's http.Server
// so many of the same methods are available
app.listen(3000);

console.log("Server available at
http://localhost:3000/");
```

Here, we are creating an application with three middleware: `favicon`, `basicAuth`, and a custom one of our own. The first two are provided by Connect, and they can each take configuration to specify their exact behavior.

Tip

Middleware is always executed in the order it was attached, which is something to keep in mind while you are determining what and when to attach.

Connect uses continuation-passing style, which means that each middleware function is given control, and must pass control to the next middleware in the continuation when it has completed. In terms of our application here, each middleware is given the request and response object and has full control over the life cycle of the request.

Since they are executed in order, let us examine how a request/response cycle operates for this application. Since middleware has full control, it can take one of the following three main courses of action:

- Respond to the request outright, ending the continuation
- Modify the request or response object for other middleware in the continuation
- Do nothing and simply initiate the next layer of middleware

Luckily for us, we have examples of all the three right here! First, when an application comes into this server, it is run through the `favicon` middleware. It checks the **uniform resource identifier** (**URI**), and if it matches `/favicon.ico`, it responds with a `favicon` icon for the browser. If the URI does not match, it simply passes over to the next middleware.

Next up, if the request proceeds, is the `basicAuth` middleware. This prompts the user to provide a username and password combination using **HTTP Basic Authentication** . If the user does not provide the correct credentials, the server responds with **401 (Unauthorized)** and ends the request. If the user successfully provides the correct username and password, the request object is modified to include the user's information and then the next middleware is initiated.

Last up is our custom middleware, which is probably the simplest one you could have. All it does is send **Hello World** as the response body. This means that no matter what URI we request (other than `/favicon.ico` of course), and as long as we provide the correct credentials, we will see **Hello World**.

Now that you have a basic understanding of how middleware works, let's move on to Express, and what it adds to Connect. Express adds routing, HTTP helpers, a view system, content negotiation, and other features using the Connect system. In fact, an Express app looks very similar to a Connect app as shown in the following code:

```
var express = require('express'),
    app = express();

app.use(express.favicon());
app.use(express.basicAuth("username", "password"));

app.get("/", function (req, res) {
    res.send('Hello World');
});
```

```
app.listen(3000);
```

Express automatically includes the Connect middleware within its own namespace, so you can use them without needing to explicitly require Connect. In addition, it adds some powerful features of its own, notably the routing feature we are using here.

Express was heavily inspired by the **Sinatra** web framework for **Ruby** . Each HTTP verb (`GET`, `POST`, and so on) has a corresponding function on the app object. Here, we are saying that an HTTP `GET` request for the URL `/` will send **Hello World**. Any other URL will get a **404 (Not Found)** error, except `/favicon.ico`, which is covered by the favicon middleware.

Express is minimalistic and remains out of your way to develop your application as you see fit. It doesn't lock you into an MVC framework, or a particular view engine, and allows you to include whatever npm modules you like to power your application.

Authentication

Authentication is a process of determining that a user is who they claim to be, when they are attempting to perform some action through your application. There are many ways to accomplish this, and I will cover some of the more common ones here. With a few exceptions, my examples will boil down to a couple of available npm modules. You are more than welcome to use others to accomplish the same goals.

HTTP Basic Authentication

The first is the **HTTP Basic Authentication**, and it is one of the simplest techniques available. It allows a username and password to be submitted along with an HTTP request, and allows the server to restrict access if the expected credentials are not sent.

While using a web browser, a page that requires the HTTP Basic Authentication will prompt the user with a dialog box asking for their username and password. After the user enters their information, the browser typically stores those credentials for a set period of time, rather than constantly prompting the user on each page.

The main advantage of this method is that it is very simple to implement. In fact, it can be done in as little as one line with Connect. In addition, this method is completely stateless and requires no out-of-band information with the request.

There are a number of important disadvantages, first of which is that it is not confidential. In other words, a basic HTTP request includes the username and password in plain text. Technically it is encoded as `base64`, but that is not an encryption method. As a result, this technique must be combined with some sort of encryption, such as HTTPS. Otherwise, the request can be intercepted by packet sniffers, and the credentials are no longer secret.

Also, the efficiency of this method is less than ideal. When a request is made for a page that requires the HTTP Basic Authentication, the server

effectively has to process that first request twice. On the first attempt, the request is denied, and the user needs to supply their credentials. On the second attempt, the credentials are sent with the request itself, and the server has to process the authentication again. Depending on how the username and password are validated, this can be an unacceptable delay that is incurred for each request.

In addition, there is no implemented way for browsers to log out while using this method, aside from closing the browser itself. The credentials are stored by the browser, and the user is not prompted to control how long it is stored, or when it should expire. To my understanding, only Internet Explorer provides such a feature, but it requires JavaScript in order to be triggered.

Last, as a developer, you have no control over the appearance of the login screen; it is entirely up to the browser. While this could boil down to simple aesthetics, it could be argued that it is more secure than a custom solution. If you desire to implement it, it is very easy to do so. One of the bundled middleware that Connect (and by extension, Express) affords is for this very purpose. It is called the `basicAuth` middleware, and it can be configured in several ways.

> ## Tip
>
> While using middleware, remember the order is very important! Make sure to place your authentication middleware early in the chain so you are authenticating all your requests, and not running unnecessary processing before verifying your user's identity.

First, you can simply provide a single username and password to the middleware, giving you a single valid set of credentials for your application as follows:

```
app.use(express.basicAuth("admin", "123456"));
```

Here, we have set up our application to require the username `"admin"` and the password `"123456"` via the HTTP Basic Authentication. This is

the simplest method of adding this authentication method.

A more advanced usage is to provide a synchronous callback function that can perform a slightly more sophisticated authentication scheme, for example, you can include a JavaScript object with username and password combinations that you can use to perform an in-memory lookup. This is illustrated in the following code:

```
var users = {
    // username: "password"
    admin: "password",
    user: "123456"
};

app.use(express.basicAuth(function (user, pass) {
    return users.hasOwnProperty(user) && users[user]
=== pass;
}));
```

We have set up `basicAuth` to check our `users` object for a corresponding username and password combination that is valid. If the callback function returns `true`, the authentication was successful. Otherwise, the authentication fails and the server responds appropriately.

Both of the methods we just used require some sort of hardcoding of credentials within our application's source code. The last method is more than likely the method you will employ if you use the HTTP Basic Authentication. This is asynchronous callback verification. This allows you to validate the request against some external source, such as a text file or database. Refer the following code:

```
app.use(express.basicAuth(function (user, pass, done)
{
    User.authenticate({ username: user, password: pass
}, done);
}));
```

In this example, we have a similar configuration in that we are using a function parameter. This function, unlike the previous example, has three arguments. It receives the username and password as before, but also receives a callback function that it needs to execute, when it has finished

validating the credentials. For the sake of brevity, I have not included specific implementation details.

The point is that you can perform the action asynchronously, and the callback function takes two parameters of its own. In Node.js fashion, the first parameter is an `Error` object, if the authentication fails. The second parameter is the user's information that will be added to `req.user` by the middleware, allowing the user's information to be accessed by later middleware functions.

After all is said and done, HTTP Basic Authentication is likely to be insufficient for most applications. Next, we will discuss **HTTP Digest Authentication** , which was originally designed to be the successor to HTTP Basic Authentication.

HTTP Digest Authentication

HTTP Digest Authentication aims to be more secure than the HTTP Basic Authentication by not sending the credentials as plain text. Instead, it employs the **MD5** one-way hashing algorithm to encrypt the user's authentication information. It is worth noting that MD5 is no longer considered a safe algorithm, which is one strike against this particular mechanism.

I am including this explanation simply for the sake of completeness. It is not popular and seldom recommended for use today, so I will not include any further details or examples.

It operates in the same way as the HTTP Basic Authentication in several ways. First, the initial request by the client is rejected when authentication is required, and the server indicates that the client needs to use the HTTP Digest Authentication. The client computes a hash of the user's credentials and the server's authentication realm. There are optional features available according to the specification for improving the hashing algorithms and preventing hijacking by malicious agents.

The one advantage that the HTTP Digest Authentication has is that the password is not transmitted over the network in plain text. This authentication method was devised in an era, where running HTTPS/SSL

for all network transactions was prohibitively expensive both in terms of money and processing power. Now that era has passed, and you should be using HTTPS consistently through your application. With that being the case, the advantages of the HTTP Digest Authentication over the HTTP Basic Authentication are practically nonexistent.

Introducing Passport.js

Now, I will be introducing a project that is a very popular authentication layer for Connect and Express applications. The project is Passport.js (http://passportjs.org/), and it is actually a collection of modules that aim to provide a consistent API for authenticating, using many different methods and providers. The rest of the examples for this section will use the Passport.js API, and I will explain some of the more common protocols along the way.

To use Passport.js in your application, you will need to configure the following three pieces:

1. Authentication strategies
2. Application middleware
3. Sessions (optional)

Passport.js uses the term "strategies" to refer to a method of authenticating a request. This could be a username and password, even third-party authentication, such as OpenID, or OAuth. This is the first thing you will configure, and it will depend on what methods of authentication you choose to support.

As a starting example, we'll look at the local strategy, where you take an HTTP `POST` request with a username and password in the body to authenticate against your own data store as shown in the following code:

```
// module dependencies
var passport = require("passport"),
    LocalStrategy = require("passport-
local").Strategy;

// LocalStrategy means we perform the authentication
ourselves
```

```
passport.use(new LocalStrategy(
    // this callback function performs the
authentication check
    function (username, password, done) {
        // this is just a mock API call
        User.findOne({ username: username }, function
(err, user) {
            // if a fatal error of some sort occurred,
pass that along
            if (err) {
                done(err);
            // if we don't find a valid user
            } else if (!user ||
!user.validPassword(password)) {
                done(null, false, { message:
"Incorrect username and password combination." });

            // otherwise, this was a successful
authentication
            } else {
                done(null, user);
            }
        });
    }
));
```

For the sake of simplicity, this does not wire into our application, this just demonstrates Passport.js middleware's API. What we are doing here is configuring a local strategy. This strategy takes a single verify callback that has three arguments: the username, password, and a callback function to be called once the authentication is complete. (Passport.js handles extracting the username and password from the `POST` request) The callback function takes three arguments of its own: an `Error` object (if applicable), the user's information (if applicable, false if the authentication fails), and an options hash.

In this case, the verify callback calls some sort of user API (the specifics of that are not important) to find a user matching the supplied username, then it proceeds with that data into the following checks:

1. If a fatal error occurs (such as the database is down, or the network is disconnected), then the callback is issued with that `Error` object as its only argument, which will be passed outside of Passport.js to be handled by your application.

2. If that username does not exist, or the password is invalid, then the callback is issued with null as the first argument (since no error occurred), `false` as its second argument (since the authentication itself failed), and an object with a single `message` property that we can use to display a message to the user (this third argument is optional).
3. If the user passes these checks, then the authentication was successful. The callback is issued with `null` first and the user's information object second.

The use of a callback in this fashion allows Passport.js to remain completely unaware of the underlying implementation. Now, let's move onto the middleware configuration step. Passport.js was specifically designed to use in Connect and Express applications, but it will work in anything that uses the same middleware style.

After configuring Passport.js and your strategies, you will need to attach at least one middleware to initialize Passport.js in your application as shown in the following code:

```
var express = require("express"),
    app = express();

// application middleware
app.use(express.cookieParser());
app.use(express.bodyParser());
app.use(express.session({ secret: "long random string
… " }));

// initialize passport
app.use(passport.initialize());
app.use(passport.session()); // optional session
support

// more application middleware
app.use(app.router);
```

This is a basic Express application, and we are attaching two Passport-related middleware: the initialization and the optional session support. Remember, the order is important, so you to initialize Passport.js after middleware like `bodyParser` and `session`, but before your application router.

The session-support middleware is optional, but recommended for most applications, as it is a very common use case, and it must be attached after Express' own `session` middleware. Last, we will configure the session support itself as shown in the following code:

```
passport.serializeUser(function (user, done) {
    // only store the user's ID in the session (to
keep it light)
    done(null, user.id);
});

passport.deserializeUser(function (id, done) {
    // we can retrieve the user's information based on
the ID
    User.findById(id, function (err, user) {
        done(err, user);
    });
});
```

Storing all of the available user data, especially as the number of concurrent users increases, can be costly. As a result, Passport.js gives developers a way to configure what is stored into the session, as well as the ability to retrieve the user's data for a single request (rather than holding it constantly in memory). This is by no means required, as using a shared database to store your session information can alleviate this problem.

The `serializeUser` function in the preceding example receives a callback that is executed, when the session is being initialized. Here, we are storing only the user's ID into the session, keeping it as light as possible, while still giving us the information we need to find their information later.

The corresponding `deserializeUser` function is called on each subsequent request, and adds the corresponding user's data to the request object. In this case, we are using a generic API to find a user, based on their ID, and issuing the callback with that data.

As you can see, configuring and using Passport.js is easy and it fits right into the Connect and Express methodology. There are over 120 strategies available for Passport.js, and you can find much more documentation and examples on their website (http://passportjs.org/).

OpenID

OpenID is an open standard for authentication on the Web by the use of a third-party service. The aim is to allow users to have a single identity on the Web that they can then use with many applications, rather than needing to register with each individual application. OpenID has no central authority, each provider is independent, and the user may choose any provider that he trusts. There are many major providers out there today, including: Google, Yahoo!, PayPal, and many others.

The OpenID authentication process operates something like this (this is a simplified explanation): a user is presented with an OpenID login form by a consumer. The user enters their provider's URL. The consumer redirects the user to their provider, the provider authenticates the user, and asks the user what information, if any, should be shared with the consumer. The provider then redirects the user back to the consumer, and the consumer allows the user to use their service.

To include OpenID in your application, we will use the `passport-openid` module. This module is a first class module of the Passport.js project, and it gives you a strategy for implementing a generic OpenID authentication process. First, let's look at the following Passport.js configuration required:

```
var passport = require('passport'),
    OpenIDStrategy = require('passport-
openid').Strategy;

// configure the OpenID Strategy
passport.use(new OpenIDStrategy(
    {
        // the URL that the provider will redirect the
user to
        returnURL:
'http://www.example.com/auth/openid/return',
        // the realm should identify your application
to the User
        realm: 'http://www.example.com/'
    },
    // this verify callback has 2 arguments:
    // identifier: the ID for your user (who they
claim to be)
```

```
    // done: the callback to issue after you've looked
the user up
    function (id, done) {
        // this is a generic API, it could be any
async operation
        User.findOrCreate({ openId: id }, function
(err, user) {
            done(err, user);
        });
    }
));
```

We have included the `passport` and `passport-openid` modules, and have configured the OpenID strategy. The configuration object (first argument) has two required properties:

- `returnURL`: This is the URL that the OpenID provider will redirect the user back to in your application
- `realm`: This is what the provider will show to the user to identify your application

The second argument is the verify callback, which only takes two arguments:

- `identifier`: This is how the user identifies himself with your application
- `done`: This is the callback your application issues after looking up the user based on the identifier

Now, you will need to configure the Express routes that you need to process the login requests, as shown in the following code:

```
// this route accepts the user"s login request,
passport handles the redirect
// over to the Provider for authentication
app.post("/auth/openid",
passport.authenticate("openid"));

// the Provider will redirect back to this URL (based
on our earlier
// configuration of the strategy) and it will tell us
whether or not
// the authentication was successful
app.get("/auth/openid/return",
```

```
passport.authenticate("openid", {
    // if successful, we'll redirect the user to the
hame page
    successRedirect: "/",
    // otherwise, send back to the login page
    failureRedirect: "/login"
}));
```

We have two configured routes, the first one takes the user's login request via `POST`, and Passport.js takes care of redirecting the user to the provider. The provider has been configured to send the user back to the `returnURL`, which corresponds to the second route we have configured earlier.

Next, you will need an HTML form on your login page that `POST` to the route, we configured earlier. This is illustrated in the following code:

```
<form action="/auth/openid" method="post">
    <div>
        <label>OpenID:</label>
        <input type="text" name="openid_identifier"/>
<br/>
    </div>
    <div>
        <input type="submit" value="Sign In"/>
    </div>
</form>
```

The only required HTML input is one that has the name `"openid_identifier"`. Each strategy has its own requirements, so make sure to read the documentation for each one as you are implementing them.

What we have configured here is a basic implementation of OpenID authentication using Passport.js. Now, we will move onto configuring a basic OAuth implementation for authentication as well.

Where OpenID aims to allow your identity to be authenticated by a trusted third-party, OAuth aims to allow users to share information between different applications without needing to give up their credentials to each separate party. If you need shared data with another service in your application, it is likely that you will be consuming an OAuth API from

that particular service. If all you need is to verify an identity, OpenID will likely be the mechanism of choice for that service.

OAuth

OAuth allows a user to share resources from one application to another without needing to share their username and password with both services. In addition, it also has the added capability of giving limited access. This limitation can be time-based, where access is revoked after a certain amount of time elapses. It could also restrict access to only a particular set of data, and potentially give the user more control over what they decide to share.

This process works by using a few different sets of keys (three to be more precise). Each stage of the authorization process builds upon the previous set of keys to construct the keys for the next step. In addition, between each step the user is redirected between the other applications, ensuring that the user only gives each application the minimum amount of information needed. The explanation I will give here is simplified, and does not cover the more technical details about topics like encryption and signatures.

The best metaphor for what OAuth does is like a "valet key". Some luxury cars have a special key that is limited in access. What I mean is that this special key only allows the car to be driven for a short distance, and only allows the valet driver to access the car as long as they have that key. This is very similar to what OAuth accomplishes, it allows the owner to give temporary and limited access to a resource that they own, while never giving up full control of that resource.

There are usually three parties involved: a `client`, a `server`, and a `resource owner`. The client is going to be requesting resources from the server on behalf of the resource owner.

To use the same real-world example that the OAuth specification uses, imagine Jane has uploaded some personal photos to a photo-sharing site and wishes to have them printed by another online service.

In order for the print service (the client) to access the photos stored with

the photo service (the server), they will need approval from Jane (the resource owner). First, any client application will need to register themselves with any server application in order to obtain the first set of keys, the client keys. These keys are known by both the client and server, and allow the server to validate the client's identity first and foremost.

Jane is ready to get her photos printed, so she visits the print service to begin the process. She wishes to have her photos pulled from the photo service rather than needing to upload them to another service, so she tells the print service that she would like photos from the photo service to be used.

Now, the printer service sends their client keys to the photo service (through a secured HTTPS request) to retrieve a set of temporary keys. These keys are used to identify a specified authorization request throughout the various redirects that take place.

Once the temporary keys are retrieved, the print service redirects Jane to the photo service. While there, Jane needs to verify her identity through whatever methods the photo service uses. In addition, the photo service can present Jane with options to limit the duration and scope of the authorization.

Once this verification is complete, Jane is redirected back to the print service with the temporary tokens. She has authorized the print service access to the photo service, which now exchanges the temporary keys for the last set of keys, the token keys.

This "access token" can now be used by the print service to request information from the photo service under the parameters that Jane has allowed, and can be revoked at any time by Jane or the photo service. Rather than using the generic `passport-oauth` module in the following examples I will stick to the Facebook module that uses OAuth v2.0. I have chosen this path to avoid needing to show all the variations of OAuth in use today, since each implementation may have their own variations. In addition, the examples here will give enough of an introduction to Passport's API that you can apply the approach to any other provider.

First, we will need to install the `passport-facebook` module, and then we will configure the Passport.js strategy as shown in the following code:

```
var passport = require('passport'),
    FacebookStrategy = require('passport-
facebook').Strategy;

// configuring the Facebook strategy (OAuth v2.0)
passport.use(new FacebookStrategy(
    {
        // developers must register their application
with Facebook
        // this is where the ID/Secret are obtained
        clientID: FACEBOOK_APP_ID,
        clientSecret: FACEBOOK_APP_SECRET,

        // this is the URL that Facebook will redirect
the user to
        callbackURL:
"http://www.example.com/auth/facebook/callback"
    },

    // the verify callback has 4 arguments here:
    // accessToken: the token Facebook uses to verify
authentication
    // refreshToken: used to extend the lifetime of
the accessToken
    // profile: the user's shared information
    // done: the callback function
    function (accessToken, refreshToken, profile,
done) {
        // here is where your application connects the
2 accounts
        User.findOrCreate(..., function (err, user) {
            done(err, user);
        });
    }
));
```

In order to use Facebook authentication, you will need to create and register an application account with Facebook Developers (https://developers.facebook.com/). This will likely be a similar process for other services; you will need some sort of registration on their side in order to coordinate safely with their users. From there, you can obtain a `clientID` and a `clientSecret`, which you put into the preceding configuration. You will also need to specify a `callbackURL`, which behaves

very much like the OpenID `returnURL`.

Next, you will need to configure routes for your Express application as shown in the following code:

```
// redirects the User to Facebook for authentication
app.get("/auth/facebook",
passport.authenticate("facebook"));

// Facebook will redirect back to this URL based on
the strategy configuration
app.get("/auth/facebook/callback",
passport.authenticate("facebook", {
    successRedirect: "/",
    failureRedirect: "/login"
}));
```

This is very similar to the routes we set up for OpenID, but with one major difference. The initial route is not an HTML form `POST`; it is a simple HTTP `GET`. This means you can just set up a simple HTML anchor that will point them to this route as follows:

```
<a href="/auth/facebook">Login with Facebook</a>
```

Passport will send the user off to Facebook for authentication. When Facebook is finished, it will redirect back to the second route, where you can redirect the user as needed (just like the OpenID implementation).

Passport.js is a great API for abstracting all of your authentication needs, so dig into its API documentation (http://passportjs.org/) and leverage any combination of the over 120 strategies they have available.

Authorization

Authorization is determining what access a user has to the restricted resources or actions in your application. Authentication deals specifically with who the user is, whereas authorization assumes we know who they are and must determine what they can do. Express gives us an elegant way of adding authorization built right into our routes, which is usually the layer where authorization takes place.

What many do not realize at first about express routing is that you are able to pass multiple handlers while defining a route. Each of them behaves like any other middleware as shown in the following code:

```
function restrict(req, res, next) {
    if (req.user) {
        return next();
    } else {
        res.send(403); // Forbidden
    }
}

app.get("/restricted", restrict, function (req, res) {
    res.send("Hello, " + req.user);
});
```

Our restrict function checks for user data (assume it is set by our authentication layer), and if the user is valid, it allows the chain to proceed. If the user is not logged in, it will simply respond with **403 (Forbidden)**.

The point here is that you can use multiple route handlers as an opportunity to handle pre-conditions, such as checking the user's authentication status, their roles, or any other rules regarding access. Much of this is highly dependent on how you structure your application, and how you determine what the user has access to.

One of the more popular authorization schemes is role-based authorization. A user can have any number of roles, such as: `"member"`, `"moderator"` or `"admin"`. Each of these roles can be used to determine what access they have on a per-action basis.

```
// dummy user data
var users = [
    { id: 1, name: "dominic", role: "admin" },
    { id: 2, name: "matthew", role: "member" },
    { id: 3, name: "gabriel", role: "member" }
];

// middleware for loading a user based on a :user
param in the route
function loadUser(req, res, next) {
    req.userData = users[req.params.user];
    return next();
}

// middleware for restricting a route to only a
specified role name
function requireRole(role) {
    // returns a function, closure allows us to access
the role variable
    return function (req, res, next) {
        // check if the logged-in user's role is
correct
        if (req.user.role === role) {
            return next();
        } else {
            return next(new Error("Unauthorized"));
        }
    };
}

// this route only loads a user's data (so it is
loaded via middleware)
app.get("/users/:user", loadUser, function (req, res)
{
    res.send(req.user.name);
});

// this route can only be called upon by an admin
app.del("/users/:user", requireRole("admin"),
loadUser, function (req, res) {
    res.send("User deleted");
});
```

In the preceding code, we have a list of available users. Assuming we have an authentication layer in place that loads a user profile data when logged in, let's look at the two middleware, we have defined.

First, `loadUser` is a simple middleware function that loads the user for the specified route (this may be a different user from the logged in user). Here, we just have a hard-coded list, but it could be a database call that we make asynchronously.

Second, the `requireRole` middleware is a bit sophisticated if you are not familiar with closure or first-class functions. What we are doing here is returning the middleware function, rather than simply using a named one. Through closure, we have access to the `role` argument inside the returned function. This middleware function ensures that the authenticated user has the role we are requiring.

We have two routes, the first (showing user data) is public, so it simply loads the user data via middleware and does no authorization check. The second route (deleting a user) requires that the authenticated user is an admin. If that check passes, the user's data is loaded and the route proceeds as expected.

There are many authorization methods available to you, with many good modules to pick from. Role-based authorization, as we have demonstrated here, is easy to implement in Express and it's generally easy to understand logically. As with authentication, your implementation depends on how you end up structuring your application. My main intent here is to define authorization and show you some examples to help you keep that mechanism as distinct as possible from the rest of your application logic.

Security logging

Another important aspect of security is logging, or recording various events within your application so that they can be analyzed for anomalies. These anomalies could be reviewed to detect places where attackers are attempting to bypass your security methods, and by detecting these activities before an actual intrusion, further steps can be taken to mitigate those risks. Beyond just security, logging can also help to detect cases in your program that cause problems for your users, and allow you to more easily reproduce and fix those problems.

Your specific application and environment will be what drives your logging methods. By methods, I mean how your logs are recorded and stored, such as the use of flat files in your filesystem, using some sort of database or even using third-party logging services. While these may differ greatly from project to project, the types of events recorded and the related information to save should be fairly consistent across the board.

The **Open Web Application Security Project** (**OWASP**) has a great guide for determining a logging strategy on their website (visit https://www.owasp.org/index.php/Logging_Cheat_Sheet for further information). They recommend the following recording logs for these specific types of events:

- Input validation failures (for example, protocol violations, unacceptable encodings, invalid parameter names, and values)
- Output validation failures (for example, database record set mismatch and invalid data encoding)
- Authentication successes and failures
- Authorization failures
- Session management failures (for example, cookie session identification value modification)
- Application errors and system events (for example, syntax and runtime errors, connectivity problems, performance issues, third party service error messages, file system errors, file upload virus detection, and configuration changes
- Application and related systems start-ups and shut-downs, and logging initialization (starting and stopping)

- Use of higher-risk functionality (for example, network connections, addition or deletion of users, changes to privileges, assigning users to tokens, adding or deleting tokens, use of administrative privileges, access by application administrators, access to payment cardholder data, use of data encrypting keys, key changes, creation and deletion of system-level objects, data import and export including screen-based reports, and submission of user-generated content especially file uploads)
- Legal and other opt-ins (for example, permissions for mobile phone capabilities, terms of use, terms and conditions, personal data usage consent, and permission to receive marketing communications)

In addition to their recommendations, OWASP also presents the following events as optional:

- Sequencing failure
- Excessive use
- Data changes
- Fraud and other criminal activities
- Suspicious, unacceptable, or unexpected behavior
- Modifications to configuration
- Application code file and/or memory changes

While determining what data to store for logs, OWASP recommends avoiding the following types of data:

- Application source code
- Session identification values (consider replacing with a hashed value if needed to track session specific events)
- Access tokens
- Sensitive personal data and some forms of personally identifiable information (PII)
- Authentication passwords
- Database connection strings
- Encryption keys
- Bank account or payment card holder data
- Data of a higher security classification than the logging system is allowed to store
- Commercially-sensitive information

- Information it is illegal to collect in the relevant jurisdiction
- Information a user has opted out of collection, or not consented to, for example, use of do not track, or where consent to collect has expired

In some cases, the following information can be useful during investigations, but should be carefully reviewed before including it in application logs:

- File paths
- Database connection strings
- Internal network names and addresses
- Non sensitive personal data (for example, personal names, telephone numbers, e-mail addresses)

Because each application and environment is different, the approaches logging can be equally diverse. The npm module we will look at here aims to provide a consistent API across many different methods, in addition to allowing you to use more than one at a time depending on the context.

The winston module (https://github.com/flatiron/winston) provides a clean and easy to use API for writing logs. In addition, it supports many methods of logging, including the capability for adding your own custom transports. A transport can be described as the storage or display mechanism for a given set of logs.

The `winston` module has built-in transports (as known as core modules) for logging to the console, logging to a file and sending logs over HTTP. Beyond the core modules, there are officially supported modules for transports, such as `CouchDB`, `Redis`, `MongoDB`, `Riak`, and `Loggly`. Lastly, there is a vibrant community using the `winston` API as well, with over 23 different custom transports out there today, including an e-mail transport and various cloud services like Amazon's **SimpleDB** and **Simple Notification Service** (**SNS**). The point is, it is likely that whatever transport you may require for your logging, there may be a module already available, and of course you are always able to write your own as well.

To get started with `winston`, install in via npm and you can use it right away using the "default logger" as shown in the following code:

```
var winston = require('winston');
winston.log("info", "Hello World");
winston.info("Hello Again");
```

This is by far the easiest way to get started quickly with `winston`, but it only uses the console transport by default. While the default logger can be extended with more transports and configuration, the more flexible approach is to create your own instances of `winston` that you can use in various contexts within your application. This can be done as shown in the following code:

```
var winston = require("winston");

var logger = new (winston.Logger)({
    transports: [
        new (winston.transports.Console)(),
        new (winston.transports.File)({ filename:
'somefile.log' })
    ]
});
```

Within your application code, I typically place the boilerplate code for such modules in their own file. From there, you can export a pre-configured object that can be imported and used throughout your application, for example, you can create a file called `lib/logger.js` that looks like the following:

```
var path = require("path"),
    winston = require("winston");

module.exports = new (winston.Logger)({
    transports: [
        // only logs errors to the console
        new (winston.transports.Console)({
            level: "error"
        }),
        // all logs will be saved to this app.log file
        new (winston.transports.File)({
            filename: path.resolve(__dirname,
"../logs/app.log")
```

```
        }),
        // only errors will be saved to errors.log,
and we can examine
        // to app.log for more context and details if
needed.
        new (winston.transports.File)({
            level: "error",
            filename: path.resolve(__dirname,
"../logs/errors.log")
        })
    ]
});
```

Then within other parts of your application, you can include the logger and use it easily as follows:

```
var logger = require("./lib/logger");
logger.log("info", "Hello World");
logger.info("Hello Again");
```

In addition, `winston` also includes other advanced features, such as custom log levels, additional transport configuration, and dealing with unhandled exceptions. Also, `winston` is not the only logging API available for Node.js, there are other alternatives that you can consider depending on your own needs. This is not even to mention developing your own custom solution to give you complete control.

Error handling

One of the important aspects of any application is how it handles errors. As mentioned before, uncaught exceptions can crash your application, so being able to handle errors properly is an important part of your development cycle.

Responding to errors within your own application is the key, so refer back to Chapter 2, *General Considerations*, for a general introduction to how to deal with errors in Node.js. Here, we will deal specifically with Connect and Express.

First, do not throw errors directly in your route handlers. While Express is smart enough to try/catch errors directly on the route handler, this will not help you if you are performing some sort of asynchronous operation (this is the case most of the time), as shown in the following code:

```
app.get("/throw/now", function (req, res) {
    // Express wraps the route handler invocation in
try/catch, so
    // this will be handled without crashing the
server
    throw new Error("I will not crash the server;
});

app.get("/throw/async", function (req, res) {
    // However, when performing some asynchronous
operation
    // time) then you will lose your server if you
throw
    setTimeout(function () {
        // try/catch does not work on
callbacks/asynchronous code!
        throw new Error("I WILL crash the server");
    }, 100);
});
```

Both of the preceding handlers throw exceptions. As mentioned before, Express will execute your handler in a `try/catch` to handle exceptions thrown in the handler itself. However, asynchronous code, such as the second route does not work with typical try/catch and end up becoming

uncaught exceptions. In short, don't use `throw` while handling errors!

In addition to the request and response objects passed to your handlers, there is a third argument you can utilize like any other middleware. This is commonly named the "next" callback, and you use it like you would in middleware, to pass along to the next item in the continuation. This is illustrated in the following code:

```
app.get("/next", function (req, res, next) {
    // this is the correct way to handle errors, as
Express will
    // delegate the error to special middleware
    return next(new Error("I'm passed to Express"));
});
```

If you execute the next callback with an `Error` object as the first argument, then Connect will take that error and delegate to any error-handling middleware that you have configured. When you set up a middleware that takes four arguments, it is always treated as error-handling middleware.

```
// 4 arguments tells Express that the middleware is
for errors
// you can have more than 1 if necessary
app.use(function (err, req, res, next) {
    console.trace();
    console.error(err);

    // just responds with a 500 status code and the
error message
    res.send(500, err.message);
});
```

This special error-handling middleware goes last in your application stack, and you are able to set up more than one if that is necessary. You can pass along control via next like any other middleware, in case you set up multiple layers of error-handling, for example, one layer can send an e-mail, one can log to a file, and one (the last one) can send a response to the user.

Connect also has a special middleware that you can utilize to deal with errors without needing to hard code your own middleware. This is the

`errorHandler` middleware, and it will automatically respond with either plain text, JSON, or HTML (depending on the client's headers) when an error occurs. This middleware is expressed as follows:

```
app.use(express.errorHandler());
```

Typically, this helper is just for development use, as your production application likely has more work to do when dealing with errors you need to be in complete control of.

In summary, always use the "next" callback function in your route handlers to communicate errors, never use throw. In addition, always configure some sort of error-handling middleware by adding a middleware function with four arguments. Use the built-in handler from Connect for development, and have your own place for production.

Summary

In this chapter, we examined high-level security considerations that apply to applications in general, such as authentication, authorization, and error-handling. In the next chapter, we will examine vulnerabilities that appear during the request phase of your applications.

Chapter 4. Request Layer Considerations

Some vulnerabilities appear at the request phase of your application. As mentioned before, Node.js does little for you by default, leaving you with complete freedom to craft a server that meets your needs.

Limiting the request size

One major request-handling feature that is commonly left out of Node.js applications is size limits. **Express** (optionally) handles buffering of request body data and parsing that request body into some meaningful data structure. While the request is still being fulfilled, the entire content of that body is in memory. If you place no limits, malicious users have a number of ways to affect your system, such as exhausting memory limits, and uploading files that take up unnecessary disk space.

Depending on your needs, you will need to determine a reasonable limit for your application. While your needs may differ, you should always set some sort of limit, Connect and Express exposes a middleware just for this purpose, called limit:

```
app.use(express.limit("5mb"));
```

This middleware needs to be added early in the stack, otherwise it won't be caught until it's too late. It takes a single piece of configuration, which is the upper limit on the request size. If you send a number, it will be translated as a number of bytes. You can also send a more readable string, such as `"5mb"` or `"1gb"`.

This middleware responds with a **413 (Request Entity Too Large)** error to be thrown, if the limit is exceeded. First, the `Content-Length` header of the request is checked, and if it is too large it denies the request outright. Of course, the header could be faked or even absent, so the middleware also monitors the incoming data and triggers an error if the actual request

body size reaches the limit.

The `bodyParser` middleware is used to parse incoming request bodies for particular content types. In fact, the `bodyParser` middleware specifically is just short hand for three different middlewares namely, `json`, `urlencoded`, and `multipart`. Each of these corresponds to a different content-type. Setting an absolute size via the limit middleware is helpful, but not always enough. Some request bodies should be limited differently than others.

For example, you may wish to allow file uploads that are up to 100 MB. However, that same amount of JSON will bring your application to a halt, while the `JSON.parse()` function runs, since it is a blocking operation. As a result, it is highly recommended to set a much smaller limit on request bodies other than multipart (since it deals with file uploads).

Therefore, I would recommend avoiding the `bodyParser` middleware, in order to be more explicit, and allow you to set different limits for each of the sub-middlewares.

```
// module dependencies
var express = require("express"),
    app = express();

// limiting the allowed size of request bodies (by
content-type)
app.use(express.urlencoded({ limit: "1kb" })); //
application/x-www-form-urlencoded
app.use(express.json({ limit: "1kb" }));       //
application/json
app.use(express.multipart({ limit: "5mb" }));  //
multipart/form-data
app.use(express.limit("2kb"));                 //
everything else
```

Tip

While setting different limits for different content types like we are talking about here, the results could be unexpected if you are not careful about the order you choose for your middleware.

If the limit middleware is used first, it will cause the other middlewares

to ignore their own size limits. Make sure that you place the global limit middleware last, so it acts as a catch-all for any other content type, and not dealt with by the `bodyParser` middleware family.

Using streams instead of buffering

Node.js includes a module called **streams**, which contains the implementation used widely throughout Node.js platform's own core modules. A stream is a lot like a Unix pipe, they can be read from, written to, or even both depending on the context. I won't go into great detail here, but streams are one of Node.js killer features, and you should be using them as much as possible in your applications and any npm modules you publish.

If you are implementing more of a RESTful API, that accepts a file upload as a `PUT` request, for example, use streams in your request handler. The following code shows an inefficient way to handle putting a request body into a file:

```
var fs = require("fs");

// handle a PUT request against /file/:name
app.put("/file/:name", function (req, res, next) {
    var data = "", // data buffer
        filename = req.params.name; // the URL
parameter

    req.on("data", function (chunk) {
        data += chunk; // each data event appends to
the buffer
    });

    req.on("end", function () {
        // write the buffered data to a file
        fs.writeFile(filename, data, function (err) {
            if (err) return next(err); // handle a
write error

            res.send("Upload Successful"); // success
message
        });
    });
```

```
});
```

Here, we are buffering the entire request body into memory, before writing it to disk. At small sizes, this is not a problem, but an attacker could simultaneously send many large request bodies, and you're putting yourself in unnecessary risk by buffering. In Node.js, with streams at your disposal, this is the long way to do it (thank goodness the shorter way is also the best way!).

The following code is an example of the same request, only using a stream to pipe the data to the destination:

```
var fs = require("fs");

// handle a PUT request against /file/:name
app.put("/file/:name", function (req, res, next) {
    var filename = req.params.name, // the URL
parameter
        // open a writable stream for our uploaded
data
        destination = fs.createWriteStream(filename);

    // if our destination could not be written to,
throw an error
    destination.on("error", next);

    req.pipe(destination).on("end", function () {
        res.send("Upload Successful"); // success
message
    });
});
```

Our example here sets up a writeable stream that represents the destination of the uploaded data. Rather than buffering the entire request body into memory, the data will simply be piped into that file, as it becomes available. It should be noted that this example does not properly filter the user input; this was entirely to stay focused on the topic of the example and should not be applied directly to production code.

Streams are a proven and effective pattern for dealing with data in numerous contexts, and leverage the event-driven model of Node.js to its full potential.

When dealing with many simultaneous users, especially with unforeseen bursts of traffic, it's important to be ready for disaster scenarios, where the load becomes too much for your server to handle. This is also applicable in mitigating **Denial of Service** (**DoS**) attacks that attempt to flood your server with more requests than it could ever possibly handle, bringing it down completely (or just slowing it down to a crawl) for every other user.

Monitoring the event loop's responsiveness

Building a server that doesn't just melt under heavy load can be done. One useful pattern is to monitor the event loop's responsiveness, and deny some requests right away, if the server is just under too much load to respond quickly. One module out there, called node-toobusy (https://github.com/lloyd/node-toobusy) does just that.

Once initialized, toobusy polls the event loop, and watches for lag or requests to the event loop that takes longer than expected. In your application, you set up a middleware layer that simply queries the monitor to determine whether or not to add to the server's current processing queue. If the server is too busy, it will respond with a **503**, (**Server Currently Unavailable**) rather than taking on more load that it is able to satisfy. Instead of crashing your server, this pattern allows you to continue serving as many requests as possible, as shown in the following code:

```
var toobusy = require("toobusy"),
    express = require("express"),
    app = express();

// middleware which blocks requests when we're too
busy
app.use(function(req, res, next) {
    if (toobusy()) {
        res.send(503, "I'm busy right now, sorry.");
    } else {
        next();
    }
});

app.get("/", function(req, res) {
    // each request blocks the event loop
    var start = (new Date()).getTime(), now;
    while (((new Date()).getTime() - start) <= 5000);
// run for 5 seconds
    res.send("Hello World");
});
```

```
var server = app.listen(3000);
process.on("SIGINT", function() {
    server.close();
    // calling .shutdown allows your process to exit
normally
    toobusy.shutdown();
    process.exit();
});
```

The preceding sample was found on node toobusy's github page. It sets up a simple server with a middleware employing the toobusy module. It also sets up a single route that blocks the event loop by running for five straight seconds. If a number of simultaneous requests that block the event loop for long enough come in, the server will start responding with a **503 (Server Currently Unavailable)** error, rather than taking on more than it should. Lastly, this also includes a graceful shutdown for the process.

This example also demonstrates a very important point about the event loop in Node.js that is worth repeating. The contract made between your code and the event loop scheduler is that all code should execute quickly, to keep from blocking the event loop for other code. This means to avoid CPU-intensive calculations in your application code, unlike the preceding example, which blocks the CPU during its while-loop iteration.

Node.js works best when your application is primarily I/O-bound, so CPU-intensive operations, such as complex calculations or very large data-set iterations should be avoided. If your system requires such operations, consider spawning the blocking portions off as separate processes to keep from hogging your application's event loop.

There are a couple methods to accomplish this, such as using the HTML5 Web Worker API for node (https://github.com/pgriess/node-webworker). In addition, a more bare-metal approach is to utilize Node's `child_process` module in conjunction with **Inter-Process Communication** (**IPC**).The IPC specifics on this are potentially heavily dependent on your platform and architecture, which is beyond the scope of this discussion.

Cross-site Request Forgery

Cross-site Request Forgery (**CSRF**) is an attack vector that exploits the trust, an application has for a specific user's browser. A request is made on the user's behalf without their consent, allowing the application to perform some action under the assumption that the trusted user initiated the request, even though they have not.

There are a number of ways this can be accomplished. One example is that an HTML image tag (for example, an `<img>`) somehow injected into the page, legitimately or not, such as via XSS, a vulnerability we will discuss in the next chapter. The browser implicitly sends a request to the URL specified in the `src` attribute, and sends any cookies it has as a part of the HTTP request. Many applications that track a user's identity do so via cookies that contains some sort of session identifier, which to the server makes it appear as though the user is making the request.

Prevention is pretty straightforward; the most common approach is requiring a generated, user-specific token to be included with each request that modifies state. In fact, Connect already includes the `csrf` middleware for just this purpose.

It works by adding a generated token to the current user's session, which can be included in an HTML form, as a hidden input field or as a query-string value in any links with side-effects. When a later request is being handled, the middleware checks to ensure the value in the user's session, matches what was submitted with the request, which fails with a **403 (Forbidden)**, in the event of a mismatch.

```
var express = require("express"),
    app = express();

app.use(express.cookieParser()); // required for
session support
app.use(express.bodyParser());   // required by csrf
app.use(express.session({ secret: "secret goes here"
})); // required by csrf
app.use(express.csrf());

// landing page, just links to the 2 different sample
```

```
forms
app.get("/", function (req, res) {
    res.send('<a href="/valid">Valid</a> <a
href="/invalid">Invalid</a>')
});

// valid form, includes the required _csrf token in
the HTML Form (hidden input)
app.get("/valid", function (req, res) {
    var output = "";
    output += '<form method="post" action="/">'
    output += '<input type="hidden" name="_csrf"
value="' + req.csrfToken() + '">';
    output += '<input type="submit">';
    output += '</form>';
    res.send(output);
});
// invalid form, does not have the required token
// throws a "Forbidden" error when submitted
app.get("/invalid", function (req, res) {
    var output = "";
    output += '<form method="post" action="/">'
    output += '<input type="submit">';
    output += '</form>';
    res.send(output);
});

// POST target, redirects back to home if successful
app.post("/", function (req, res) {
    res.redirect("/");
});

app.listen(2500);
```

This example application has some defined middleware, namely the `bodyParser`, `cookieParser`, and `session`. These are all required by `csrf`, which is why they go first in the order. In addition, there are a few routes which are as follows:

- The homepage, which just provides links to both sample forms
- The form action/target, which simply redirects the user home on a successful submission
- The valid form, which includes the token as a hidden input and successfully submits it
- The invalid form, which does not include the token and consequently fails, when submitted (with a (**403 Forbidden)** HTTP response)

This method prevents an attacker from successfully making false requests, as the required token will be different for each form submission.

Input validation

While protecting against many attack vectors, such as XSS, which we will deal with in the next chapter, it is important to filter and sanitize your inputs as you receive them from the user. This occurs during the request phase of a web application, so we will address it here. The general rule of thumb is to always validate inputs and escape outputs.

A popular library for validating user input is node-validator (https://github.com/chriso/node-validator). This library is by no means the only option, but it is the one we will be using in our examples.

There are several goals of input validation, first of which is to verify that incoming user input matches the criteria of our application and its workflow; for example, you may want to ensure that a user submits a valid e-mail address. I am not referring to sending an e-mail for confirmation to test that the e-mail address is real, instead I am just talking about ensuring that they do not enter an erroneous value in the first place. Another example is to ensure that the number matches a particular range, such as being greater than zero.

Secondly, input filtering is meant to prevent bad data from making it into your system that could compromise another subsystem; for example, if you accept an input for a certain numeric input, which you then pass along to another subsystem for some additional processing, such as a report or some other remote API. If your users, intentionally or not, submit some other unexpected value, like a symbol or an alphabetic character, it could cause problems in future operations. In large part, computers are garbage-in, garbage-out, so we need to make sure we are careful with any user input.

Thirdly, as already mentioned briefly before, input filtering is a helpful (albeit incomplete) preventative measure against attacks like **Cross-Site Scripting** (**XSS**). XSS attacks in HTML, CSS, and JavaScript, there are big problems with access control, meaning that any script has the same access as every other one. This means that if an attacker can find a way to inject further code into your page, they will have a great degree of control, which is potentially harmful to your users. Input filtering can help

by removing malicious code that may be cleverly embedded in other user input.

In addition to the base node-validator library, there is also a middleware plugin (express-validator: https://github.com/ctavan/express-validator), made especially for Express.js, which we will be using for our examples.

Our first example will be a form that accepts a large variety of inputs, just to help demonstrate as much as possible. Consider the following HTML form:

```
<form method="post">
    <div>
        <label>Name</label>
        <input type="text" name="name">
    </div>
    <div>
        <label>Email</label>
        <input type="email" name="email">
    </div>
    <div>
        <label>Website</label>
        <input type="url" name="website">
    </div>
    <div>
        <label>Age</label>
        <input type="number" name="age">
    </div>
    <div>
        <label>Gender</label>
        <select name="gender">
            <option>-- choose --</option>
            <option value="M">Male</option>
            <option value="F">Female</option>
        </select>
    </div>

    <button type="submit">Validate</button>
</form>
```

This sample code sets up an HTML form with five fields: `name`, `e-mail`, `website`, `age`, and `gender`. A user can enter values in the provided inputs and `POST` to the same URL. While processing the `POST` request, we will validate the data and give some sort of response. The next code sample will be our application code:

```
// module dependencies
var express = require("express"),
    app = module.exports = express();

app.use(express.bodyParser());          // required
by csrf
app.use(require("express-validator")()); // the
validation middleware

// an HTML form to be validated
app.get("/", function (req, res) {
    res.sendfile(__dirname + "/views/validate-
input.html");
});

/**
 * Validates the input, will either:
 *  - sends a 403 Forbidden response in the event of
validation errors
 *  - send a 200 OK response if the data validates
successfully
 */
app.post("/", function (req, res, next) {
    // validation
    req.checkBody("name").notEmpty().is(/\w+/);
    req.checkBody("email").notEmpty().isEmail();
    req.checkBody("website").isUrl();
    req.checkBody("age").isInt().min(0).max(100);
    req.checkBody("gender").isIn([ "M", "F" ]);
    // filtering
    req.sanitize("name").trim();
    req.sanitize("email").trim();
    req.sanitize("age").toInt();

    var errors = req.validationErrors(true);

    if (errors) {
        res.json(403, {
            message: "There were validation errors",
            errors: errors
        });
    } else {
        res.json({
            name: req.param("name"),
            email: req.param("email"),
            website: req.param("website"),
            age: req.param("age"),
            gender: req.param("gender")
        });
```

```
    }
});
```

This example sets up a basic web server with only two routes, a `GET /` which just sends the HTML form, we showed earlier as the response. The second route is a `POST /` which takes the data submitted from the aforementioned form, and first validates it with the following rules:

Field	Rules
`name`	This field cannot be empty. It must match a regular expression (this one means it must be only alphabetic, numeric, whitespace, and a few select symbols).
`e-mail`	This must be a valid e-mail address.
`website`	This must be a valid URL.
`age`	This must be a number. It must be greater than or equal to 0. It must be less than or equal to 100.
`gender`	This must be either "M" or "F".

In addition to validating the input, it also performs some filtering and transforming before the output, according to the following rules:

Field	Rule
`name`	Trim leading and trailing whitespace.
`e-mail`	Trim leading and trailing whitespace.
`age`	Convert to an integer.

Depending on how the validation goes, it will either respond with a **403 (Forbidden)**, with the list of validation errors, or it will respond with a **200 (OK)** with the filtered input.

This should demonstrate that it is pretty straightforward to add input validation and filtering to your applications, and the rewards are well worthwhile. You can ensure that the data matches expected formats for your various workflows, and help preemptively protect against some attack vectors.

Summary

In this chapter, we specifically examined request vulnerabilities, and provided some ways to avoid and deal with those vulnerabilities. In the next chapter, we will look at the response phase of applications, and the vulnerabilities that appear there.

Chapter 5. Response Layer Vulnerabilities

The last interaction you will have with a user request is, of course, the response. The discussion here will focus on vulnerabilities and best practices for this portion of your application code. This will include **Cross-site Scripting** (**XSS**), some vectors for **Denial of Service** (**DoS**) attacks, and even HTTP headers that various browsers use for implementing specific security policies.

Cross-site Scripting (XSS)

Cross-site Scripting (XSS) is one of the more popular topics while dealing with web applications, as it is the default behavior of HTML/CSS/JavaScript in many respects. Specifically, XSS is an attack vector that is used to inject untrusted and likely malicious code into a web page. Usually, this is taken as an opportunity to inject JavaScript code into your page that now has access to just about anything the client has access to in that particular web page.

By default, JavaScript is executed in a global scope in the browser, including code that was injected by an untrusted source. This is the same behavior that your own, trusted code has, making it a dangerous vector with many possibilities. The malicious script could find the user's session ID (usually in a cookie), and use AJAX to send that information to someone that can then hijack the user's session.

The injection commonly comes from the user input that is not filtered or sanitized before being output to the browser. Consider the following example code:

```
var express = require("express"),
    app = express();

app.get("/", function (req, res) {
    var output = "";
    output += '<form action="/test">';
```

```
    output += '<input name="name" placeholder="enter a
name">';
    output += '</form>';

    res.send(output);
});

app.get("/test", function (req, res) {
    res.send("Hello, " + req.query.name);
});

app.listen(3000);
```

This script creates a server that simply sends an HTML form that is submitted (via `GET`) to another page. The second route simply outputs the user's input value to the browser.

If the user inputs their name (like Dominic) everything is well, and the user sees **"Hello, Dominic"** on the next page. However, what if the user enters something else, like raw HTML? In this case, it just outputs the HTML alongside our own HTML, and the browser can't tell the difference.

If you enter `<script>alert('hello!');</script>` in that text field instead, when you open the next page, you'll see **"Hello,"** and the browser will trigger an alert with **"hello!"** in the box. This is a harmless example, but this vulnerability has a huge potential for damage. These attacks are accomplished through what is known as untrusted data, which could be raw user input, information stored in a database, or accessed via a remote data source. The untrusted data is then used by your application to construct some sort of command that is then executed. The danger comes when the command is manipulated to perform some action that was not the original intent of the developers.

The prototypical example of this type of attack is a SQL injection, which is where the untrusted data is used to alter a SQL command. Consider the following code:

```
var sql = "SELECT * FROM users WHERE name = '" +
username + "'";
```

Assume that the username variable comes from the user input, and the

point is that it is an untrusted data as we have defined it. If the user enters something innocuous, like `'Dominic'`, then all is well, and the generated SQL looks like the following code:

```
SELECT * FROM users WHERE name = 'Dominic'
```

What if someone enters something less harmless, like: `'' OR 1=1`, then the generated SQL becomes like the following:

```
SELECT * FROM users WHERE name = '' OR 1=1
```

This changes the meaning of the query entirely, rather than restricting to one user with a matching name, now every row is returned. This could be even more disastrous, consider the value: `''; DROP TABLE users;`, it would generate a SQL like the following:

```
SELECT * FROM users WHERE name = ''; DROP TABLE users;
```

Without any additional access, the user has caused a devastating loss of data to our application, probably bringing the entire application down for all users.

As it turns out, XSS is another type of injection attack, and the web browser and the HTML, CSS, and JavaScript that they execute, are optimized for these types of attacks. There are many different contexts within each of these languages that we need to be aware of. Consider the following template:

```
<h2>User: <%= username %></h2>
```

With our untrusted data, we could easily cause trouble by injecting additional HTML into this value, such as `<script>alert('xss');</script>`, which would generate the following HTML code:

```
<h2>User: <script>alert('xss');</script></h2>
```

The solution here is to use an HTML escaping on any untrusted data added to the page in this context. This technique turns characters that

are important in HTML, such as angle brackets and quotes, into their corresponding HTML entity; preventing them from altering the structure of the HTML they are embedded within. The following table is an example of this conversion:

Character	**Entity**
Less than sign (<)	`<`
Greater than sign (>)	`>`
Double quote (")	`&quot;`
Single quote (')	`' (&apos;` is not a valid HTML and should be avoided)
Ampersand (&)	`&`
Forward slash (/)	`/`

This method of escaping makes it harder for an attacker to alter the structure of your HTML, making this a very important technique for securing your web pages. However, different contexts will require further escaping techniques that we will address shortly.

Tip

Many popular templating libraries include automatic HTML escaping by default, but some do not. This should be an important factor to you for choosing a template framework or library.

HTML attributes could be injected with other HTML meant to create a new context, such as closing the attribute and starting a new attribute. Further still, this injected HTML could be used to close the HTML tag, and inject more HTML in another context. Consider the following template:

```
<img height=<%= height %> src="...">
```

Consider the following injected value for height: `100 onload="javascript:alert('XSS');"`, which would generate the following HTML:

```
<img height=100 onload="javascript:alert('XSS');"
src="...">
```

The result is injected JavaScript code. HTML encoding as we used before is not enough in this particular context, as the preceding is still a perfectly valid HTML. In addition to HTML escaping as we mentioned before, you should require quotes around all HTML attributes, particularly when untrusted data is involved. To cover all cases, even unquoted attributes, you could encode all ASCII values below 256 to their HTML entity format or an available named entity like `&quot;`, if available).

HTML attributes that involve URLs, such as `href` and `src`, are another context altogether that require their own encoding. Consider the following template:

```
<a href="<%= url %>">Home Page</a>
```

If the user enters the following data: `javascript:alert('XSS');`, then the following HTML is generated:

```
<a href="javascript:alert('XSS');">Home Page</a>
```

An HTML encoding is not applicable here, as the preceding is a valid HTML markup. Instead, a fully-qualified URL should be checked for unexpected protocols. Here, we used `javascript:`, which gets the browser to execute arbitrary code, behaving like the `eval()` function. Lastly, the output should be escaped via the built-in JavaScript function called `encodeURI()`, which escapes characters that are invalid in URLs.

The last example I will show here is partial URLs within attributes like the ones mentioned previously. Using the following template:

```
<a href="/article?page=<%= nextPage %>">Next</a>
```

The `nextPage` variable is being used as a part of a URL, rather than being the URL itself. The `encodeURI()` function we mentioned earlier has a companion called `encodeURIComponent()`, which escapes more characters, because it is meant to encode a single query-string parameter.

Another common anti-pattern is injecting JSON data into a page directly to share data between the server and client while rendering a page. Consider the following template:

```
<script>
var clientData = <%= JSON.stringify(serverData); %>;
</script>
```

This particular technique, while convenient, can allow for XSS attacks as well. Let's assume the `serverData` object has a single property called `username` that reflects the current user's name. Let's also assume that this value is able to be set by the user without any sort of filtering between the user's input and the display on the page (which of course should not happen).

If the user changes his name to `</script><script>alert('XSS')</script>` then the output HTML would look like the following:

```
<script>
var clientData = {"username":"</script>
<script>alert('XSS');</script>"};
</script>
```

According to the HTML specification, a `</` character (even within a JavaScript string, as we have here) will be interpreted as a closed tag, and the attacker has just created a brand new script tag that, like any other script tag, has full control over the page.

Rather than simply trying to escape the JSON data directly, the best way to mitigate this problem is to inject your JSON data, using another method altogether:

```
<script id="serverData" type="application/json">
<%= html_escape(JSON.stringify(data)) %>
</script>

<script>
var dataElement =
document.getElementById("serverData");
var dataText = dataElement.textContent ||
dataElement.innerText; // unescapes the content of the
script
var data = JSON.parse(dataText);
 </script>
```

This method uses a script tag, with a predefined ID that we can use to retrieve it. When a browser encounters a script type it does not understand, it will simply not execute it, in addition to leaving it hidden from the user. The contents of this script tag will be an HTML-escaped version of our JSON, which ensures we have no context boundary crossing.

Next, we use another script (preferably in an external file, but by no means required) with the code that finds the script element we defined, and retrieves its text content. By using the `textContent/innerText` property instead of `innerHTML`, we get additional escaping that the browser performs for us, just in case. Lastly, we run the JSON data through `JSON.parse` to actually perform the JSON decoding.

While this method requires more fanfare, and is going to be a bit slower than the first example, it is going to be far more secure, which is a great trade-off to make.

These examples are by no means an exhaustive list, but they should illustrate the point that HTML, CSS, and JavaScript each have contexts that allow for various types of code injection. Never trust your user input, and make sure you use the appropriate escaping method depending on the context.

The **Open Web Application Security Project** (**OWASP**) is a foundation that maintains a wiki (http://www.owasp.org/) that specifically addresses security considerations for all web applications. They have articles on many attack vectors, including a more comprehensive checklist for preventing many more varieties of XSS attacks.

Denial of Service

A **Denial of Service** (**DoS**) attack can come in a variety of forms, but the main intent is to prevent users from having access to your application. One method is to flood your server with a large amount of requests, tying up your server's resources and preventing legitimate requests from being fulfilled.

Request flooding typically targets multithreaded servers, like **Apache**. This is because the process of spawning a new thread for each request gives an easy-to-reach upper limit on the number of simultaneous requests. With Node.js platform's event loop, this particular type of attack is not usually as effective, although that's not to say that it is impossible.

The event loop can still expose applications if used improperly, I cannot stress enough how important it is to understand how it works, while writing any Node.js application. The contract your application code has with the event loop is to always run as fast as possible. There is only one piece of your application running at once, so CPU-intensive can tie up resources as well. This applies in all cases, but I mention it in this chapter to specifically address your response handlers. Generally, receiving the request itself is less resource-intensive than performing the actions necessary to generate the appropriate response.

As mentioned earlier, use streams whenever possible, especially while dealing with network requests or the filesystem. Dealing with large blobs of data can be time-consuming depending on how you are processing that data, the use of streams can break those large operations into many small chunks, allowing other requests to be satisfied in the process.

Security-related HTTP headers

There are some HTTP headers available that can help add some security to our web applications. We will be looking at a module called **helmet**, which is written as a collection of Connect/Express middleware that adds these headers depending on your configuration. We will examine each of the middleware functions that helmet includes, as well as a brief explanation of their effects.

Content security policy

First, helmet supports setting headers for a newer security mechanism for HTML and web applications called **Content Security Policy** (**CSP**). XSS attacks circumvent the **Same-Origin Policy** (**SOP**) by using other methods to trick browsers into delivering harmful content.

For browsers that support this feature, you can restrict resources, such as images, frames, or fonts to be loaded via white-listed domains. This limits the impact of XSS attacks by hopefully preventing access to untrusted domains for loading malicious content.

CSP is communicated to a browser via one or more `Content-Security-Policy` HTTP headers, such as:

```
Content-Security-Policy: script-src 'self'
```

This header will instruct the browser to require that all scripts load from the current domain only. Any scripts the browser detects coming from any other domains will be blocked outright.

A CSP header is constructed as a list of directives separated by semicolons. An example of a header that implements multiple CSP restrictions looks like the following:

```
Content-Security-Policy: script-src 'self'; frame-src
'none'; object-src 'none'
```

This header instructs the browser to restrict scripts to only the current domain (like our previous example), and forbids the use of frames (including iframes) and objects altogether.

Each directive is named `*-src`, and it is followed by a space-separated list of either predefined keywords (which must be wrapped in quotes) or domain URLs.

The available keywords include the following:

- `'self'`: This restricts script to the current domain
- `'none'`: This restricts all domains (none can be loaded at all)
- `'unsafe-inline'`: This allows inline code (you are highly advised to avoid this, more discussion later)
- `'unsafe-eval'`: This allows text-to-JavaScript mechanisms like `eval()` (also highly advised against)

The following directives are available:

- `connect-src`: This restricts the domains that can be connected to via XHR and WebSockets
- `font-src`: This limits the domains that can be used to download font files
- `frame-src`: This limits the domains that frames (including inline frames) can load
- `img-src`: This limits the domains that images can be loaded from
- `media-sr`: This limits the origins for video and audio
- `object-src`: This allows control over the origins for objects (for example, Flash)
- `script-src`: This restricts the domains that scripts can be loaded from
- `style-src`: This limits the domains that stylesheets can be loaded from
- `default-src`: This acts as a shorthand for all the directives combined

Leaving out a directive leaves its policy wide open, (as is the default behavior) unless you specify the `default-src` directive.

Helmet can construct the headers for each supported User Agent (for example, browser) based on the configuration you pass to the

middleware. By default, it will give the following CSP header:

```
Content-Security-Policy: default-src 'self'
```

This is a very strict policy, as it will only allow external resources to be loaded from the current domain, and nowhere else. In most cases, this is simply too restrictive, particularly if you are going to be using a CDN or allowing external services to communicate with your own.

You can configure helmet via the middleware definition function, by adding a property called `defaultPolicy` that contains your directives as an object hash, for example:

```
app.use(helmet.csp.policy({
    defaultPolicy: {
        "script-src": [ "'self'" ],
        "img-src": [ "'self'", "http://example.com/" ]
    }
}));
```

This will instruct helmet to send the following header:

```
Content-Security-Policy: script-src 'self'; img-src
'self' http://example.com/
```

This will restrict scripts and images to the current domain as well as the domain http://example.com/.

CSP also includes a reporting capability that you can use for auditing your own applications and detect vulnerabilities quickly. There is a `report-uri` directive just for this purpose, which tells the browser what URI to send a violation report to. Refer the following example code:

```
Content-Security-Policy: default-src 'self'; ...;
report-uri /my_csp_report_parser;
```

When the report is sent by the browser, it is a JSON document with the following structure:

```
{
```

```
  "csp-report": {
    "document-uri": "http://example.org/page.html",
    "referrer": "http://evil.example.com/",
    "blocked-uri": "http://evil.example.com/evil.js",
    "violated-directive": "script-src 'self'
https://apis.google.com",
    "original-policy": "script-src 'self'
https://apis.google.com; report-uri
http://example.org/my_amazing_csp_report_parser"
  }
}
```

This report includes most of the information you should need to track down the violation, namely:

- `document-uri`: The page that the violation occurred on
- `blocked-uri`: The violating resource
- `violated-directive`: The specific directive that was violated
- `original-policy`: The page's policy (the contents of the CSP header)

When first starting out with CSP, it may not be wise to set up a policy and start blocking right away. While you are in the process of detailing your application's policy, you can set up CSP to respect report-only mode.

This allows you to set up a complete policy, and rather than blocking users right away, you can simply receive reports detailing violations. This gives you a way to fine-tune your policy before putting it into effect.

To enable report-only mode, you simply change the HTTP header name. Instead of what we've been using, you simply use `Content-Security-Policy-Report-Only`, leaving everything else the same:

```
Content-Security-Policy-Report-Only: default-src
'self'; ...; report-uri /my_csp_report_parser;
```

In helmet, you enable report-only mode by including the `reportOnly` parameter in your configuration object:

```
express.use(helmet.csp.policy({
    reportOnly: true,
    defaultPolicy: {
        "script-src": [ "'self'" ],
```

```
            "img-src": [ "'self'", "http://example.com/" ]
        }
    }));
```

This sets up the same policy we used earlier, just with the addition of report-only mode.

CSP is an excellent security mechanism that you should start using right away, despite browser support isn't entirely there. As of this writing, it is a **W3C Candidate Recommendation**, and browsers are expected to implement this feature at a rapid pace.

HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (**HSTS**) is a mechanism that communicates to a user agent (for example, a web browser) that a particular application should only be accessed via HTTPS, because it is an encrypted communication. If your application ideally exists only over a secure connection, this allows you to officially declare it to the browser.

There are only two parameters to this header, the `max-age` directive that tells the browser how long (in seconds) to respect the configuration, as well as the `includeSubDomains` directive that treats subdomains of the current domain in the same fashion. Like CSP, this is communicated via an HTTP header:

```
Strict-Transport-Security: max-age=15768000
```

This tells the browser, for around six months, that the current domain from now on should be accessed via HTTPS (even if the user accessed it via HTTP). This is the default configuration set by helmet, which is the simplest to implement:

```
app.use(helmet.hsts());
```

This sets up the middleware for HSTS using the previously stated configuration, the middleware definition function also takes two optional parameters. First, the `max-age` directive can be set as a number (which

should be represented in seconds). Second, the `includeSubDomains` directive can be set as a simple Boolean value:

```
app.use(helmet.hsts(1234567, true));
```

This will set the following header:

```
Strict-Transport-Security: max-age=1234567;
includeSubdomains
```

Browser support is not currently as complete as CSP, but is expected to proceed down that path. In the meantime, it is worth adding to your application's security detail.

X-Frame-Options

This header controls whether or not a particular page is allowed to be loaded into either a `<frame>` or an `<iframe>` element. This is useful mainly to prevent malicious users from hijacking (or "clickjacking") your users, and thereby tricking them into performing actions they otherwise had no intention of doing.

This is communicated to the browser via another HTTP header, so when the browser loads a URL for a frame/iframe, it will check for this header to determine the course of action to take. The header looks like the following:

```
X-Frame-Options: DENY
```

Here, we are using the value `DENY`, which is the default when configured via helmet. Other available options include `sameorigin`, which only allows the domain to be loaded in a frame when on the current domain. The last option is the `allow-from` option that allows you to specify a whitelist of URIs that can render the current page in a frame.

In most cases, the default should work just fine, and you can set that up via helmet like so:

```
app.use(helmet.xframe());
```

This adds the header as we saw it previously displayed. To configure using the `sameorigin` option, use the following configuration:

```
helmet.xframe('sameorigin');
```

Lastly, this sets up the `allow-from` variant, which also gives you the second parameter for setting allowed URIs:

```
helmet.xframe('allow-from', 'http://example.com');
```

Browser support for this security mechanism is quite good, so it's safe to implement right away. The `allow-from` header is a caveat, which is not supported evenly, so make sure you research the specifics depending on your requirements before using it.

X-XSS-Protection

This next header is specific to Internet Explorer, and it enables the XSS filter. Rather than explain it myself, here is an explanation from the **Microsoft Developer Network** (**MSDN**).

> ## Note
>
> The XSS filter operates as an Internet Explorer 8 component with visibility into all requests/responses flowing through the browser. When the filter discovers likely XSS in a cross-site request, it identifies and neuters the attack if it is replayed in the server's response. For further information please visit: http://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx

This featured is likely enabled by default, but in case the user has disabled it themselves or in some select zones, it can be enabled with a simple header that looks like the following:

```
X-XSS-Protection: 1; mode=block
```

By setting the header as 0, forces the XSS filter to be disabled, but that configuration is not exposed via helmet. In fact, it has no configuration at all, so its usage is as simple as:

```
app.use(helmet.iexss());
```

X-Content-Type-Options

This is another header that puts a stop to a specific behavior in certain browsers (Internet Explorer and Google Chrome are currently the only browsers that support this). In this case, the browser will attempt to "sniff" (for example, guess) the MIME type of a returned resource, even if that resource sets a valid `Content-Type` header on its own.

This could allow the browser to be fooled into executing or rendering a file in a way that was unintended by the developers, causing potential security vulnerabilities depending upon a number of factors. The point is that your server's `Content-Type` header should be the only consideration the browser makes, rather than trying to guess on its own.

Like the previous example, there is no real configuration available, and the following header will simply be added to your application:

```
X-Content-Type-Options: nosniff
```

This header is configured with helmet via:

```
app.use(helmet.contentTypeOptions());
```

Cache-Control

The last middleware that helmet provides is one for setting the `Cache-Control` header to `no-store` or `no-cache`. This prevents browsers from caching a given response. This middleware also has no configuration, and is included via:

```
app.use(helmet.cacheControl());
```

You would use this middleware and header to prevent the browser from storing and caching pages that may contain sensitive user information. However, the trade-off is that you could take a serious performance hit when applying it across the board.

When it comes to static files and assets, such as style sheets and images, this header will only slow your site down, and likely adding no security benefits while doing so. Make sure to be careful how and where you apply this particular middleware within your overall application.

The helmet module is a quick way to add these useful security features to your application, which is enabled by the powerful middleware architecture that Connect has created. There's a lot to many of these security features that cannot be addressed here, and will likely change in the future, so it's best to become familiar with all of them.

Summary

In this chapter, we looked at vulnerabilities that show up in the response phase of application processing, such as XSS and DoS. We also looked at ways to mitigate those specific problems, whether by defensive coding or using newer security standards and policies to our advantage.

Index

A

- age field / Input validation
- Apache / Denial of Service
- authentication
    - about / Authentication
    - HTTP Basic Authentication / HTTP Basic Authentication
- authorization
    - about / Authorization

C

- cache-control header
    - setting / Cache-Control
- Connect / Introduction to Express
- Cross-Site Scripting / Input validation
- Cross-site Scripting (XSS)
    - about / Cross-site Scripting (XSS)
- CSRF
    - about / Cross-site Request Forgery

D

- Denial of Service (DoS)
    - about / Denial of Service

- Denial of Service (DOS) / Using streams instead of buffering
- deserializeUser function / Introducing Passport.js
- directives / Content security policy
- Domains / Domains

E

- e-mail field / Input validation
- encodeURI() function / Cross-site Scripting (XSS)
- error handling / Error handling
- ES5
    - features / ES5 features, Strict mode, Object property descriptors
- eval() function / JavaScript security, Cross-site Scripting (XSS)
- event loops responsiveness
    - monitoring / Monitoring the event loop's responsiveness
- Express
    - about / Introduction to Express

/ Limiting the request size

F

- Facebook Developers / OAuth
- Function constructor / JavaScript security

G

- gender field / Input validation

H

- helmet / Security-related HTTP headers

- HTTP Basic Authentication / Introduction to Express, HTTP Basic Authentication, HTTP Digest Authentication
- HTTP Digest Authentication / HTTP Basic Authentication
- HTTP Strict Transport Security (HSTS) / HTTP Strict Transport Security (HSTS)

I

- input validation
  - about / Input validation
- IPC / Monitoring the event loop's responsiveness

J

- JSHint / Static program analysis
- JSON.parse() function / Limiting the request size

K

- keywords / Content security policy

M

- Microsoft Developer Network (MSDN) / X-XSS-Protection

N

- name field / Input validation
- nginx web server / How Node.js differs?
- Node.js
  - history / History of Node.js
  - features / How Node.js differs?
  - applications, securing / Securing Node.js applications

    - considerations / Considerations for Node.js, EventEmitter error handling, Uncaught exceptions, Domains

- Node.js applications
    - securing / Securing Node.js applications

- Node Security Project / npm modules (third-party code)
- npm (Node Packaged Modules) / History of Node.js
- npm modules / npm modules (third-party code)

O

- OAuth
    - about / OAuth

- Object.create() function / Object property descriptors
- Object.defineProperty() function / Object property descriptors
- Object.freeze() function / Object property descriptors
- Object.preventExtensions() function / Object property descriptors
- Object.seal() function / Object property descriptors
- OpenID
    - about / OpenID

- Open Web Application Security Project (OWASP) / Security logging, Cross-site Scripting (XSS)

P

- Passport.js / Introducing Passport.js
    - about / Introducing Passport.js

- process.exit() function / Uncaught exceptions

R

- request size
  - limiting / Limiting the request size
- Ruby / Introduction to Express

S

- sameorigin option / X-Frame-Options
- security-related HTTP headers
  - about / Security-related HTTP headers
  - Content Security Policy (CSP) / Content security policy
  - Same-Origin Policy (SOP) / Content security policy
  - HTTP Strict Transport Security (HSTS) / HTTP Strict Transport Security (HSTS)
  - X-Frame-options / X-Frame-Options
  - X-XSS-protection / X-XSS-Protection
  - X-Content-Type-options / X-Content-Type-Options
  - cache-control header, setting / Cache-Control
  - cache-control / Cache-Control
- security logging
  - about / Security logging
- serializeUser function / Introducing Passport.js
- Sinatra / Introduction to Express
- SNS / Security logging
- static analysis / Static program analysis
- streams
  - about / Using streams instead of buffering
  - using, instead of buffering / Using streams instead of buffering

U

- -use_strict command line flag / Strict mode
- uncaughtException event / Domains

- uniform resource identifier (URI) / Introduction to Express

V

- vm module / JavaScript security

W

- website field / Input validation
- winston module / Security logging

X

- X-Content-Type-Options / X-Content-Type-Options
- X-Frame-Options / X-Frame-Options
- X-XSS-Protection / X-XSS-Protection