Extracted from:

Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

Karl Düüna

edited by Fahmida Y. Rashid

Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

Karl Düüna

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-085-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2016

By failing to prepare, you are preparing to fail.

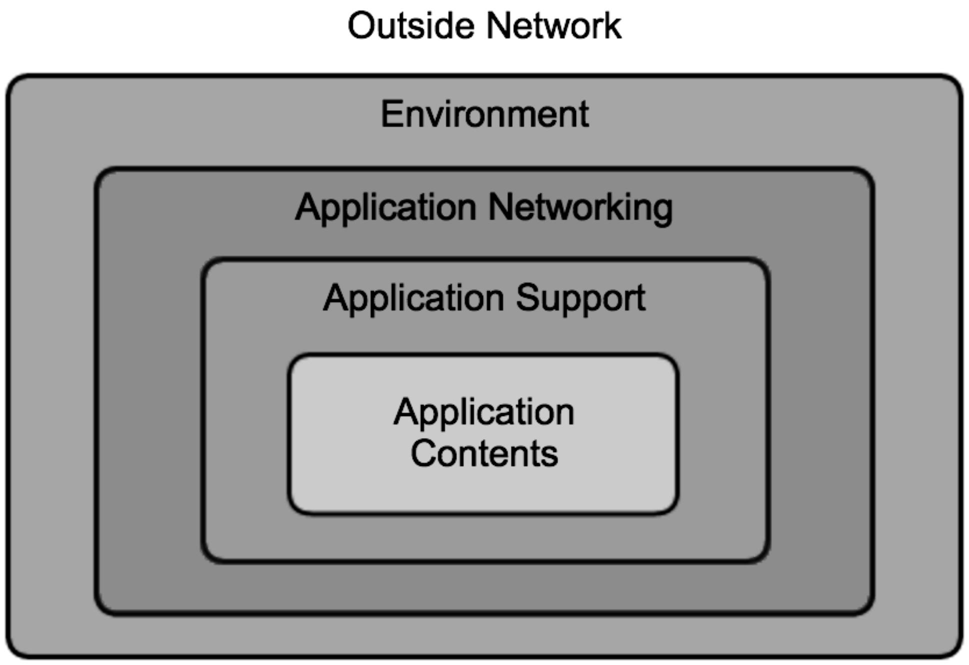
➤ *Benjamin Franklin*

CHAPTER 2

Set Up the Environment

You should now have a better understanding of how your tools work and, more importantly, how they can cause problems if not used correctly. In this chapter, we'll start working on the foundation—the server. There are many things to secure before we can write Node.js code.

You're looking at the title and wondering why I'm talking about the server instead of Node.js. Application security is a layered concept—we start from the outside and first secure our environment, network, and other auxiliary systems before we can even start work on the core application, as the following illustration shows.



Why? Because if we don't secure the surrounding layers, the inner defenses in our application matter little. We can't just put a password on a computer and say that computer is safe from thieves. We first need to lock the front door, right? That password won't stop a thief from simply taking the computer and walking out the door.

Every application has to live somewhere—a server, a phone, a device—an environment. Before we can secure higher levels of the stack and adopt secure coding practices, we must work our way up.

In this chapter, we'll discuss the *principle of least privilege*, how to properly configure our server, and ways to manage different environments. Yes, this isn't writing code, but good security starts with making sure the server is set up correctly. And while some of these topics might seem basic to you, I've seen time and time again that often it's the basics that get overlooked.

So, let's get started.

Follow the Principle of Least Privilege

The principle of least privilege (PLP) will help us design better security throughout the application.

In PLP, every abstraction layer in an application—program, user, process—has access only to the information and resources that it needs to complete its task. If the application layer can't access privileged resources, then it can't be abused to give attackers access to those resources. PLP limits damages in case of a breach.

A common example of PLP can be seen in the operating systems; as a user, you have a regular account for working with installed applications. When you want to do something that requires higher privileges, such as installing an application, you see a prompt asking for higher privileges. This kind of manual privilege escalation system makes it harder for attackers to execute malicious programs on victims' machines.

We can also see PLP in web applications, where the server process doesn't have read access outside the web application directory. This prevents attackers who somehow found a loophole in the server configuration from abusing it to read and modify other files on the server.

In reality, true PLP is practically impossible because it's extremely difficult to determine all the resources that a program needs and at what point in time. However, even a moderate implementation of the concept increases application security by a great deal.

From our web application standpoint, we have the following rules:

- The web application should not be run with `root` privileges. It should instead use a limited account that has access to only the required resources.
- The database account should not be a root account. The account should have limited privileges over the database tables. We touch upon this in Chapter 5, *Secure Your Database Interactions*, on page ?.
- The users of the web application should be given the minimum set of privileges they need.

Following this simple list while developing the application greatly increases the security and fault tolerance because the impact of all errors and vulnerabilities is contained within their specific areas.

Start with the Basics: Secure the Server

PLP isn't enough if the hardware housing our application is riddled with holes. Attackers are looking for any way in and will target both the production and development servers hosting the application. If we forget to secure the server itself in the rush to code a secure application, all the things we're going to discuss in later chapters will no longer matter.

What good is session management in our application if the server has a weak password? Does it matter if we implement a rock-solid authentication scheme if the server is running old and vulnerable software? No. Remember, we need to lock the front door before we password-protect the computer.

Since this book is about Node.js security and not server security, we'll keep things brief and basic.

The first step is authentication, because it's the most important aspect of server security. Keep the following guidelines in mind for a secure authentication scheme:

- *Do not use the root account all the time.* Using an ordinary account and `sudo` to elevate permissions when required minimizes the attack vector by limiting the timeframe and execution rights.
- *Do not give the same account to everyone.* It makes it hard to separate permissions of individuals or determine a point of attack later on.
- *Use dedicated machines.* Having the production site running on a machine otherwise used for email or web browsing opens up so many attack vectors that anything that you might save on hardware would be gobbled up by balancing security.

- *Keep access to the production server to a minimum.* There's no reason for someone from accounting to have root access to the production server hosting your web application. Let access be limited to the minimum number of people possible.
- *Change the default password or use key-based authentication.* Most cloud services provision machines with default root accounts and send the passwords by email. Change those!

The application server should be single purposed. Running a pet development app on the same server as a business-critical application is a great idea—if you want to sabotage your production environment. (That was sarcasm.) Don't do it, because you're just offering up a buffet of attack vectors to break into the server and the resident applications.

Set up a proper firewall. Block all network traffic that should not be occurring in the first place. If necessary, you can also set up a reactive firewall to block denial-of-service attacks when they occur.

Another basic step, but an important one, is to make sure all the software installed on the server is up to date. If the history of computers has shown us anything, it's that complex software without bugs is like a miracle—some say they have seen it, some even say that they have made such a program, and the rest of us just shake our head in disbelief. Keep your system up to date to limit exposure time to vulnerabilities as they're found.

In 2014 alone, two serious bugs were found in commonly used networking software and required a software update for almost all servers around the world: Heartbleed[1] and ShellShock.[2] Make sure you're running updated server software to ensure these two and other security bugs don't affect you.

Securing the server operating system, setting up firewalls, and hardening the environment are all broad topics and out of scope for this book. I recommend taking the time to understand network and OS-level security. Here are a few good online tutorials and lists to get you started:

- *Securing and Optimizing Linux: The Ultimate Solution*[3]
- An Introduction to Securing Your Linux VPS[4]
- 20 Linux Server Hardening Security Tips[5]

1. http://heartbleed.com/
2. http://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability
3. http://www.tldp.org/LDP/solrhe/Securing-Optimizing-Linux-The-Ultimate-Solution-v2.0.pdf
4. https://www.digitalocean.com/community/tutorials/an-introduction-to-securing-your-linux-vps
5. http://www.cyberciti.biz/tips/linux-security.html

Avoid Security Configuration Errors

Now that our server won't fall to the first script kiddie that comes along, let's make sure we won't make errors configuring our software stack. Breaches due to misconfiguration are more common than those due to *zero-day vulnerabilities*.

What Is a Zero-Day Vulnerability?

Some software bugs let attackers remotely execute commands on targeted machines. Although software developers quickly release patches as soon as these issues are found, sometimes attackers find the bugs first. Zero-day vulnerabilities are bugs attackers are using before the software developer has the chance to fix them. Zero days have passed since the patch for that bug, hence the name.

There are as many possible areas of misconfiguration as there are different combinations of software installed on your server, making it impossible to cover them all in a single book. Let's focus on common configuration mistakes and how to configure the production and development environments. These examples should give you a good grounding of what you should and shouldn't do in your setup.

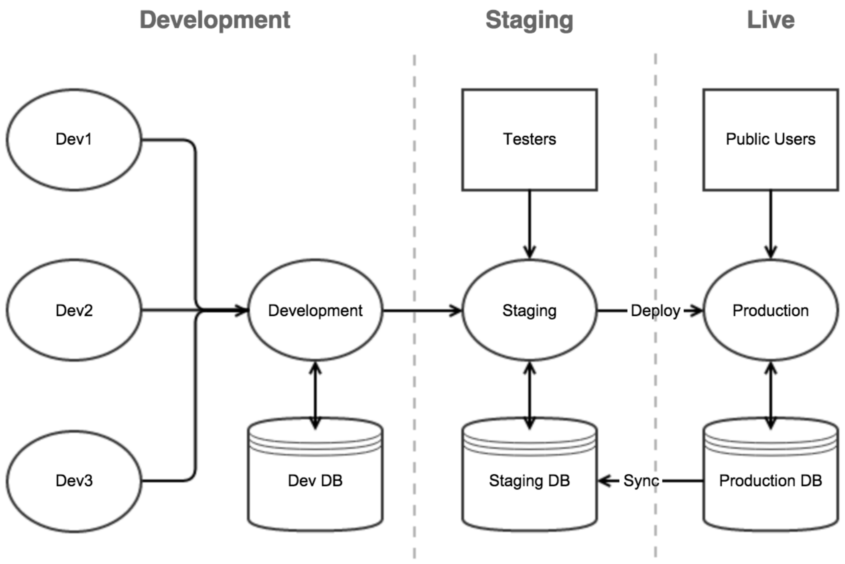
Change Default Users and Passwords

First, you need to keep an eye on default accounts. We all like it when things work right out of the box—little to no setup and everything runs smoothly. While the fact that some frameworks and content management systems (CMSs) ship with default accounts pre-created makes installation convenient, it presents a serious security threat.

What's the threat, you ask? Well, anyone who has either installed the software or read the documentation will know about those accounts. So unless you change them, anyone can use those accounts and walk through all your layers of security. This is a widespread issue, since default accounts can also be found on networking equipment, databases, and cloud server instances, to name a few. Any default accounts in the software and hardware stack *must* be either deactivated or reconfigured.

Set Up Separate Development and Production Servers

In a proper software development environment, you test and stage code so that you iron out any bugs before production. However, there's a security paradox: you want to keep the development, testing, and staging environments separate and homogenous at the same time. What you end up should be similar to the following diagram.

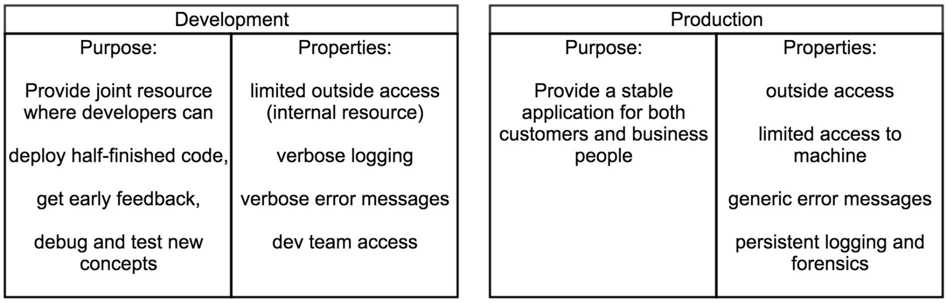


Let's start with the separation. Production and development environments should not be on the same machine. You might wonder why, especially since consolidating would reduce development costs.

The answer is simple: development versions of the application are by definition incomplete and have bugs. Attackers can exploit those issues to access production data or look at the source code to understand how the application works. The development environment should be treated as an internal resource, one that cannot be accessed directly from the web and lives behind an authentication screen to make sure only authorized users can get in.

To reiterate, keep your production code separate from everything else.

You want to make sure the application runs as expected in production, which means the development, testing, and production environments have to have the same software and settings. But it's neither optimal nor safe to configure production exactly the same as other environments, because they fulfill different purposes, as the following illustration shows.



Development	
Purpose:	Properties:
Provide joint resource where developers can	limited outside access (internal resource)
deploy half-finished code,	verbose logging
get early feedback,	verbose error messages
debug and test new concepts	dev team access

Production	
Purpose:	Properties:
Provide a stable application for both customers and business people	outside access
	limited access to machine
	generic error messages
	persistent logging and forensics

Development environment tends to have more relaxed security and verbose logging for debugging. All the developers on the team need to have access to the development environment. In comparison, the whole team typically doesn't work on production servers, so fewer users should be able to log in. As I also discuss in *Decide What Gets Logged*, on page ?, verbose logging in production is not a good idea.

There are a few ways to configure the development and production environments. You can do it manually with the `process.NODE_ENV` environment variable, use a configuration manager, or look for a built-in solution, such as the environment in the `express` framework. The manual process isn't recommended because it gets hard to maintain.

I prefer a configuration manager, but it depends on the complexity and size of the application. I like easy-config[6] (which I wrote), but there are dozens available, such as node-config[7] and nconf.[8]

I suggest using environmental variables such as `NODE_ENV` to differentiate between them externally. This is less error prone than using runtime arguments, and you're less likely to start up an environment with the wrong settings. Even the `express` framework recognizes `NODE_ENV`.

To sum up, live, or production, environments should have restricted policies, with fewer people having access and with fewer privileges. All third-party software should have separate accounts used only within the production environment. And finally, live environments should also have less-verbose logging and error handling, which we look at next.

6. https://github.com/DeadAlready/node-easy-config
7. https://github.com/lorenwest/node-config
8. https://github.com/flatiron/nconf

Limit Error Messages in Production

In the development environment it's useful to have descriptive error messages and the stack trace printed out for easy debugging. However, they shouldn't be shown in the production environment because they would provide attackers with extra information about the application structure and could possibly expose some vulnerabilities or attack vectors.

For example, with *SQL injection*, which we'll cover in Chapter 5, *Secure Your Database Interactions*, on page ?, there's a vast difference between regular SQL injection and blind SQL injection. The first shows descriptive error messages that provide attackers with insight into what exactly happened and what they have to do to construct a valid attack:

```
Error: ER_PARSE_ERROR: You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version
for the right syntax to use near '"karl""' at line 1
at Query.Sequence._packetToError (mysql/lib/protocol/sequences/Sequence.js:48:14)
at Query.ErrorPacket (mysql/lib/protocol/sequences/Query.js:82:18)
at Protocol._parsePacket (mysql/lib/protocol/Protocol.js:271:23)
at Parser.write (mysql/lib/protocol/Parser.js:77:12)
at Protocol.write (mysql/lib/protocol/Protocol.js:39:16)
...
```

Blind SQL is much more difficult for attackers since error messages provide no information about what went wrong. That's what we want to see more of. Or in this case less of.

In the express framework, the default error handler watches for NODE_ENV to determine if the detailed stack trace information gets shown. If you set NODE_ENV=production, then all you see is the message, Internal Server Error.

As it should be. As long as you're tight lipped in your production environment, then you can feel good—you're doing things the way they should be done.

Locking the Environment

We've been talking for a while about how the production and development environments should be on separate machines and have different configuration settings. But at the same time we also need the environments to be homogeneous.

Before you start roll your eyes and say, "What!" let me explain. I am talking about the software stack. In a perfect world, the servers are clones in terms of operating system, packages installed, and configuration settings.

The recommended setup for Node.js projects places used modules in a package.json file, to look something like the following:

```
{
    "name": "security-misconfiguration",
    "version": "0.0.1",
    "main": "environment.js",
    "dependencies": {
        "connect-redis": "*",
        "easy-session": "*"
    }
}
```

(This example package.json is useless; it was severely gutted to shorten the examples that follow.)

package.json lets you use the npm install command to install all the dependencies. Since we marked each dependency with *, the latest available versions from the repository will be installed. This sounds like a good idea, except now we don't know whether any of the packages have been updated since they were installed in the development environment. The production environment may wind up with newer versions installed than the ones in development.

We talked about having updated software earlier, so having newer versions is better, right? While that's true to some extent, if you haven't tested your application against the newer version of software, then you don't know about potential problems. Maybe the new version of the software causes your application to break or introduces some weird inconsistencies. If you don't know that your software works exactly the same way in production as it does in development, that's not a good thing.

So we should change the package.json to tie down the versions:

```
{
    "name": "security-misconfiguration",
    "version": "0.0.1",
    "main": "environment.js",
    "dependencies": {
        "connect-redis": "~1.4.6",
        "easy-session": "0.0.2"
    }
}
```

This defines more precisely the versions of the packages you want to use. The first is added as an approximate version and the last as a specific version of the package. To make it easier to add approximate versions during development, use the --save flag:

```
npm install express --save
```

This will install the express module and add "express": "~3.4.8" under "dependencies".

This seems to solve the problem, except for the fact that required dependencies frequently have subdependencies. So it didn't really fix the issue, did it? The following example lists the dependencies for connect-redis:

```
"dependencies": {
  "redis": "0.9.x",
  "debug": "*"
},
"devDependencies": {
  "connect": "*"
}
```

This might make it look like the node_modules folder should be included within the repository itself. However, since Node.js supports modules written in C and C++ as well, some of them might need compiling and compiled modules tend to break when moved around. To mitigate, we can use shrinkwrap[9] to lock up the whole dependency tree.

For example, let's look at our original project to install two dependencies. Run the following to create the npm-shrinkwrap.json:

```
npm shrinkwrap
```

```
{
  "name": "security-misconfiguration",
  "version": "0.0.1",
  "dependencies": {
    "connect-redis": {
      "version": "1.4.6",
      "from": "connect-redis@*",
      "dependencies": {
        "redis": {
          "version": "0.9.2",
          "from": "redis@0.9.x"
        },
        "debug": {
          "version": "0.7.4",
          "from": "debug@*"
        }
      }
    },
    "easy-session": {
      "version": "0.0.2",
      "from": "easy-session@*"
    }
  }
}
```

9. https://www.npmjs.org/doc/cli/npm-shrinkwrap.html

Now we can see all dependencies listed in the tree. When we run `npm install` and the `npm-shrinkwrap.json` file is in the directory next to `package.json`, then `shrinkwrap` installs the same versions we used originally. This keeps the environments homogeneous. I won't go into more detail here, but I recommend looking up more information about the versioning and contents of `package.json`.[10]

Wrapping Up

Proper server configuration is critical for any secure web application because you're building the base of the application. In this chapter, we looked briefly at securing the web server, password security, and separating development and production environments. The last thing you want is poor setup or a bug in development to compromise production.

We continue our focus on layered security by moving up another step. This time we'll secure the network layer, which defines how we communicate with the world. Let's move on.

10. https://www.npmjs.org/doc/files/package.json.html#dependencies