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

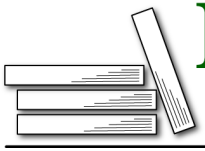
Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

Karl Düüna

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

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

If you know the enemy and know yourself, you need not fear the result of a hundred battles.

➤ *Sun Tzu*

CHAPTER 7

Bring Authentication to Your Application

You've securely set up your server and database, and you now have an application with valuable information people want to see. But how do you know a user is who he or she claims to be, and how do you avoid malicious impersonators? You don't want to hand out personal information to just anyone, so you need to think about authentication.

The level of security you need when dealing with user accounts and how to validate them depends on the application and how much personal information you're storing. Consider how much damage an attacker can do to the customer if the account is breached. If the application stores credit card information, then it must have extra levels of validation to protect users and their data. This chapter focuses on the common username/password authentication system because you're already familiar with it and because it's easy to understand.

Don't Forget About PCI DSS if You Store Credit Cards

When dealing with credit card information, you have to follow the Payment Card Industry Data Security Standard (PCI DSS).[1]

A user sets up an account by providing a secret (a password), and later you verify that the user knows the secret before allowing access. Controlling this knowledge lets you assume that you're dealing with a trusted party. But because everyone uses this form of authentication, there are many attack vectors that specifically attempt to break it.

We'll look at various parts of this system and how to harden your setup so it will be more robust and not easily fooled.

1. https://www.pcisecuritystandards.org/security_standards/

Store the Secret in a Safe Place

Let's start with storage—you have to store the password somewhere so that you can validate that the user knows the secret. There's a big difference between saving the password and saving the password *securely*.

First off, and I do hope I stress this enough, *never, ever store passwords in plain text*. That's just asking for trouble. You may think that people who will see the passwords will already have access to the data—so what's the big deal. Oh, how wrong you would be.

There are two important differences between storing passwords in plain text and hashed: impersonation and collateral damage. First off, seeing the password in hashed format will not allow you to simply log in as the user because you still don't know the secret. Depriving a malicious party of this is already a big win; however, the bigger issue is your users themselves. People tend to reuse passwords on different sites—so knowing a user's password on one site potentially gives access to several other accounts on other sites as well.

If you store passwords in plain text on the disk, then you're both putting an awful lot of trust in your administrations and gambling with your users' data, since even a simple breach in your security will allow the attacker to compromise not only your site but likely other accounts on other sites as well. So do everyone a favor and say no to plain text.

Plain text is the worst choice you can make. But if you were thinking about using general-purpose hash functions such as MD5, SHA1, SHA256, SHA512, and SHA-3, then you would also be wrong. They're designed to calculate the hash as fast as possible, a feature you don't want in a password-hashing function. You want the attackers to have to spend a long time trying to crack the hashes in case of a database breach.

Why Passwords Get Cracked

Let me digress to discuss cracking for a moment. If the application stores its passwords in hash format, attackers who get their hands on the hashes can't use the hash directly to log into the application. They need to find a plain-text version of the password. This gets harder if the password is stored using one-way hashing functions. Then the only way to determine the original password is to generate hashes for all possiblities until they find a matching one. How fast the function can calculate the hash has a significant impact on password safety.

A modern server can calculate an MD5 hash of about 330 MB every second. This means that if you have lowercase, alphanumeric passwords that are six characters long, then every possible combination can be calculated in 40 seconds. That's without investing any real money. If attackers are willing to pay, they can drastically increase computational power and reduce the time required to crack hashes. Also, sometimes flaws are discovered in the hashing algorithm itself that help narrow the possible inputs, greatly decreasing the time it takes to find a solution.

Attackers take the time and effort to crack the passwords because they can go after other user accounts on the current site as well as any place the passwords may have been reused.

"Hey, but I salt my passwords!" you may be saying right now. Well, salts won't help if the attackers are intent on cracking your passwords. Salts were designed to defend against dictionary attacks, but computational power is so cheap nowadays that attackers just go straight to brute-force cracking.

Salting

Password salting means adding a secret string to all passwords before hashing them in order to avoid getting the same hash for common passwords. This helps mitigate dictionary attacks, where attackers search for a match of the password hash from a large precomputed list—much faster than brute forcing.

Use Hash Functions Designed for Passwords

With the quick lesson in cracking, you should now know why general hash functions aren't recommended for passwords. Instead, use *bcrypt*[2] or *scrypt*,[3] which are specifically designed for passwords.

bcrypt is based on the Blowfish[4] cipher and is slow in hash calculation depending on the number of iterations. scrypt was developed to make it costly to perform custom hardware attacks by raising the memory requirements of the hash calculation, thus increasing the cost of hardware implementations.

Let's see how these compare to each other on my laptop. In my code I used the most popular bcrypt[5] and scrypt[6] libraries for Node.js. Using twelve iterations

2. http://bcrypt.sourceforge.net
3. https://www.scrypt.com
4. https://www.schneier/com/blowfish.com
5. https://github.com/ncb000gt/node.bcrypt.js
6. https://github.com/barrysteyn/node-scrypt

on bcrypt and the default settings for scrypt (with max calculation time of 1s) I got the following results:

```
      md5: 5d41402abc4b2a76b9719d911017c592
md5: 7μs
sha1: aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
sha1: 7μs
sha256: 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
sha256: 8μs
sha512: 9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890caadae2dff72519673ca723
23c3d99ba5c11d7c7acc6e14b8c5da0c4663475c2e5c3adef46f73bcdec043
sha512: 8μs
bcrypt: $2a$12$N.xKbRQZ10Bzd9QAFBFfBu2abQMUWZHsKcoctu30nU2iw2YI0DNwG
bcrypt: 283ms
scrypt: 736372797074001000000000800000005a539df65707e021f8afde283021dac7423
b8ebc3ecd5653b1dc0eb0a7e96c1212d95502588785cde34e05913cc874f9f496a2e388b83
994a3321413c15278915923dcf94b771d69cf64b53bc96282a28
scrypt: 640ms
```

As you can see, bcrypt takes about 0.3s to calculate and scrypt about 0.6s, whereas the rest are in microseconds. When cracking passwords this difference will translate every second into a timescale of days. That makes a huge difference when cracking passwords, since something that would otherwise take a day would now take more than 100 years.

bcrypt and scrypt also incorporate a salt to resist rainbow table attacks and are adaptive functions. This means that you can increase the calculation costs by changing the settings—making it resistive to brute force even when computational power increases dramatically.

I personally use bcrypt because it's easier to use than scrypt and my security requirements are usually not that high.

So how do you use it? Let's look at an example of a basic Mongoose model that uses bcrypt and hooks to store the password securely:

chp-7-authentication/mongoose-bcrypt.js

```
'use strict';

var mongoose = require('mongoose');
var bcrypt = require('bcrypt');
var Schema = mongoose.Schema;

var accountSchema = new Schema({
    email:  { type: String, required: true, index: { unique: true } },
    password: { type: String, required: true }
});

// Define pre-save hook
```

```
accountSchema.pre('save', function (next) {
    var user = this;

    // only hash the password if it has been modified (or is new)
    if (!user.isModified('password')) {
        return next();
    }

    bcrypt.hash(user.password, 12, function (err, hash) {
        if(err) {
            next(err);
            return;
        }
        user.password = hash;
        next();
    });
});

// Define a method to verify password validity
accountSchema.methods.isValidPassword = function (password, callback) {

    bcrypt.compare(password, this.password, function (err, isValid) {
        if(err) {
            callback(err);
            return;
        }
        callback(null, isValid);
    });
};

module.exports = accountSchema;
```

With these tools, you can now store your user passwords in a manner that will keep attackers cracking at them for years.

Enforce Password Strength Rules on Your Users

Now that we've covered storage, let's talk about the password itself. Most users aren't security conscious, so you have to help the user when selecting a password. The table on page 10 is a top-ten list of the most popular passwords from 2014[7] It's obvious that people don't really think about account security.

Don't let your users use common dictionary passwords, because your high-tech security measures are useless if the user is using monkey (position 12) or letmein (position 13) as a password. When the user selects a password, compare

7. https://www.teamsid.com/worst-passwords-of-2014/.

Position	Password	2014 Rank	Position	Password	2014 Rank
1.	123456	Unchanged	6.	123456789	Unchanged
2.	password	Unchanged	7.	1234	Up 9
3.	12345	Up 17	8.	baseball	New
4.	12345678	Down 1	9	dragon	New
5.	qwerty	Down 1	10.	football	New

Table 1—Top-ten Passwords of 2014

the string against a known dictionary of common passwords to make sure it isn't weak. You can easily find lists of common passwords[8] with a simple online search. The following example uses one such list to validate if the selected password exists in a dictionary:

chp-7-authentication/dictionary-validator.js

```
'use strict';

var fs = require('fs');

var dictionary = {};

// Since we are doing it only once on startup then use sync function
fs.readFileSync(__dirname + '/data/dictionary.txt', 'utf8')
    .split('\n')
    .forEach(function (password) {
        dictionary[password] = true;
    });

// This function will return an error message if the password is not good
// or false if it is proper
module.exports.isImproper = function check(username, password) {

    // About 3 percent of users derive the password from the username
    // This is not very secure and should be disallowed
    if(password.indexOf(username) !== -1) {
        return 'Password must not contain the username';
    }

    // Compare against dictionary
    if(dictionary[password]) {
        return 'Do not use a common password like: ' + password;
    }
    return false;
};
```

8. https://wiki.skullsecurity.org/index.php?title=Passwords

The more complete the dictionary, the better protection against weak passwords it will provide, but even the smallest dictionaries with just 500 common passwords would provide some protection.

To further increase password security, you should force the user to select stronger passwords. Instead of forcing the user to create passwords with special characters that are hard to remember, have them select longer passwords. Long passwords are easier to remember and offer better security because the resulting hashes take a longer time to crack.

Depending on the nature of the application, I also suggest the user should be forced to change passwords periodically, whether that's once a month, once a quarter, or even twice a year. This limits the timeframe in which attackers can try to break in with stolen passwords. And they have to start over and recrack the new password after every change. If you do require users to change their passwords, don't let them use previously used passwords. Just keep the previous hashes and compare the hash of the new one to make sure the user isn't trying to reuse the password.

Force users to use longer passwords, disallow common passwords, and change them periodically. These three tips will help keep data stored by your application safe.

Move the Password Securely to the Server

We've established that the user needs to set a strong password and have covered how to store it. How do you move it from the web browser to the server? The first step, of course, is to use HTTPS. In fact, you should use HTTPS not just on login and registration pages but for the whole site. You will need HTTPS for login and registration pages to prevent man-in-the-middle attacks that try to steal passwords, but if you don't use HTTPS for the whole site, your session can still be stolen. This is discussed in length in Chapter 8, *Focus on Session Management*, on page ?.

Second, *do not send a plain-text password to the user's email* as a reminder. If the application is generating the password on the user's behalf, then force the user to change it immediately the first time the user logs in. Having a permanent plain-text record of a user's password in an email inbox is like the employee who writes passwords on Post-It notes and puts them next to the screen.

Third, *insert delays* in your login mechanism. We already covered brute-forcing at the storage level, but you can also slow down brute-forcing attempts on the application layer. A common way to do this is to punish the user for

repeatedly failing to log into the application. You can ban the user for a while, such as fifteen minutes after five failed attempts, or make the user fill out a CAPTCHA challenge. Banning the user is a double-edged sword, because an attacker can maliciously block legitimate users by intentionally entering bad passwords, so use it carefully. The other approach is to create a universal delay for each failed login for a certain period of time. The legitimate user won't feel the delay; the attacker will.

Let's look at how you can ban the user's IP for a period of time if the user fails to log in a certain number of times:

chp-7-authentication/ban-user.js

```
var maxFailedCount = 5; // Max tries
var forgetFailedMins = 15; // time the user will be blocked
var blockList = {};

// Check if ip is still allowed
function isAllowed(ip) {
    return !blockList[ip] || blockList[ip].count < maxFailedCount;
}
// Remove ip from blockList
function successfulAttempt(ip) {
    if(blockList[ip]) {
        if(blockList[ip].timeout) {
            clearTimeout(blockList[ip].timeout);
        }
        delete blockList[ip];
    }
}
// Increment blocklist counter
function failedAttempt(ip) {
    if(!blockList[ip]) {
        blockList[ip] = {
            count: 0
        };
    }
    blockList[ip].count++;
    if(blockList[ip].timeout) {
        clearTimeout(blockList[ip].timeout);
    }
    blockList[ip].timeout = setTimeout(function () {
        delete blockList[ip];
    }, forgetFailedMins * 60 * 1000);
}

app.post('/login', function (req, res, next) {
    if(!isAllowed(req.ip)) { // Check if user is blocked
        req.session.error = 'You have been blocked for ' +
            forgetFailedMins + ' minutes';
```

```
        res.redirect('/');
        return;
    }
    validateUser(req.body, function(err, valid) {
        if(err) {
            next(err);
            return;
        }
        if(valid.success) { // Validation success. Create authorized session.
            successfulAttempt(req.ip); // Clear from blocklist
            req.session.login({userId: valid.userId}, function () {
                res.redirect('/user/' + valid.userId);
            });
        } else {
            failedAttempt(req.ip); // Register the failed attempt
            req.session.error = valid.error;
            res.redirect('/');
        }
    });
});
```

Node.js also lets us easily set a universal delay in answering:

chp-7-authentication/delay.js

```
app.post('/login', function (req, res, next) {
    function end(url) {
        setTimeout(function () {
            res.redirect(url);
        }, 1000);
    }
    validateUser(req.body, function(err, valid) {
        if(err) {
            next(err);
            return;
        }
        if(valid.success) { // Validation success. Create authorized session.
            req.session.login({userId: valid.userId}, function () {
                // delay before answer
                end('/user/' + valid.userId);
            });
        } else {
            req.session.error = valid.error;
            // delay before answering
            end('/');
        }
    });
});
```

However, the delay mechanism won't stop attackers from running parallel checks about a user's account. We can fix that problem, though:

chp-7-authentication/delay-no-parallel.js

```
// Map our authentications
var inProgress = {};

app.post('/login', function (req, res, next) {
    var key = req.ip + ':' + req.body.username;
    // check if we are already authenticating this user from the given IP
    if(inProgress[key]) {
        req.session.error = 'Authentication already in progress';
        res.redirect('/');
        return;
    }
    inProgress[key] = true;
    function end(url) {
        setTimeout(function () {
            delete inProgress[key];
            res.redirect(url);
        }, 1000);
    }
    validateUser(req.body, function(err, valid) {
        if(err) {
            delete inProgress[key];
            next(err);
            return;
        }
        if(valid.success) { // Validation success. Create authorized session.
            req.session.login({userId: valid.userId}, function () {
                // delay before answer
                end('/user/' + valid.userId);
            });
        } else {
            req.session.error = valid.error;
            // delay before answering
            end('/');
        }
    });
});
```

Examples Are Not Production-Ready Code

The previous examples have been simplified and are not directly usable in a production environment. For example, using session to transfer the error message can cause issues. Holding the list of in-progress validations in memory won't be valid if the process is forked.

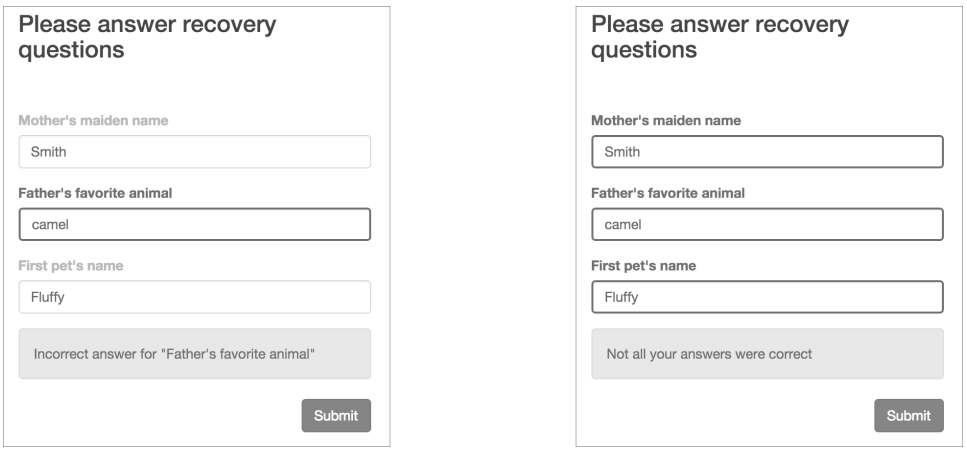
This will stop brute-force and dictionary attacks directly against the application since the delays would slow down the attacker too much to make it worthwhile, unless of course the user's password is in the top ten.

Deal with the Fact That Users Will Forget

Humans are not computers and will forget things, even important things like credentials to an awesome web application such as yours. So let's talk about setting up a secure password-recovery mechanism.

The most common recovery system in modern web applications uses email. A link is sent to the registered email address to prompt the user to change the current (forgotten) password to a new one. While sufficient for most applications, if your application is extremely critical, you need a more secure recovery process. One option is to add a set of recovery questions or a secondary password that the user has to provide as part of the recovery process. This will stop attackers who have access of the victim's email inbox because they won't know the answers to those questions or the secondary password.

You should consider recovery answers as passwords that are intended to be easier to remember. As such they should also be hashed, and the application should validate the recovery questions as a group. If there are three questions, the person has to get the answers to all of them correct in order to proceed. If one of them is wrong, the person will be shown an error message, but this is important: do not specify which question was incorrect. The illustration shows the bad way to handle recovery questions (on the left) and the good way (on the right).



This is also how you would want to handle the actual login form. Do not say specifically if the username or the password was incorrect—just say that the combination was wrong. And as with login forms you also want to protect against brute-force attempts to figure out answers to recovery questions, so limit the number of attempts and insert delays.

With email-based recovery systems, always validate the user's email address, and don't let the user change it without revalidating it and using another layer of authentication. Without secondary verification when changing email addresses, an attacker who manages to break into the account can change the address unchallenged and permanently lock out the legitimate user.

Add Other Authentication Layers for Better Security

For important applications, add another layer of authentication besides username and passwords. This will increase security because the attacker now has more layers to cut through. Some ways to achieve this include using hidden usernames, second passwords, and multi-factor authentication.

A hidden username is a two-username system. One is the username other users see and the other is used only for logging in. This is common in forums, where everyone sees a username, but you log in with your email address.

You can also let the user set up two passwords. The first password is used for logging in, and the second one is reserved for special requests and operations. One example is to use the second password to change the email address associated with the account. The session lifetime of the second password should be short. This will stop attackers who have gained access to the session or the first password from doing much damage.

Multi-factor authentication is becoming increasingly popular. The most common form, two-factor authentication, uses a third-party system like Google Authenticator to generate special codes. The idea is to force the user to log in with something only the user knows (a password) and something the user has (a phone running the Google Authenticator app, for example). This requires the attacker to also steal or compromise the phone in order to successfully log in. Multi-factor authentication schemes make credential theft much more difficult because the attacker has to bypass a second system to gain access.

Wrapping Up

In this chapter we looked into hardening one of the backbones of web application security—authenticating the user. We looked at ways to store passwords securely, how to force users to use stronger passwords, how to protect against brute-force attacks, and how to add a second layer of protection.

Having covered the bases for authenticating a user, we now look at how the application remembers the user for a set period of time. We'll cover sessions in the next chapter so that your users won't have to keep typing in their password everytime they want to do something.