7
Implementing Security, Encryption, and Authentication

In this chapter we will cover:

- Implementing Basic Authentication
- Cryptographic password hashing
- Implementing Digest Authentication
- Setting up an HTTPS web server
- Preventing cross-site request forgery

Introduction

When it comes to production web servers, security is paramount. The importance of security correlates with the importance of the data or services we provide. But even for the smallest projects, we want to ensure our systems aren't vulnerable to attack.

Many web development frameworks provide built-in security, which is a two-sided coin. On one side, we don't have to overly concern ourselves with the details (except for the basics, like cleaning user input before passing it into an SQL statement), but on the other we implicitly trust that the vendor has plugged all the holes.

If a largely used server-side scripting platform, such as PHP, is discovered to contain a security vulnerability, this can become public knowledge very quickly and every site running the vulnerable version of that framework is open to attack.

With Node, server-side security is almost entirely on our shoulders. Therefore, all we need to do is educate ourselves on the potential vulnerabilities and harden our systems and code.

For the most part, Node is minimalistic: if we don't specifically outline something it doesn't happen. This leaves little room for exploitation of unknown parts of our system or obscure configuration settings because we coded and configured our system by hand.

Attacks take place from two angles: exploiting technical flaws and taking advantage of user naiveté. We can protect our systems by educating ourselves and conscientiously checking and rechecking our code. We can also protect our users by educating them.

In this chapter, we will learn how to implement various types of user-authenticated logins, how to secure these logins, and encrypt any transferred data, along with a technique for preventing authenticated users from falling victim to exploits of the browser's security model.

Implementing Basic Authentication

The Basic Authentication standard has been in place since the 1990s and can be the simplest way to provide a user login. When used over HTTP, it is in no way secure since a plain text password is sent over the connection from browser to server.

For information on Basic Authentication see `http://en.wikipedia.org/wiki/Basic_authentication`.

However, when coupled with SSL (HTTPS), Basic Authentication can be a useful method if we're not concerned about a custom-styled login form.

We discuss SSL/TLS (HTTPS) in the *Setting up an HTTPS web server* recipe of this chapter. For extra information see `http://en.wikipedia.org/wiki/SSL/TLS`.

In this recipe, we'll learn how to initiate and process a Basic Access Authentication request over plain HTTP. In following recipes, we'll implement an HTTPS server, and see an advancement of Basic Authentication (Digest Authentication).

Getting ready

We just need to create a new `server.js` file in a new a folder.

How to do it...

Basic Authentication specifies a username, password, and realm, and it works over HTTP. So we'll require the HTTP module, and set up some variables:

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

var username = 'dave',
  password = 'ILikeBrie_33',
  realm = "Node Cookbook";
```

Now we'll set up our HTTP server:

```
http.createServer(function (req, res) {
  var auth, login;

  if (!req.headers.authorization) {
    authenticate(res);
    return;
  }

  //extract base64 encoded username:password string from client
  auth = req.headers.authorization.replace(/^Basic /, '');
  //decode base64 to utf8
  auth = (new Buffer(auth, 'base64').toString('utf8'));

  login = auth.split(':'); //[0] is username [1] is password

  if (login[0] === username && login[1] === password) {
    res.end('Someone likes soft cheese!');
    return;
  }

  authenticate(res);

}).listen(8080);
```

Notice we make two calls to a function named `authenticate`. We need to create this function, placing it above our `createServer` call:

```
function authenticate(res) {
  res.writeHead(401,
     {'WWW-Authenticate' : 'Basic realm="' + realm + '"'});
  res.end('Authorization required.');
}
```

When we navigate to `localhost:8080` in our browser we are asked to provide a username and password for the `Node Cookbook` realm. If we provide the correct details, our passion for soft cheese is revealed.

How it works...

Basic Authentication works via a series of headers sent between server and browser. When a browser hits the server, the `WWW-Authenticate` header is sent to the browser and the browser responds by opening a dialog for the user to login.

The browser's login dialog blocks any further content from being loaded in the browser until the user either cancels or attempts to log in. If the user presses the **Cancel** button, they see the **Authorization required** message sent with `res.end` in the `authenticate` function.

However, if the user attempts to log in, the browser sends another request to the server. This time it contains an `Authorization` header in response to the `WWW-Authenticate` header. We check for its existence at the top of the `createServer` callback with `req.headers.authorization`. If the header exists, we skip the call to `authenticate` and go on to verify the user credentials. The `Authorization` header looks like this:

```
Authorization: Basic ZGF2ZTpJTGlrZUJyaWVfMzM=
```

The text following `Basic` is a Base64-encoded string that holds the username and password separated by a colon, so the decoded Base64 text is:

```
dave:ILikeBrie_33
```

In our `createServer` callback, we decode the Base64 header by first stripping the `Basic` portion from it, load it into a buffer which converts Base64 to binary, then run `toString` on the result converting it to a UTF8 string.

See `http://en.wikipedia.org/wiki/Base64` and `http://en.wikipedia.org/wiki/Comparison_of_Unicode_encodings` for information on Base64 and string encodings like UTF-8.

Finally, we `split` the login details with a colon, and if the provided username and password match our stored credentials, the user is granted access to authorized content.

There's more...

Basic Authentication comes bundled with the Express framework as middleware.

Basic Authentication with Express

Express (via Connect) provides the `basicAuth` middleware, which implements this pattern for us. To implement the same in Express:

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

var username = 'dave',
  password = 'ILikeBrie_33',
  realm = "Node Cookbook";

var app = express.createServer();

app.use(express.basicAuth(function (user, pass) {
  return username === user && password === pass;
}, realm));

app.get('/:route?', function (req, res) {
  res.end('Somebody likes soft cheese!');
});

app.listen(8080);
```

If we now head to `http://localhost:8080`, our Express server will behave in the same way as our main recipe.

See *Chapter 6*, *Accelerating Development with Express*, for information on using Express to develop web solutions.

See also

- *Setting up a router* discussed in *Chapter 1*, *Making a Web Server*
- *Implementing Digest Authentication* discussed in this chapter
- *Setting up an HTTPS web server* discussed in this chapter

Cryptographic password hashing

Effective encryption is a fundamental part of online security. Node provides the `crypto` module which we can use to generate our own MD5 or SHA1 hashes for user passwords. Cryptographic hashes, such as MD5 and SHA1 are known as message digests. Once the input data has been digested (encrypted), it cannot be put back into its original form (of course if we know the original password, we can regenerate the hash and compare it to our stored hash).

We can use hashes to encrypt a user's password before we store them. If our stored passwords were ever stolen by an attacker, they couldn't be used to log in because the attacker would not have the actual plain text passwords. However, since a hash algorithm always produces the same result, it could be possible for an attacker to crack a hash by matching it against hashes generated from a password dictionary (see the *There's more ...* section for ways to mitigate this).

See `http://en.wikipedia.org/wiki/Cryptographic_hash_function` for more information on hashes.

In this example, we will create a simple registration form, and use the `crypto` module to generate an MD5 hash of a password gained via user input.

As with Basic Authentication, our registration form should be posted over HTTPS, otherwise the password is sent as plain text.

Getting ready

In a new folder, let's create a new `server.js` file along with an HTML file for our registration form. We'll call it `regform.html`.

We'll use the Express framework to provide the peripheral mechanisms (parsing POST requests, serving `regform.html`, and so on), so Express should be installed. We covered more about Express and how to install it in the previous chapter.

How to do it...

First, let's put together our registration form (`regform.html`):

```
<form method=post>
  User  <input name=user>
  Pass <input type=password name=pass>
  <input type=submit>
</form>
```

For `server.js`, we'll require `express` and `crypto`. Then create our server as follows:

```
var express = require('express');
var crypto = require('crypto');

var userStore = {},
  app = express.createServer().listen(8080);

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

`bodyParser` gives us POST capabilities and our `userStore` object is for storing registered user details. In production we would use a database.

Now to set up a GET route as shown in the following code:

```
app.get('/', function (req, res) {
  res.sendfile('regform.html');
});
```

This uses Express' `sendfile` method to stream our `regform.html` file.

Finally, our POST route will check for the existence of `user` and `pass` inputs, turning a user's specified password into an MD5 hash.

```
app.post('/', function (req, res) {
  if (req.body && req.body.user && req.body.pass) {
    var hash = crypto
      .createHash("md5")
      .update(req.body.pass)
      .digest('hex');

    userStore[req.body.user] = hash;
    res.send('Thanks for registering ' + req.body.user);
    console.log(userStore);
  }
});
```

When we use our form to register, the console will output the `userStore` object, containing all registered user names and password hashes.

How it works...

The password hashing portion of this recipe is:

```
    var hash = crypto
      .createHash("md5")
      .update(req.body.pass)
      .digest('hex');
```

We've used the dot notation to chain some `crypto` methods together.

First, we create a vanilla MD5 hash with `createHash` (see the *There's more ...* section on how to create unique hashes). We could alternatively create a (stronger) SHA1 hash by passing `sha1` as the argument. The same goes for any other encryption method supported by Node's bundled `openssl` version (0.9.8r as of Node 0.6.17).

For a comparison of different hash functions, see `http://ehash.iaik.tugraz.at/wiki/The_Hash_Function_Zoo`.

This site labels certain hash functions as broken, which means a weakness point has been found and published. However, the effort required to exploit such a weakness will often far exceed the value of the data we are protecting.

Then we call `update` to feed our user's password to the initial hash.

Finally, we call the `digest` method, which returns a completed password hash. Without any arguments, digest returns the hash in binary format. We pass `hex` (base 16 numerical representation format of binary data, see `http://en.wikipedia.org/wiki/Hexadecimal`) to make it more readable on the console.

There's more...

The `crypto` module offers some more advanced hashing methods for creating even stronger passwords.

Uniquifying hashes with HMAC

HMAC stands for **Hash-based Message Authentication Code**. This is a hash with a secret key (authentication code).

To convert our recipe to using HMAC, we change our `crypto` portion to:

```
var hash = crypto
    .createHmac("md5",'SuperSecretKey')
    .update(req.body.pass)
    .digest('hex');
```

Using HMAC protects us from the use of rainbow tables (pre-computed hashes from a large list of probable passwords). The secret key mutates our hash, rendering a rainbow table impotent (unless an attacker discovers our secret key, for instance, by somehow gaining root access to our server's operating system, at which point rainbow tables wouldn't be necessary anyway).

Hardened hashing with PBKDF2

PBKDF2 is the second version of Password-Based Key Derivation Function, which is part of the Password-Based Cryptographic standard.

A powerful quality of PBKDF2 is that it generates hashes of hashes, thousands of times over. Iterating over the hash multiple times strengthens the encryption, exponentially increasing the amount of possible outcomes resulting from an initial value to the extent that the hardware required to generate or store all possible hashes becomes infeasible.

`pbkdf2` requires four components: the desired password, a salt value, the desired amount of iterations, and a specified length of the resulting hash.

A salt is similar in concept to the secret key in our HMAC in that it mixes in with our hash to create a different hash. However, the purpose of a salt differs. A salt simply adds a uniqueness to the hash and it doesn't need to be protected as a secret. A strong approach is to make each salt unique to the hash being generated, storing it alongside the hash. If each hash in a database is generated from a different salt, an attacker is forced to generate a rainbow table for each hash based on its salt rather than the entire database. With PBKDF2, thanks to our salt, we have unique hashes of unique hashes which adds even more complexity for a potential attacker.

For a strong salt, we'll use the `randomBytes` method of `crypto` to generate 128 bytes of random data, which we will then pass through the `pbkdf2` method with the user-supplied password 7,000 times, finally creating a hash 256 bytes in length.

To achieve this, let's modify our POST route from the recipe.

```
app.post('/', function (req, res) {
  if (req.body && req.body.user && req.body.pass) {
    crypto.randomBytes(128, function (err, salt) {
      if (err) { throw err;}
      salt = new Buffer(salt).toString('hex');
      crypto.pbkdf2(req.body.pass, salt, 7000, 256, function (err,
hash) {
        if (err) { throw err; }
        userStore[req.body.user] = {salt : salt,
          hash : (new Buffer(hash).toString('hex')) };
        res.send('Thanks for registering ' + req.body.user);
        console.log(userStore);
      });
    });
  }
});
```

`randomBytes` and `pbkdf2` are asynchronous, which is helpful because it allows us to perform other tasks or improve the user experience by immediately taking them to a new page while their credentials are being encrypted. This is done by simply placing `res.send` outside of the callbacks (we haven't done this here but it could be a good idea since encryption of this magnitude could take around a second to calculate).

Once we have both our hash and salt values we place them into our `userStore` object. To implement a corresponding login we would simply compute the hash in the same way using that user's stored salt.

We chose to iterate 7,000 times. When PBKDF2 was standardized the recommended iteration count was 1,000. However, we need more iterations to account for technology advancements and reductions in the cost of equipment.

See also

- *Implementing Digest Authentication* discussed in this chapter
- *Setting up an HTTPS web server* discussed in this chapter
- *Generating Express scaffolding* discussed in *Chapter 6*, *Accelerating Development with Express*

Implementing Digest Authentication

Digest Authentication combines Basic Authentication with MD5 encryption, thus avoiding the transmission of plain text passwords, making for a more secure login method over plain HTTP.

On its own, Digest Authentication is still insecure without an SSL/TLS-secured HTTPS connection. Anything over plain HTTP is vulnerable to man in the middle attacks, where an adversary can intercept requests and forge responses. An attacker could masquerade as the server, replacing the expected Digest response with a Basic Authentication response, thus gaining the password in plain text.

Nevertheless, in the absence of SSL/TLS, Digest Authentication at least affords us some defense in the area of plain text passwords requiring more advanced circumvention techniques.

So in this recipe, we will create a Digest Authentication server.

Getting ready

To begin with, we simply have to create a new folder with a new `server.js` file.

How to do it...

As in the Basic Authentication recipe we create an HTTP server, we'll also be using the `crypto` module to handle the MD5 hashing:

```
var http = require('http');
var crypto = require('crypto');

var username = 'dave',
  password = 'digestthis!',
  realm = "Node Cookbook",
  opaque;
```

```
function md5(msg) {
  return crypto.createHash('md5').update(msg).digest('hex');
}

opaque = md5(realm);
```

We've made an `md5` function as a shorthand interface to the `crypto` hash methods. The `opaque` variable is a necessary part of the `Digest` standard. It's simply an MD5 hash of `realm` (as also used in Basic Authentication). The client returns the opaque value back to the server for an extra means of validating responses.

Now we'll create two extra helper functions, one for authentication and one to parse the `Authorization` header as follows:

```
function authenticate(res) {
  res.writeHead(401, {'WWW-Authenticate' : 'Digest realm="' + realm +
'"'
    + ',qop="auth",nonce="' + Math.random() + '"'
    + ',opaque="' + opaque + '"'});

  res.end('Authorization required.');
}

function parseAuth(auth) {
  var authObj = {};
  auth.split(', ').forEach(function (pair) {
    pair = pair.split('=');
    authObj[pair[0]] = pair[1].replace(/"/g, '');
  });
  return authObj;
}
```

Finally, we implement the server as shown in the following code:

```
http.createServer(function (req, res) {
  var auth, user, digest = {};

  if (!req.headers.authorization) {
    authenticate(res);
    return;
  }
  auth = req.headers.authorization.replace(/^Digest /, '');
  auth = parseAuth(auth); //object containing digest headers from
client
  //don't waste resources generating MD5 if username is wrong
  if (auth.username !== username) { authenticate(res); return; }
```

```
    digest.ha1 = md5(auth.username + ':' + realm + ':' + password);
    digest.ha2 = md5(req.method + ':' + auth.uri);
    digest.response = md5([
      digest.ha1,
      auth.nonce, auth.nc, auth.cnonce, auth.qop,
      digest.ha2
    ].join(':'));

    if (auth.response !== digest.response) { authenticate(res); return;
  }
    res.end('You made it!');

  }).listen(8080);
```

Within the browser, this will look exactly the same as Basic Authentication, which is unfortunate as a clear difference between Digest and Basic dialogs could alert the user to a potential attack.

How it works...

When the server sends the `WWW-Authenticate` header to the browser, several attributes are included, consisting of: `realm`, `qop`, `nonce`, and `opaque`.

`realm` is the same as Basic Authentication, and `opaque` is an MD5 hash of the `realm`.

`qop` stands for Quality of Protection and is set to `auth`. `qop` can also be set to `auth-int` or simply be omitted. By setting it to `auth`, we cause the browser to compute a more secure final MD5 hash. `auth-int` is stronger still, but browser support for it is minimal.

`nonce` is a similar concept to a salt, it causes the final MD5 hash to be less predictable from an attacker's perspective.

When the user submits their login details via the browser's authentication dialog, an `Authorization` header is returned containing all of the attributes sent from the server, plus the `username`, `uri`, `nc`, `cnonce`, and `response` attributes.

`username` is the user's specified alias, `uri` is the path being accessed (we could use this to secure on a route by route basis), `nc` is a serial counter which is incremented on each authentication attempt, `cnonce` is the browser's own generated `nonce` value, and `response` is the final computed hash.

In order to confirm an authenticated user, our server must match the value of `response`. To do so, it removes the `Digest` string (including the proceeding space) and then passes what remains of the `Authorization` header to the `parseAuth` function. `parseAuth` converts all the attributes into a handy object, and loads it back into our `auth` variable.

The first thing we do with `auth` is check that the username is correct. If we have no match, we ask for authentication again. This could save our server from some unnecessary heavy lifting with MD5 hashing.

The final computed MD5 hash is made from the combination of two previously encrypted MD5 hashes along with the server's `nonce` and `qop` values and the client's `cnonce` and `nc` values.

We called the first hash `digest.ha1`. It contains a colon (`:`) delimited string of the `username`, `realm`, and `password` values. `digest.ha2` is the `request` method (GET) and the `uri` attribute, again delimited by a colon.

The final `digest.response` property has to match `auth.response` which is generated by the browser, so the ordering and specific elements must be precise. To create our `digest.response` we combine `digest.ha1`, the `nonce`, `nc`, the `cnonce`, `qop`, and `digest.ha2` each separated by a colon. For easy reading we put these values into an array running JavaScript's `join` method on them to generate the final string, which is passed to our `md5` function.

If the given username and password are correct, and we've generated `digest.response` correctly, it should match the browser's `response` header attribute (`auth.response`). If it doesn't, the user will be presented with another authentication dialog. If it does, we reach the final `res.end`. We made it!

There's more...

Let's tackle the logout problem.

Logging out of authenticated areas

There is little to no support in browsers for any official logging out method under Basic or Digest Authentication, except for closing the entire browser.

However, we can force the browser to essentially lose its session by changing the `realm` attribute in the `WWW-Authenticate` header.

In a multiuser situation, if we change our global `realm` variable it will cause all users to log out (if there was more than one). So if a user wishes to log out, we have to assign them a unique realm that will cause only their session to quit.

To simulate multiple users, we'll remove our `username` and `password` variables, replacing them with a `users` object:

```
var users = {
              'dave' : {password : 'digestthis!'},
              'bob' : {password : 'MyNamesBob:-D'},
            },
  realm = "Node Cookbook",
  opaque;
```

Our sub-objects (currently containing `password`) will potentially gain three extra properties: `uRealm`, `uOpaque`, and `forceLogOut`.

Next, we'll modify our `authenticate` function as follows:

```
function authenticate(res, username) {
  var uRealm = realm, uOpaque = opaque;
  if (username) {
    uRealm = users[username].uRealm;
    uOpaque = users[username].uOpaque;
  }
  res.writeHead(401, {'WWW-Authenticate' :
     'Digest realm="' + uRealm + '"'
    + ',qop="auth",nonce="' + Math.random() + '"'
    + ',opaque="' + uOpaque + '"'});

  res.end('Authorization required.');
}
```

We've added an optional `username` parameter to the `authenticate` function. If `username` is present, we load the unique `realm` and corresponding `opaque` values for that user, sending them in the header.

Inside our server callback we replace this code:

```
//don't waste resources generating MD5 if username is wrong
if (auth.username !== username) { authenticate(res); return; }
```

With the following code:

```
//don't waste resources generating MD5 if username is wrong
if (!users[auth.username]) { authenticate(res); return; }

if (req.url === '/logout') {
  users[auth.username].uRealm = realm + ' [' + Math.random() + ']';
  users[auth.username].uOpaque = md5(users[auth.username].uRealm);
  users[auth.username].forceLogOut = true;
  res.writeHead(302, {'Location' : '/'});
  res.end();
  return;
}

if (users[auth.username].forceLogOut) {
  delete users[auth.username].forceLogOut;
  authenticate(res, auth.username);
}
```

We check whether the specified username exists inside our `users` object, saving us from further processing if it doesn't. Providing the user is valid, we check the route (we'll be supplying a logout link to the user). If the `/logout` route has been hit, we set up a `uRealm` property on the logged in user's object and a corresponding `uOpaque` property containing an MD5 hash of `uRealm`. We also add a `forceLogOut` Boolean property, setting it to `true`. Then we redirect the user away from the `/logout` to `/`.

The redirect triggers another request, upon which the server detects the presence of our `forceLogOut` property for the currently authenticated user. `forceLogOut` is then removed from the `users` sub-object to prevent it from getting in the way later. Lastly, we pass control over to the `authenticate` function with the special `username` parameter.

Consequently, `authenticate` includes the user-linked `uRealm` and `uOpaque` values in the `WWW-Authenticate` header, breaking the session.

To finish off, we make a few more trivial adjustments.

`digest.ha1` requires the `password` and `realm` values, so it's updated as follows:

```
digest.ha1 = md5(auth.username + ':'
  + (users[auth.username].uRealm || realm) + ':'
  + users[auth.username].password);
```

The `password` value is fed in via our new `users` object, and the `realm` value is chosen based upon whether our logged-in user has unique realm (a `uRealm` property) set or not.

We change the last segment of our server's code to the following:

```
if (auth.response !== digest.response) {
    users[auth.username].uRealm = realm + ' [' + Math.random() + ']';
    users[auth.username].uOpaque = md5(users[auth.username].uRealm);
    authenticate(res, (users[auth.username].uRealm && auth.username));
    return;
  }
  res.writeHead(200, {'Content-type':'text/html'});
  res.end('You made it! <br> <a href="logout"> [ logout ] </a>');
```

Notice the inclusion of a logout link, the final piece.

New `uRealm` and `uOpaque` attributes are generated if the hashes don't match. This prevents an eternal loop between the browser and server. Without this, when we log in as a valid user and then log out, we'd be presented with another login dialog. If we enter a non-existent user, the new login attempt is rejected by the server as normal. However, the browser attempts to be helpful and falls back to the old authentication details with our first logged-in user and original realm. But, when the server receives the old login details, it matches the user to their unique realm, demanding authentication for `uRealm`, not `realm`. The browser sees the `uRealm` value and matches our non-existent user back to it, attempting to authenticate again as that user, thus repeating the cycle.

By setting a new `uRealm`, we break the cycle because an extra realm is introduced which the browser has no record of, so it defers to the user by asking for input.

See also

- *Implementing Basic Authentication* discussed in this chapter
- *Cryptographic password hashing* discussed in this chapter
- *Setting up an HTTPS web server* discussed in this chapter

Setting up an HTTPS web server

For the most part, HTTPS is the solution to many of the security vulnerabilities such as (network sniffing, and man in the middle) faced over HTTP.

Thanks to the core `https` module. It's really simple to set up.

Getting ready

The greater challenge could be in actually obtaining the necessary SSL/TLS certificate.

In order to acquire a certificate, we must generate an encrypted Private Key, and from that we generate a Certificate Signing Request. This is then passed onto a Certificate Authority (a commercial entity specifically trusted by browser vendors—naturally this means we have to pay for it). Alternatively, the CA may generate your Private Key and Certificate Signing Request on your behalf.

The company, StartSSL provides free certificates. An article about using StartSSL certificates with Node can be found at `https://www.tootallnate.net/setting-up-free-ssl-on-your-node-server`.

After a verification process, the Certificate Authority (CA) will issue a Public Certificate enabling us to encrypt our connections.

We can short cut this process and authorize our own certificate (self-sign), naming ourselves as the CA. Unfortunately, if the CA isn't known to a browser, it will vividly warn the user that our site isn't to be trusted and that they may be under attack. This isn't so good for positive brand image. So while we may self-sign during development, we would most likely need a trusted CA for production.

For development, we can quickly use the `openssl` executable (available by default on Linux and Mac OS X, and we can obtain a Windows version from `http://www.openssl.org/related/binaries.html`) to generate necessary Private Key and Public Certificate:

```
openssl req  -new -newkey rsa:1024 -nodes   -subj '/O=Node Cookbook'
-keyout key.pem -out csr.pem && openssl x509 -req -in csr.pem -signkey
key.pem -out cert.pem
```

This executes `openssl` twice on the command line: once to generate basic Private Key and Certificate Signing Request, and again to self-sign Private Key, thus generating Certificate (`cert.pem`).

In a real production scenario, our `-subj` flag would hold more details, and we would want to acquire our `cert.pem` file from a legitimate CA. But this is fine for private, development, and testing purposes.

Now that we have our key and certificate, we simply need to make our server, so we'll create a new `server.js` file.

How to do it...

Within `server.js` we write the following code:

```
var https = require('https');
var fs = require('fs');

var opts = {key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')};

https.createServer(opts, function (req, res) {
  res.end('secured!');
}).listen(4443); //443 for prod
```

And that's it!

How it works...

The `https` module depends upon the `http` and `tls` modules, which in turn rely upon the `net` and `crypto` modules. SSL/TLS is transport layer encryption, meaning that it works at a level beneath HTTP, at the TCP level. The `tls` and `net` modules work together to provide an SSL/TLS-encrypted TCP connection, with HTTPS layered on top of this.

When a client connects via HTTPS (in our case, at the address `https://localhost:4443`), it attempts a TLS/SSL handshake with our server. The `https` module uses the `tls` module to respond to the handshake in a series of fact-finding interchanges between the browser and server. (For example, what SSL/TLS version do you support? What encryption method do you want to use? Can I have your public key?)

At the end of this initial interchange, the client and server have an agreed shared secret. This secret is used to encrypt and decrypt content sent between the two parties. This is where the `crypto` module kicks in, providing all of the data encryption and decryption functionality.

For us, it's as simple as requiring the `https` module, providing our certificates, then using it just like we would an `http` server.

There's more...

Let's see a few HTTPS use cases.

HTTPS in Express

Enabling HTTPS in Express is just as simple:

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

var opts = {key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')};

var app = express.createServer(opts).listen(8080);

app.get('/', function (req, res) {
  res.send('secured!');
});
```

Securing Basic Authentication with SSL/TLS

We can build anything into our `https` server that we could into an `http` server. To enable HTTPS in our Basic Authentication recipe all we do is alter:

```
https.createServer(function (req, res) {
```

To the following:

```
var opts = {key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')};

https.createServer(opts, function (req, res) {
```

See also

- *Cryptographic password hashing* discussed in this chapter
- *Implementing Basic Authentication* discussed in this chapter

Preventing cross-site request forgery

There's a problem with every browser's security model that, as developers, we must be aware of.

When a user has logged in to a site, any requests made via the authenticated browser are treated as legitimate—even if the links for these requests come from an email, or are performed in another window. Once the browser has a session, all windows can access that session.

This means an attacker can manipulate a user's actions on a site they are logged in to with a specifically crafted link, or with automatic AJAX calls requiring no user interaction except to be on the page containing the malicious AJAX.

For instance, if a banking web app hasn't been properly CSRF secured, an attacker could convince the user to visit another website while logged in to their online banking. This website could then run a POST request to transfer money from the victim's account to the attacker's account without the victim's consent or knowledge.

This is known as a **Cross - Site Request Forgery** (**CSRF**) attack. In this recipe, we'll be implementing a secure HTML login system with CSRF protection.

Getting ready

We'll be securing our Profiler Web App from the *Making an Express Web App* recipe discussed in *Chapter 6*, *Accelerating Development with Express*. We'll want to get a hold of our `profiler` app, with the `profiler/app.js` and `profiler/login/app.js` files open and ready for editing.

Without SSL/TLS encryption, HTML-based logins are subject to at least the same vulnerabilities as Basic Authorization. So for basic security, we'll be adding HTTPS to our app. So we need our `cert.pem` and `key.pem` files from the previous recipe.

We'll also need to have MongoDB running with our stored user data from recipes in *Chapter 6*, *Accelerating Development with Express*, since our `profiler` app relies upon it.

```
sudo mongod --dbpath [PATH TO DB]
```

How to do it...

First, let's secure our entire app with SSL, the top of `profiler/app.js` should look like the following code:

```
var express = require('express')
  , routes = require('./routes')
    , fs = require('fs');

var opts = {key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')};

var app = module.exports = express.createServer(opts);
```

The `admin` section of `profiler` is where a CSRF attack could take place, so let's open up `profiler/login/app.js` and add the `express.csrf` middleware. The top of the `app.configure` callback in `profiler/login/app.js` should look like the following code:

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');

  app.use(express.bodyParser());
  app.use(express.methodOverride());

  app.use(express.cookieParser());
  app.use(express.session({secret: 'kooBkooCedoN'}));

  app.use(express.csrf());
//rest of configure callback
```

Express 3.x.x

Don't forget, in Express 3.x.x the secret goes as a string into `cookieParser` instead of an object into `session`: `express.cookieParser('kooBkooCedoN');`.

The `csrf` middleware is dependent upon the `bodyParser` and `session` middleware, so it must be placed below these.

Now if we navigate to `https://localhost:3000/admin` and attempt to log in (dave, expressrocks), we will receive a `403 Forbidden` response, even though we use correct details.

That's because our login app is now looking for an additional POST parameter called `_csrf` in all of our POST forms, which must match the `_csrf` value stored in the user's `session` object.

Our views need to know the value of `_csrf` so it can be placed in our forms as a hidden element.

We'll use `dynamicHelper` to supply `req.session._csrf` to all login views.

```
app.dynamicHelpers({ //Express 3.x.x would use app.locals.use instead
  user: function (req, res) { //see Chapter 6 for details.
    return req.session.user;
  },
  flash: function (req, res) {
    return req.flash();
  },
  _csrf: function (req) {
    return req.session._csrf;
  }
});
```

Next, we'll create a view called `csrf.jade` in the `login/views` folder as follows:

```
input(type="hidden", name="_csrf", value=_csrf);
```

Now we include `csrf.jade` in each of our POST forms.

`login.jade`:

```
//prior login jade code above
if user
  form(method='post')
    input(name="_method", type="hidden", value="DELETE")
    include csrf
    p Hello #{user.name}!
      a(href='javascript:', onClick='forms[0].submit()') [logout]

  include admin
  include ../../views/profiles


else
  p Please log in
  form(method="post")
    include csrf
    fieldset
      legend Login
//rest of login.jade
```

`addfrm.jade`:

```
fields = ['Name', 'Irc', 'Twitter', 'Github', 'Location',
'Description'];
form#addfrm(method='post', action='/admin/add')
  include csrf
  fieldset
    legend Add
//rest of addfrm.jade
```

Updating and maintaining a site with many different POST forms could pose as challenging. We would have to manually alter every single form. See how we can auto-generate CSRF values for all forms in the *There's More...* section.

Now we can log in, add profiles, and log out without a `403 Forbidden` response.

However, our `/del` route is still susceptible to CSRF. The GET requests are not typically supposed to trigger any changes on the server. They are intended simply to retrieve information. However, like many other apps in the wild, the developers (that's us) were lazy when they built this particular functionality and decided to coerce a GET request to do their bidding.

We could turn this into a POST request and then secure it with CSRF, but what about an app that has hundreds of these deviant GETs?

Let's find out how to protect our `/del` route. In `login/routes/index.js` add the following code:

```
exports.delprof = function (req, res) {
  if (req.query._csrf !== req.session._csrf) {
    res.send(403);
    return;
  };
    profiles.del(req.query.id, function (err) {
      if (err) { console.log(err); }
        profiles.pull(req.query.p, function (err, profiles) {
          req.app.helpers({profiles: profiles});
          res.redirect(req.header('Referrer') || '/');
        });
    });

}
```

Our changes make it so we can't delete profiles until we include `_csrf` in the query string, so in `views/admin.jade`:

```
mixin del(id)
  td
    a.del(href='/admin/del?id=#{id}&p=#{page}&_csrf=#{_csrf}')
      &#10754;
//rest of admin.jade
```

How it works...

The `csrf` middleware generates a unique token which is held in the user's session. This token must be included in any actioning requests (logging in, logging out, adding or deleting) as an attribute named `_csrf`.

If the `_csrf` value in the request body (or query string for GET) doesn't match the `_csrf` token stored in the `session` object, the server denies access to that route and therefore prevents the action from occurring.

How does this prevent a CSRF attack? In a plain CSRF exploit, the attacker has no way of knowing what the `_csrf` value is, so they are unable to forge the necessary POST request.

Our `/del` route protection is less secure. It exposes the `_csrf` value in the address, potentially creating a very small, but nonetheless plausible, window of opportunity for an attacker to grab the `_csrf` value. This is why it's best for us to stick with the POST/DELETE/PUT requests for all action-related endeavors, leaving GET requests for simple retrieval.

Cross-site scripting (XSS) circumvention

This protection is rendered moot in the event of an accompanied XSS exploit, whereby an attacker is able to implant their own JavaScript within the site (for example, through exploiting an input vulnerability). JavaScript can read any elements in the page it resides on, and view non-HttpOnly cookies with `document.cookie`.

There's more...

We'll take a look at a way to automatically generate CSRF tokens for login forms, but we should also bear in mind that CSRF protection is only as good as our ability to code tightly.

Auto-securing the POST forms with the CSRF elements

Ensuring that all the POST forms in our app contain a hidden `_csrf` input element could be an arduous task on a site of any significant scale.

We can interact directly with some Jade internals to automatically include these elements.

First, in `login/app.js` we add the following setting to our configuration:

```
  app.set('view options', {compiler: require('./
customJadeCompiler')});
```

Express allows us to push particular options to whatever view engine we are using. One of the Jade options (our view engine) is `compile`, which allows us to define our own custom Jade interpreter.

Let's create `customJadeCompiler.js` placing it in the `login` directory.

First, we'll require some modules and set up our new compiler class as follows:

```
var jade = require('jade');
var util = require('util');

//inherit from Jades Compiler
var CompileWithCsrf = function (node, options) {
  jade.Compiler.call(this, node, options);
};
```

Next we use `util.inherits` to inherit our new compiler's prototype from the Jades compiler.

```
//inherit from the prototype
util.inherits(CompileWithCsrf, jade.Compiler);
```

Then we modify the Jade's internal `visitTag` method (which we've inherited from `jade.Compiler`):

```
CompileWithCsrf.prototype.visitTag = function (tag) {

   if (tag.name === 'form' && tag.getAttribute('method').match(/
post/i)) {

    var csrfInput = new jade.nodes.Tag('input')
      .setAttribute('type', '"hidden"')
      .setAttribute('name', '"csrf"')
      .setAttribute('value', '_csrf');

    tag.block.push(csrfInput);

  }
  jade.Compiler.prototype.visitTag.call(this, tag);
};
```

Finally, we load our new compiler into `module.exports` so it's passed via `require` to the `compiler` option of the `view options` setting in `app.js`:

```
module.exports = CompileWithCsrf;
```

We create a new class-type function, applying the `call` method to `jade.Compiler`. When we pass the `this` object to the `call` method, we essentially inherit the main functionality of `jade.Compiler` into our own `CompileWithCsrf` class-type function. It's a great way to re-use code.

However, `jade.Compiler` also has a modified prototype which must be incorporated into our `CompileWithCsrf` in order to fully mimic `jade.Compiler`.

We used `util.inherits`, but we could instead have said:

```
CompileWithCsrf.prototype = new jade.Compiler();
```

Or:

```
CompileWithCsrf = Object.create(jade.Compiler);
```

Or even:

```
CompileWithCsrf.prototype.__proto__ = jade.Compiler.prototype;
```

`Object.create` is the Ecmascript5 way, `new` is the old way and `__proto__` is the non-standard way that should probably be avoided. These all inherit the additional methods and properties of `jade.Compiler`. However, `util.inherits` is preferred because it also adds a special `super` property containing the object we originally inherited from.

Using `call` and `util.inherits` essentially allows us to clone the `jade.Compiler` object as `CompileWithCsrf`, which means we can modify it without touching `jade.Compiler` and then allow it to operate in place of `jade.Compiler`.

We modify the `visitTag` method, which processes each tag (for example, `p`, `div`, and so on) in a Jade view. Then we look for the `form` tags with methods set to `post`, using a regular expression since the `method` attribute may be in upper or lower case, being wrapped in double or single quotes.

If we find a `form` with POST formatting, we use the `jade.Nodes` constructor to create a new input `node` (a Jade construct, in this case rolling as an HTML element), which we then call `setAttribute` (an internal Jade method) on three times to set the `type`, `name` and `value` fields. Notice `name` is set to `'"_csrf"'` but `value` contains `'_csrf'`. The inner double quotes tell Jade we intend a string. Without them, it treats the second parameter as a variable, which is exactly what we want in the case of `value`. The `value` attribute is therefore rendered according to `_csrf` `dynamicHelper` defined in `app.js` (which is likewise taken from `req.session._csrf` as generated by the `express.csrf` middleware).

Now that our `_csrf` tokens are automatically included in every POST form, we can remove the `csrf.jade` includes from the `login.jade` and `addfrm.jade` views.

Eliminating cross-site scripting (XSS) vulnerabilities

Cross-site scripting attacks are generally preventable, all we have to do is ensure any user input is validated and encoded. The tricky part comes where we improperly or insufficiently encode user input.

When we take user input, much of the time we'll be outputting it to the browser at a later stage, this means we must embed it within our HTML.

XSS attacks are all about breaking context. For instance, imagine we had some Jade that links to a user profile by their username:

```
a (href=username)  !{username}
```

This code is exploitable in two ways. First, we used `!{username}` instead of `#{username}` for the text portion of our anchor link element. In Jade `#{}`interpolation escapes any HTML in the given variable. So if an attacker was able to insert:

```
<script>alert('This is where you get hacked!')</script>
```

As their username, `#{username}` would render:

```
&lt;script&gt;alert('This is where you get hacked!')&lt;/script&gt;
```

Whereas, `!{username}` would be unescaped (for example, HTML would not be replaced by escape characters like `<` in place of <). The attacking code could be changed from an innocent (though jaunty) `alert` message, to successfully initiated Forged Requests, and our CSRF protection would be futile since the attack is operating from the same page (JavaScript has access to all data on the page, and the attacker has gained access to our page's JavaScript via XSS).

Jade HTML-escapes variables by default, which is a good thing. However, proper escaping must be context aware, and simply converting HTML syntax into its corresponding entity codes is not enough.

The other vulnerable area in our bad Jade code is the `href` attribute. Attributes are a different context to simple nested HTML. Unquoted attributes are particularly susceptible to attack, for instance, consider the following code:

```
<a href=profile>some text</a>
```

If we could set `profile` to `profile onClick=javascript:alert('gotcha')`, our HTML would read:

```
<a href=profile onClick=javascript:alert('gotcha')>some text</a>
```

Again, Jade partially protects us in this sense by automatically quoting variables inserted to attributes. However, our vulnerable attribute is the `href` attribute, which is another sub context of the URL variety. Since it isn't prefixed with anything, an attacker might input their username as `javascript:alert('oh oh!')` so the output of:

```
a (href=username)  !{username}
```

Would be:

```
<a href="javascript:alert('oh oh!')">  javascript:alert('oh oh!') </a>
```

The `javascript:` protocol allows us to execute JavaScript at the link level, allowing a CSRF attack to be launched when an unsuspecting user clicks a malicious link.

These trivial examples are elementary. XSS attacks can be much more complex and sophisticated. However, we can follow the Open Web Application Security Projects 8 input sanitizing rules that provide extensive protection against XSS:

`https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`

Validator module

Once we understand how to clean user input, we could use regular expressions to quickly apply specific validation and sanitization methods. However, for a simpler life, we could also use the third-party `validator` module which can be installed with `npm`. Documentation is available on the Github page: `https://www.github.com/chriso/node-validator`.

See also

- *Setting up an HTTPS web server* discussed in this chapter
- *Initializing and using a session* discussed in *Chapter 6, Accelerating Development with Express*
- *Making an Express web app* discussed in *Chapter 6, Accelerating Development with Express*