

Authentication and Authorization

Terminology

- **Authentication:** Checking to see if someone is who they say they are
 - Example: Username / Password, Biometrics, etc.
- **Authorization:** Checking to see if the person is allowed to use a resource
 - Example: Only you should be able to edit your profile on a social network, only an administrator should see all account usernames, etc.

Authentication on the Web

HTTP Basic Authentication:

- When a server sees a request come in to something that should be protected, it can respond with an HTTP 401 (Not Authorized)
- In addition, it places the following in the response **header**

```
WWW-Authenticate: Basic realm="insert realm"
```

It is then up to the browser to display some mechanism for the user to enter their username and password (for the given “realm”).

Authentication on the Web

Once the browser gets the user's credentials, they are sent with a new HTTP request inside a request header field:

Authorization: Basic sfrees:thisisnotmypassord

Actually - its not sent exactly like that... the username:password string concatenation is turned into Base64 binary format first

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

Note - this is reversible - and easily stolen. Its actually superfluous.

Basic Authentication

- The advantage of this mechanism is that it can be stateless from the server's perspective
- The browser can cache username/passwords for a given “realm”, and only actually ask the user at reasonable time intervals
- It can continue to send the username/password with each request, so the server can authenticate each and every HTTP request.

Basic Authentication

The problems are significant though...

- First - if your site is using basic authentication, you'll need to either
 - Use the browser's built in username/password dialog (probably not very professional looking)
 - Implement your own UI for username/password and stuff the result in the header fields (can be tricky).
- Second - the username / password are easily stolen, since they are in plain text.

Lets deal with the second one first....

Encryption

Encryption is typically a method in which a string (or binary) can be obfuscated in some way.

- Encryption is *reversible*, in that you can take the result and apply some sort of algorithm to get the original content back.
- Since its a reversible process, its only as secure as long as the reversal process is a secret.
- Examples range from simple (rot13 cypher) to the very complex RSA encryption scheme

Encrypting HTTP traffic

If we want to protect our HTTP requests and response from prying eyes, we can encrypt it.

- Browser encrypts request
- Server decrypts request
- Server encrypts response
- Browser decrypts request

So... how do the client and server communicate the “reversal” mechanism?

TLS / SSL Protocols

- If the client and server are going to encrypt data, they both need to know how to decrypt each other's messages.
- Encryption / Decryption usually done with a “key”
- You can think of this simply as
message + encryption key -> **message digest**
message digest + decryption key -> original message
- The encryption and decryption keys could be the same (symmetric keys) or different (asymmetric)

TLS / SSL Protocols

- Symmetric keys typically aren't workable on the general internet, since the keys would need to be exchanged (think about that for a moment...)
- On the general internet, asymmetric keys can be used, where a set of keys is used for **each** way (encryption/decryption keys for client to server, and encryption/decryption keys for server to client)

TLS / SSL Protocols

On connection:

- Server send the client an encryption key for sending data to the server. Server has decryption key.
- Client will respond with an encryption key for the server to use when sending data back to the client. Only the client has the corresponding decryptions.

In addition, typically the server will exchange a *certificate* to prove it is who the client thinks it is.

- A certificate authority can be used by the client to verify the certificate (a string) matches the IP address of the server.
- Some authorities include Verisign/Symantec and GoDaddy

TLS / SSL Protocols

Once the entire handshaking process is complete, the “session” between the client and server is secure.

The browser will display the url as starting with **https** instead of http.

Basic Authentication

The problems are significant though...

- First - if your site is using basic authentication, you'll need to either
 - Use the browser's built in username/password dialog (probably not very professional looking)
 - Implement your own UI for username/password and stuff the result in the header fields (can be tricky).
- ~~Second - the username / password are easily stolen, since they are in plain text.~~
 - At this point, you could use Basic Authentication, or you could actually just handle the username / password yourself (as an HTML form)
 - This tends to be more popular, people like fancy login forms.

Authentication, at the server

- So now imagine you are the server - you got a username/password from a client.
- The only way to authenticate is to check to see if they match “your records”
- Your records could be
 - a variable in the application (yikes)
 - a file (eh)
 - a database (most likely)

Is it as simple as storing a username/password in the database?

Clear-text passwords



Never, ever, store passwords in plain text anywhere!

Your files and databases are only as secure as the vendors who made them.

If the passwords are valuable, you **will** be hacked.
Ask Target, LinkedIn, Adobe, and hundreds more.

Encrypted passwords?



Encrypted passwords are nearly as bad!

- Again, the encryption process is reversible, so you just need to steal the key.
- While in HTTPS, the key would give you one session, here the key gives you ALL password!
- More valuable, more likely to be hacked.

Answer: Hashing

Hashing is a **one way** algorithm, which also uses a key, but is not reversible.

- Common Hashing algorithms include SHA and MD5.

Password + Key = message digest.

Key + message digest != Password.

The critical point here is even if someone steals the encrypted password, AND the key, they still don't know the regular password!

Hashing passwords

- Server creates an account by storing the username and hashed password in a DB
 - sfrees, 7xf7dfkasmxl38dlsag9
- When a username/password comes across an HTTP session (encrypted enroute by SSL!), the plain text password is hashed.
 - thisisnotmypassword + key = 7xf7dfkasmxl38dlsag9
- If the hashed values match, then the password was correct.

Vulnerabilities

- The only vulnerability of a hashing algorithm is collisions and guessing.
- If multiple message hash to the same value, its easier to just simply guess!
 - This is incredibly rare with SHA algorithms
- However - if you do know the key, you could simply hash all possible “usernames” using the same algorithm, and then see what turns up!
 - These are called “Rainbow Tables”

Salt

- Rainbow tables are easy to thwart though.
- Salt is a randomly selected sequence of bytes that will be registered with the username at creation.
- Store username, salt, and digest
 - $\text{digest} = \text{username}:\text{salt} + \text{key}$
- Now, you can't compute a rainbow table for all usernames, you need a rainbow table for all username/salt combinations.
- If your salt is 128 bits, it will take centuries :)

Final Answer...

- When storing usernames and passwords, all passwords should be **salted** and hashed using the best possible algorithm (SHA-512).
- Anytime a user sends usernames and passwords over HTTP as form data, it should be over HTTPS to ensure that it is encrypted.
 - Since SSL Certificates are expensive, people skip this when doing development. Beware of no SSL on production though...

Using Authentication

Once a user is authenticated, you don't want to keep authenticating on each page.

The common approach is to store a “user” object in the session after login.

- The user object can be checked to **authorize** the use of resources - until the session expires.
- Redirect any traffic without the session user to the login page.

Other solutions exist as well, including using Basic Authentication, or Digest Authentication

Simple implementation

Authentication in Express

passport is a popular module to assist with the standard workflow of authentication

passport can be used to authenticate against OAuth and other 3rd-party services (i.e. Facebook) as well.

npm install passport

Then, in app.js

```
var passport = require('passport')  
var LocalStrategy = require('passport-local').Strategy;
```


Using passport

Before you configure your router, add passport as middleware

```
app.use(passport.initialize());  
app.use(passport.session()); // there are other options
```

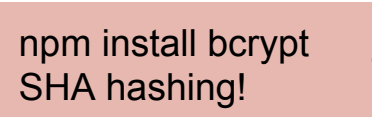
Then you need to tell passport how to authenticate a given user

Performing the hash

... assume “password” is the hashed version of the user’s password
and pswd is what the user just typed.

```
passport.use(new LocalStrategy(  
  { usernameField: 'username', passwordField: 'password' },  
  function(user, pswd, done) {  
    if ( user !== username ) {  
      console.log("Username mismatch");  
      return done(null, false,);  
    }  
    bcrypt.compare(pswd, password, function(err, isMatch) {  
      if (err) return done(err);  
      done(null, isMatch);  
    });  
  }  
));
```

Performing the hash (typical)

```
passport.use(new LocalStrategy(  
  function(username, password, done) {  
    lookup_user_in_my_db({ username: username }, function(err, user) {  
      if (err) { return done(err); }  
      if (!user) {  
        return done(null, false, { message: 'Incorrect username.' });  
      }  
       bcrypt.compare(password, user.password,  
        function(err, isMatch) {  
          if (err)  
            return done(null, false,  
              { message: 'Incorrect password.' });  
  
          return done(null, user);  
        });  
    });  
  });  
});
```

Logging in

Now when you have post hit your login route....

```
app.post('/login',  
  passport.authenticate('local',  
    { successRedirect: '/home',  
      failureRedirect: '/' }));
```

- Passport will grab the username / password fields from the form, call the previous slide's authentication function
- redirect to the /home page if successful -
 - **It will also add “user” to all requests hereafter**
- otherwise to /

Authorization

Express supports authorization out of the box, using middleware on its routes.

Route middleware is a function that accepts req and res, but also a “next” callback. It is plugged in before your normal route handler.

Authorization

```
function restrict(req, res, next) {  
  if (req.user) {  
    next(); // proceed to actual route handler  
  } else {  
    req.session.error = 'Access denied!';  
    console.log("Unauthorized page access");  
    res.redirect('/');  
  }  
}
```

Now, in your route setup:

```
app.get('/start', restrict, routes.start);
```

Poor demo...

Lets do something a bit silly, and make our guessing game password protected.

- The only user will be “guest”, and the password is “guest” as well.
- Instead of a DB, we'll just use some global variables.
- Unless the user has logged in, they can't access any of the pages...