codecademy

Password Authentication

Cryptography

Cryptography is the process of encrypting and decrypting data in order to keep that data safe when storing or transmitting it.

- **Encryption** is a way of *hiding* data by converting it to an encoded format.
- **Decryption** is a way of *revealing* encrypted data by decoding it from its encoded format.

Symmetric Vs. Asymmetric Encryption

Ciphers can be symmetric or asymmetric.

- **Symmetric** encryption uses the *same* key to encrypt and decrypt information.
- **Asymmetric** encryption uses a *public* key to encrypt data and a different *private* key to decrypt data.

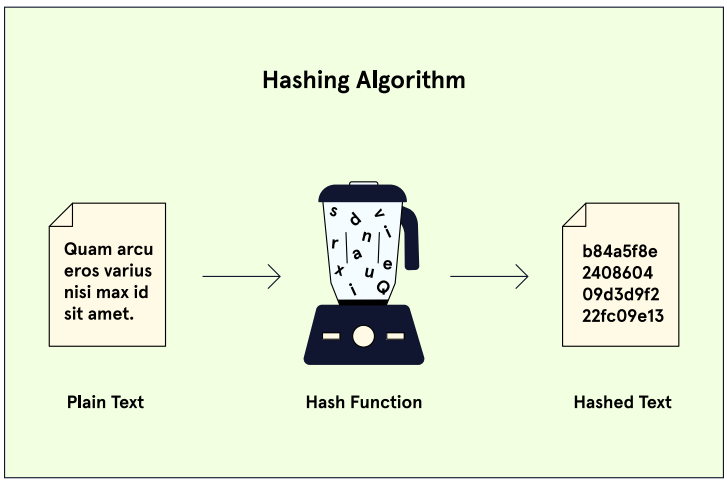
Asymmetric ciphers can be slower than symmetric ciphers but have additional use-cases in authentication and non-repudiation.

Hashing

Hashing is a *one-way process* that takes data of any size and represents it as a *unique* hash value of a fixed size. No matter how large or complex your file is, hashing provides a fast, reliable way to compare files and verify their authenticity.

Hashing lets you check if two pieces of information are the same, without knowing what the information itself actually is.

Hashing can be used to store sensitive data in a *secure* way.



Encoding

Encoding transforms data into a form that can be used by a different type of system.
It is NOT a way to secure data because encoded information is easily reversed and only requires knowledge of the algorithm used to decode information.
This example shows part of a basic ASCII table.

Letter	Decimal Encoding	Binary Encoding
A	65	100 0001
B	66	100 0010
C	67	100 0011

Obfuscation

Obfuscation means to hide the meaning of something by making it difficult to understand.
Programmers might do this to hide the meaning of code or make it harder for users to hack. Malicious actors might also do this to make it harder to discover software is actually a virus!

Passport's Local Strategy

The `LocalStrategy` object will take in an anonymous function with the parameters: `username`, `password`, and the callback function `done`.
`done` takes in:

- An error or `null` if no error found
- A user or `false` if no user found

`done` supplies a user to Passport if a user is authenticated. The anonymous function will:

1. Verify login
2. If login details are valid, done is invoked and the user is authenticated
3. If the user is not authenticated, pass `false` into done

```
passport.use(new LocalStrategy(
  function (username, password, done) {

    db.users.findByUsername(username,
(err, user) => {
      if(err) return done(err);

      if(!user) return done(null, false);

      if(user.password != password) return
done(null, false);

      return done(null, user)
    });
  })
);
```

Serializing & Deserializing Users with Passport

`serializeUser()` takes a user id and stores it internally on `req.session.passport` which is Passport's internal mechanism to keep track of things.

`deserializeUser()` uses the `id` to retrieve the data and return an object with data.

If there is an error, we return the error.

If there is no error the first argument in `done()` is `null`, and the second argument is the user id, the value that we want to store in our Passport's internal session.

```
passport.serializeUser((user, done) => {
  done(null, user.id);
});

passport.deserializeUser((id, done) => {
  db.users.findById(id, function (err,
user) {
    if (err) return done(err);
    done(null, user);
  });
});
```

Logging In with `passport-local`

We can process the authentication and, if successful, serialize the user for us. This is shown in the `app.post("/login" ...); code.

`passport.authenticate()` is middleware and takes in:

- Which strategy to employ. We should use `"local"`.
- An object with `failureRedirect` set to `"/login"`. This will redirect to `"/login"` if login fails.

If successful, we go to the user's profile.

The "/profile" endpoint uses the serialized user found in `req.user` and renders the `profile` page.

```
app.post("/login",
  passport.authenticate("local", {
failureRedirect : "/login"}),
  (req, res) => {
    res.redirect("profile");
  }
);

app.get("/profile", (req, res) => {
  res.render("profile", { user: req.user
});
});
```

Registering User's with `passport-local`

We will create an asynchronouse route handler using `async/await`. We'll retrieve the user data from `req.body` and `await` as we call our helper function to create the new user.

- If a `newUser` is successfully created, we send a status code of `201` and a relevant `msg` response.
- If there is an error, we return a status code of `500`.

> NOTE: In a real development environment, passwords should be hashed whenever a user registers.

```
app.post("/register", async (req, res) =>
{
  const { username, password } = req.body;
  const newUser = await
db.users.createUser({ username, password
});
  if (newUser) {
    res.status(201).json({
      msg: "Insert Success Message Here",
      newUser
    });
  } else {
```

```
    res.status(500).json({ msg: "Insert
Failure Message Here" });
  }
}
```

Logging Out using `passport-local`

`req.logout` will log a user out of the application.
`res.redirect("/login");` will redirect the user to the login page.
By terminating the session, the user will have to re-authenticate in order to create a new session.

```
app.get("/logout", (req, res) => {
  req.logout();
  res.redirect("/login");
});
```

Hashing Passwords with `bcrypt.js`

You can generate a salt and hash with `bcrypt.js` using 3 steps:

- Generate Salt.

The built-in `genSalt()` function automatically generates a salt for us.

- Hash Password

`bcrypt.hash()` takes in a `password string` and a `salt`. We `await` and `return` this function call since it will return the hashed password.

- Return `null` if there's an error

In the `catch` block, we print out the error. Then, we `return null`.

```
const passwordHash = async (password,
saltRounds) => {
  try {
    const salt = await
bcrypt.genSalt(saltRounds);
    return await bcrypt.hash(password,
salt);
  } catch (err) {
    console.log(err);
  }
  return null;
};
```

Verifying a Password Using `bcrypt.js`

The process of comparing passwords is:

- Retrieve Plaintext Password
- Hash Password
- Compare Hashed Password with Hash Stored in database

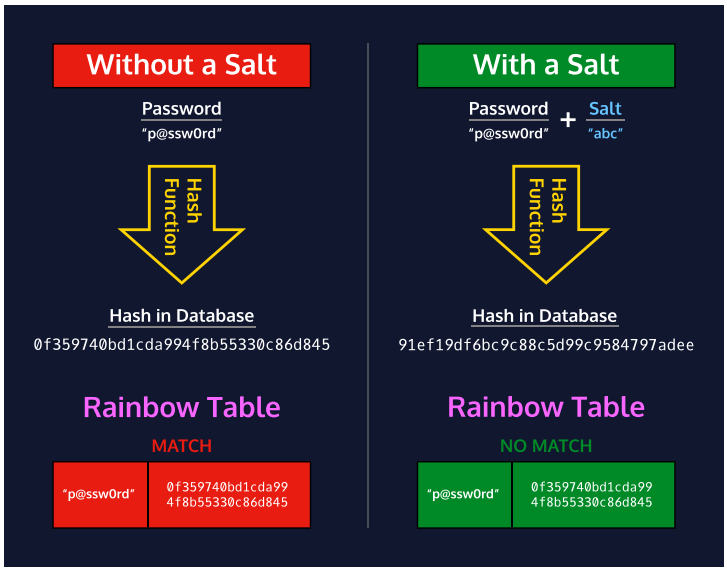
We use `bcrypt.compare()` to compare the provided password, `password`, and the stored hashed password, `hash`. `bcrypt.compare()` can deduce the salt from the provided hash and hashes the provided password for comparison.
The `return` value will be `true` if the provided password is valid.
We `return false` outside of the `try / catch` block if there is an error.

```
const comparePasswords = async (password,
hash) => {
  try {
    const matchFound = await
bcrypt.compare(password, hash);
    return matchFound;
  } catch (err) {
    console.log(err);
  }
  return false;
};
```

codecademy

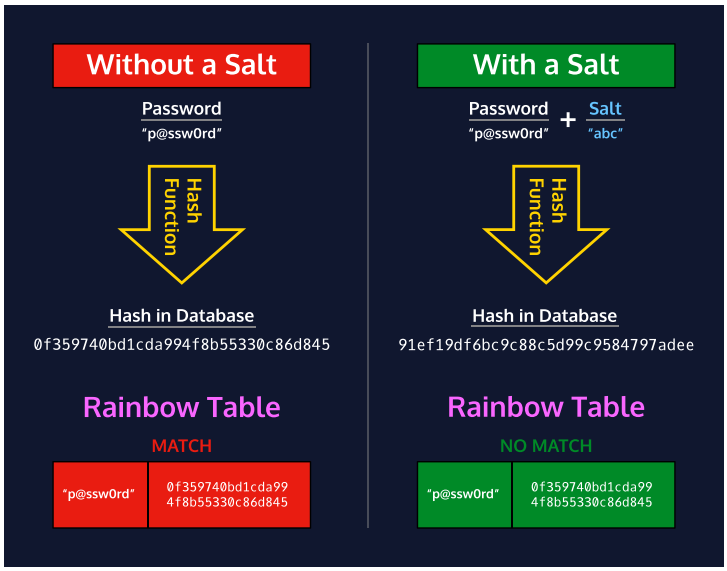
Rainbow Tables

A *rainbow table* is a massive table of common passwords and password-hash combinations used by attackers to break into accounts. One common technique we can take to protect ourselves from rainbow table attacks is the use of salts.



Salts

A *salt* is a secret random string that is combined with a password prior to hashing specifically to defend against the use of rainbow tables.
Rainbow tables are large lookup databases that consist of pre-computed password-hash combinations which correlate plaintext passwords with their hashes.



Save Print Share ▾