Learning Objectives

- **Identify the ways in which Django is secure by default**
- **Define hashing and explain why it is important**
- **Define salting**
- **Explain how Django stores passwords**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Built-In Security

Django's Built-In Security & OWASP Top 10

When building with a web framework, or any software that might be accessed by untrusted users or exposed to the internet, it's good to know what security features it provides, or lacks. Or maybe you've come from another programming language that isn't as secure as Django is by default, and you're wondering why there are some steps that are required in other languages but not in Django.

One example might be PHP. If you're not using a framework, you need to be very careful to escape HTML strings before outputting them; similarly falling back to low-level database functions can lead to SQL injection attacks. Django prevents these with HTMl escaping by default in templates; and its ORM, respectively.

A good primer for web security is the Open Web Application Security Project® (OWASP) Top 10, which is a list of the top ten web application security risks. Some of them are not risks to your application unless you're building some specific types of tools, but we'll discuss which one Django mitigates.

Injection

Injection flaws are usually attacks against SQL, but could also affect NoSQL databases, files or executed commands. Django protects against SQL injection with its ORM, but if you decided to write raw SQL to execute queries, be sure to parameterize any input (if you're having to write raw SQL you probably know how to do this). When using other databases (for example, Cassandra or MongoDB) be sure to use up-to-date libraries to connect and run queries – these should handle escaping for you.

Broken Authentication

Broken authentication refers to any way that your application could be broken into by exploiting its authentication system. This could involve brute-force login attempts, leaving default passwords enabled, allowing weak passwords, or many other methods. Django has protection for some of these: it doesn't have any default passwords set up, and blocks a user from setting a weak password. But, it doesn't prevent brute-force logins or have built-in support for two-factor authentication. A tool like Django Defender can help with repeated login attempts but if you want to set up 2FA it will depend on your 2FA provider and method.

Sensitive Data Exposure

Solving the problem of Sensitive data exposure can be difficult. Django stores passwords with strong hashing/salting methods by default (more on this later), but how you store other sensitive data is very domain-specific. Rather than storing credit card numbers, for example, you should tokenize them for rebilling and only store the token. Other sensitive data could be stored in a seperate database, like Vault, so if an attacker gets access to your main database that data is still encrypted (maybe – assuming it's set up correctly).

It's also important to use SSL/TLS to encrypt data during transit. Django can help ensure this by adding the setting `SECURE_SSL_REDIRECT = True`, which forces a redirect from the `http` to `https` version of the URL. Your web server will need to be configured correctly for SSL for this to work though.

XML External Entities (XXE)

XML External Entities is something you only need to consider if working with XML documents. It allows specially-crafted XML to include other resources inside it. For example, you can create an XML that, when rendered on the server, will execute and include arbitrary files you specify. This could allow an attacker to read private data files on the server. If you do work with XML files, consider using defusedxml which prevents these attacks.

Broken Access Control

If your application has broken access control, it's probably not a fault of Django itself. Instead, someone might have written a view that doesn't correctly check for a valid user. Instead of checking inside a view, consider using the `login_required` decorator, which will also automatically redirect to a login page. This will make it clear which views have or haven't had this protection applied.

But it's not a magic bullet. Even if you check that a user has logged in, you need to make sure you're fetching objects that belong to that user and not someone else. This basically comes down to being fastidious with code and encouraging code review from your team to make sure nothing has been missed.

Security Misconfiguration

Security misconfiguration is "the most commonly seen issue", according to OWASP. Django is secure by default, and doesn't send stack traces or verbose information to users.

But it's only one piece of the puzzle, your whole infrastructure needs to be secure, have patches applied, not expose any ports it shouldn't, and so on. If you're unsure about configuration, have someone experience help out.

Cross Site Scripting (XSS)

We discussed XSS in Module 1 when we were building our custom template tags and filters. As we saw, Django is safe by default in how it outputs strings in templates. But if you're building custom filters that generate HTML, be sure to use the `format_html` function. If you do need to output HTML from untrusted sources, use something like the Bleach library to allow only trusted tags and attributes.

Insecure Deserialization

Insecure deserialization is not something Django itself is vulnerable to. However if you do write your application to accept serialized data from a user, you should be very careful about trusting it, and make sure it only contains absolute minimum necessary amount of data. Check out the OWASP page on this for some example attacks.

Using Components with Known Vulnerabilities

Some components, libraries and tools have known vulnerabilities: published ways they can be exploited. The Django project keeps a list of CVEs (Common Vulnerabilities and Exposures), which is here as of Django 3.2.

This is not something Django can proactively prevent against – if they knew they had security vulnerabilities then they would fix them! Instead, it's up to the end user to keep Django, other Python libraries, and the OS, up to date and patched.

Insufficient Logging and Monitoring

The right logging and monitoring can help mitigate the severity of attacks by being able to respond to them quickly. This kind of monitoring is not something Django has built in, and it might not be the best place to have these kind of checks. You could use an Intrusion Detection System to help with alerting, but logging admin logins, failed login attempts, and other suspicious requests, as well as having a process for reviewing the logs regularly, is a good process to have in place.

More about Django's security

A great primer on Django's security is its own documentation: Security in Django. This is a must read to understand in more details how Django prevents against specific types of attacks, as well as giving a jumping off

point for more specific resources.

If you're unsure about the security of your project or infrastructure, it's probably better to pay a security expert to audit it than have you or your customer's information compromised.

Hashing

Hashing Intro

Before reading this section, please understand that it is designed to give background on how Django stores passwords, and not advice on how to write your own password storage method. We will go through a few examples of insecure methods of doing it, so that you know what not to do. In short, trust Django's built-in methods, and don't write your own. If you're not interested in how passwords are stored you can skip this section, but it's a good primer into some of the issues which might seem like a straightforward problem.

When a user wants to log into Blango (or just about any other website), they put in their username and password. How does the website validate that these are correct?

One way might be to store the username and password in the database, but this is a **terrible idea**, never do this! Sure, it makes it extremely simple to check that the credentials are correct, just compare the username and password and make sure they match exactly, but it's a security nightmare.

If your database were to leak, then all your users' passwords are exposed. Many people use the same username and password for multiple websites, and so the attacker would have access to any of them too (although the increasing prevalence of password managers has reduced this risk).

Instead of storing the password in plain text, we should *hash* it. A hash function is a one-way method of encoding an arbitrary length data to a short string of bytes. The length of the output (the *hash*) is always the same for a particular hash function, regardless of the length of the input. The chances of two pieces of data having the same hash is tiny (for a good hash function), and so you can be pretty sure that a hash is an accurate proxy for the original data. Note that hashing is not encryption, as the hashed data can't be decrypted to get the original data back (not in the traditional sense anyway, more on this later).

Here's an example of created some SHA-256 hash in Python. Notice that their digests (more commonly referred to as the "hash") are all the same length:

```
import hashlib

print(hashlib.sha256(b"password").hexdigest())
print(hashlib.sha256(b"hello wold").hexdigest())
print(hashlib.sha256(b"password123").hexdigest())
```

Your code should print:

```
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
03457c36995bc52df8fa0430393535842195c72ffff555a9febdf2098c959ce8
ef92b778bafe771e89245b89ecbc08a44a4e166c06659911881f383d4473e94f
```

A better way of storing passwords is to hash them first. So for a user, instead of storing *password123* we would store *ef92b778bafe771e89245b89ecbc08a44a4e166c06659911881f383d4473e94f*. When a user logs in we can hash the password they provide and then check it matches the hash we stored in the database. This means if our database is compromised then only the hash leaks, and not the users' passwords. But this is still not secure!

To give an example, if we allowed the user to have the password *password*, we'd store the string *5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8*.

As we mentioned earlier, hashes can't be "decrypted" as such, but some people have run many words and common passwords through a hash function, and then stored the result in a database. It is then possible to search for the hash and be shown the input word. For example, a Google search for the hash above (*5e88489...*) will return pages that show the original input. This type of database is called a Rainbow Table. So if our database leaked, the attacker could run it through a rainbow table and extract a lot of common passwords from our users.

The next level of security is to add a *salt*: a small amount of random data that's added to the password before it's hashed, when it's first set. The salt is stored unhashed, and is read and reused on password validation.

For example, let's say we generate the salt `abc123` when the user sets their password/registers. The password hash would be generated like this:

```
import hashlib

print(hashlib.sha256(b"abc123" + b"password123").hexdigest())
```

Your code should print:

```
43940af87374ecbbdc596a2842d0eb8c0a3f9edc31b6fdaefdb6437f6f61f484
```

We'd store a string like `abc123$43940af87374ecbbdc596a2842d0eb8c0a3f9edc31b6fdaefdb6437f6f61f484` in the database.

When the user tries to log in, we'd fetch this salt + hash from the database. Then we'd extract the salt (the bit before the `$`), prepend it to the password they supplied, and hash that string. We know the password is correct if that resulting hash matches the bit *after* the `$`.

This prevents rainbow table attacks: a Google search for `43940af87374ecbbdc596a2842d0eb8c0a3f9edc31b6fdaefdb6437f6f61f484` doesn't currently return anything.

Our security is getting stronger, but most hash functions have a problem: they are designed to be fast – this is probably the only time that we'd say fast code is a "problem"! We can improve our security more by changing our hash function and using multiple iterations. We'll return to a new hash function in the next section, but let's talk about why the number of iterations is important now.

If it's fast to validate a password, then it can be fast to brute-force login attempts. Let's say that it takes 100ms (100 milliseconds, or one tenth of a second) to check a login, and 1ms (1 thousandth of a second) of that is hashing the supplied password. In reality it's actually a lot quicker to run the hash, but for the sake of example we'll use simple numbers.

We can increase the time it takes calculate the hash by simply re-hashing the output over and over, say 1,000 times. Here's some Python code that illustrates how this is done:

```
import hashlib

hash = hashlib.sha256(b"abc123" + b"password123").hexdigest()
for i in range(1000):
  hash = hashlib.sha256(b"abc123" +
         hash.encode('ascii')).hexdigest()

print(hash)
```

Your code should print:

```
019598128a03681d7c83bae0ea9db7ebfa1c63c1749bd08ebc7aa8e95e99dcf8
```

The output that's stored in the database is the final hash *01959*…. In our scenario, the time it takes to log in a user goes from 0.1 seconds to 1.099 seconds (or about an extra second). Again, in reality the actual durations will be different, but even in our case this is probably not that noticeable to a user, especially since they would probably only log in once per session, not on every page. But the effect on an attacker is much more dramatic. It will slow down brute force attempts from 10 per second to less than 1 per second, meaning it would take 10 times longer to guess the password of a user.

This introduction has given you an idea of how not to store passwords, and what pitfalls there could be to building your own password storage. Luckily, we don't have to do that as Django provides good (but not the best!) security by default. Let's look at how Django stores passwords and how you can improve on its default security.

Storing Passwords

How Django Stores Passwords

If you look in the Django database, you'll see how its password hashes are stored (not that these aren't visible in the Django admin). Here's an example:

```
pbkdf2_sha256$260000$ud2D0L3h8b98JDjGaffUnQ$LAYlrboOAUGZdRhEh7xT
/oom2Nv/dpJpmDOpnmPze0k=
```

The format of this string is:

```
<algorithm>$<iterations>$<salt>$<hash>
```

So in our case:

- algorithm (or "hash function"") is `pbkdf2_sha256`
- number of iterations is `260000`
- salt is `ud2D0L3h8b98JDjGaffUnQ`
- the actual hash is `ud2D0L3h8b98JDjGaffUnQ$LAYlrboOAUGZdRhEh7xT/oom2Nv/dpJpmDOpnmPze0k=`

The hash functions that Django can use are set using the `PASSWORD_HASHERS` setting, which defaults to:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
]
```

The way this list works is that the first element (`'django.contrib.auth.hashers.PBKDF2PasswordHasher'`) is the default algorithm that Django will use to hash passwords, but it is able to validate passwords hashed with the others in the list. If a password is stored in the database that isn't hashed with one of these, it won't be validated. This allows us to disable algorithms if they become insecure.

The algorithm that Django recommends is actually Argon2, which was the winner of the 2015 Password Hashing Competition. However it is not enabled by default as it requires installing a third party library.

Let's make Blango a bit more secure, by switching to Argon2 for password hashing. First install the Argon2 algorithm with `pip` in the terminal.

```
pip3 install django[argon2]
```

▼ **Installing [argon2]**

Depending on your shell, you might have to quote the name of the package:

```
pip3 install "django[argon2]"
```

Then the `PASSWORD_HASHERS` setting can be updated to make Argon2 the default:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
]
```

Since we leave the other algorithms in the list, we can still validate our password that was set with `PBKDF2PasswordHasher`.

Try updating your password in Django Admin. While you can't see the whole password you can see the hash it uses. You should see this update to argon2.

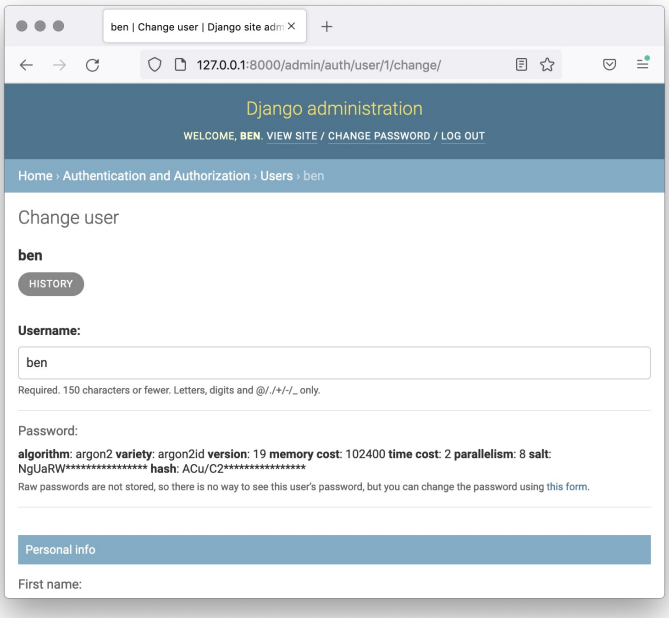

Figure 1

View Blog

If you want to make your passwords even more secure and harder to compute, check out the Password management in Django. It gives you some example of how to change the parameters for the algorithm you've chosen.

That's all we're going to cover for security in Django for now. In future courses if we use a specific filter or output method we might point out the security rationale behind it and the reason for doing it this specific way.

We're going to finish this module with a brief look at some deployment options for Django.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish security ad passwords"
```

- Push to GitHub:

```
git push
```