Edinburgh Napier University

SET09103 - Advanced Web Technologies

Topic 08 - Keeping Data Safe

Dr Simon Wells

1 Overview

GOAL: Our aim in this practical is to investigate some topics related to how we keep data safe in our dynamic web apps. In this topic, that means specifically the following:

1. Encryption
2. Password Hashing
3. Secure Logins

Again, the 'Hello World' web-app from the first topic will be the foundation for the majority of the work that we do in the rest of the trimester so it is important to be comfortable with it. If you're still unsure, try it again from the beginning and consider asking a demonstrator to explain things a little.

Don't just go through the practical once and then move on. Instead, try the following approach:

1. Go through the practical once exactly as presented.
2. Go through the lab at least once more so that you are used to everything involved. We will build on the knowledge from this lab and assume it in subsequent topics so get comfortable with it before moving on.

The practicals are built around the following basic structure:

1. An introduction to give you a high-level overview.
2. Some exercise activities that you should iterate through until **you** feel confident with the material covered.
3. Some challenges to give you a starting place for play and exploration.
4. Some additional resources & documentation to help you find more information and start filling in any gaps.

IMPORTANT: If something doesn't work, or you're not sure whether something has worked, then stop for a moment, think and consider what output you are aiming for rather than just plowing on. If you move ahead before understanding something then, if you ahve made a mis-step, you will be compounding error upon error, making it more difficult to extricate yourself. A good rule of thumb when coding is to *aim to always keep things working*. So if something breaks, or doesn't give you the result you want, you should unwind things back to a *known good state*.

2 Activities: Exercises

We've pretty much got everything that we need for most features in a modern dynamic web-app, however, so far, we haven't really considered the security of our deployed system. At the moment, anyone listening in on the network between your client and server could eavesdrop on private information sent between client and server. We should consider security and privacy overall.

One approach to security is to think of it as a property that comes as a result of a combination of appropriate authentication, authorisation, and auditing. Dealing with auditing first, this is basically keeping track of what happens when your web-app is running and is accounted for, on a basic level, by the logging topic we considered earlier. You should decide what information you want your logs to contain, that might help you to fix problems, or detect that someone is trying to gain access to your system. So things like Python errors are worth logging, because then you can detect errors at runtime so that you can fix them. You might also want to log failed log-in attempts, which might help you to detect that someone is trying to hack your site. Similarly, anything that falls outside the normal usage of the site, for example, logging 404 errors can be useful because these happen when someone tries to access a page that doesn't exist, and might be as a result of someone exploring the possible address space and looking for pages that give them access. Authentication and authorisation work together. Authentication let's you identify a user, for example, by answering a challenge with the correct credentials. Authorisation is about restricting access to specific resources or pages either to only specific users, or users with a specific role.

We won't deal with privacy in depth here, partly because it is a large, complex, and actively developing area. Also, for the most part, it is a less explicitly technical issue in many respects. Yes, we get privacy as a side-effect of good security in a strictly technical sense, for example, if a machine is deployed using secure HTTP (HTTPS or HSTS), then it makes it less easy for eavesdroppers to listen in. By also using an appropriate combination of authentication, authorisation, and auditing, we can secure routes and make them only available to users who are authenticated and authorised to access that route. What we really should consider though, as far as building privacy preserving systems, is what data the system collects, whether that data is needed, and how long that data should be kept. This is a data design issue in which you should interrogate your assumptions about what data you want to collect about your users. The rule of thumb that I work with is that if I don't have a positive and clear reason to collect some information, then I don't collect it. The basic idea is that if I don't have the information, then I can't misplace or lose it, which simplifies my life immensely and removes a lot of potential legal jeopardy. If you do need to collect any information about your user then you must be careful about how you store it, ideally encrypted so that even if someone gained access to your server then they couldn't use it. Once you no longer need the information then you should consider deleting it, because, again, if you don't have it you are at less risk from it. The worst thing you can do is to collect information about your users just in case you think it might be important later on. Then you might have information that is potentially legally problematic, that is not even benefitting you.

2.1 Getting Started

Log into your lab machine. For this practical you will need a web browser, a text editor, the windows terminal (cmd.exe), and the file manager open as you will probably use a mix of all of them at various points. You will also need uv installed as we covered in the previous lab as well as your working hello world Flask app as a starting point. Check that uv[1] works before proceeding.

2.2 Using PyCryptoDome Library for data encryption

Encryption is necessary to ensure that the data that we collect in our web-apps, for example, about our users, is stored safely. A good general approach is to not store any data about your users that you don't actually need for the purposes of your application, but if you do need to store data, especially data that might be described as personal information about individual users, then it is a good idea to enryptt it.

PyCryptoDome[2] is the Python Cryptography Toolkit and provides a range of useful tools for encrypting and working with encrypted data in Python. The following examples will mostly be run in the Python shell, but you can include the same commands into your Flask app as required.

PyCryptodome is a Python library that incorporates some useful cryptography tools. We can install it as follows:

```
$ uv pip install pycryptodome
$
```

We can now check that this library is installed properly by using the following command:

```
$ uv python -c "import Crypto"
$
```

If you get any output other than a new prompt then the library is not available (in which case you should contact the module leader for a fix). You can now import and use PyCryptodome within your web-apps, however let's first explore what the library can do in the Python REPL, so launch python in your shell, e.g.

```
$ uv python
```

[1] You might need to add it to your path again.
[2] https://www.pycryptodome.org/en/latest/

We can now look at some of the things that we can use PyCryptodome to do. First we shall look at Hashing, taking input and calculating a fixed-length output called a hash-value, then we will look at Encryption, taking input and a key and producing a cypher text. The main difference between the two in practise is that hash algorithms are designed, as a rule, to be one-way or *trapdoor* functions, whilst, so long as you know the key, encryption algorithms are designed to enable you to recover the input from the cyphertext.

2.3 Hashing

The idea with Hashing is to take as input a string and calculate a fixed length output string called the *hash value*. This is a useful way to quickly determine whether two strings are the same, for example if wanting to ensure that the string that was transmitted is the one that was received. In that case we would calculate the hash value associated with the string, then transmit both the string and the hash value to the remote recipient who then recalculates the hash value from the string then compares the two hash values. if they differ then the string was altered in transit and if they are the same the string that was recieved is the one that was sent. This kind of usage is called *file integrity checking* but is also known as *checksumming* or *calculating a checksum*.

Hashing relies on various assumptions:

1. It should be very difficult (if not impossible) to guess the input string based upon the output string
2. It should be very difficult (if not impossible) to find two different input strings that have the same output (known as a *collision*)
3. It should be very difficult (if not impossible) to modify the input string without also modifying the hash value as a result.

We can hash a value, e.g. a string, using a range of hash functions but we can start with SHA256. First we will hash the value passed directly to SHA256 then we will compare it to the same value stored in a string and passed in, and a different value:

```
>>> from Crypto.Hash import SHA256
>>> SHA256.new('Hello Napier'.encode('utf-8')).hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5694'
>>>
>>> hello_napier = 'Hello Napier'.encode('utf-8')
>>> SHA256.new(hello_napier).hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5694'
>>>
>>> SHA256.new('Goodbye Napier'.encode('utf-8')).hexdigest()
'ebccfcde2209e3dff4deff67fdbae71129ea87692e40295768de48317244347e'
>>>
```

2.4 Encryption with Block Cyphers

Block cyphers work on their input data in *blocks* of a fixed size, for example, blocks of 8 or 16 bytes in length. We'll use the Data Encryption Standard (DES)[3] as our example. The basic form of DES is pretty comprehensively broken as a secure encryption standard, however Triple-DES is still considered secure. There are many cryptographic algorithms available and for real-world uses it is worth investigating current best cryptographic practises and algorithms to adopt. DES works in various modes, e.g. Electronic CodeBook (ECB) mode or Cypher Feedback (CFB) mode amongst others. We will start with a demonstration of encrypting data using ECB.

```
>>> from Crypto.Cipher import DES
>>> des = DES.new(b'secret!!', DES.MODE_ECB)
>>> test = b'greeting'
>>> cipher_text = des.encrypt(test)
>>> cipher_text
b'(\xe5H\xe8\x10k\xc1['
>>>
>>> des.decrypt(cipher_text)
b'greeting'
>>>
```

[3] https://en.wikipedia.org/wiki/Data_Encryption_Standard

Notice that we first imported the DES part of the library. We then create a new DES instance, set the mode 'DES.MODE_ECB' and our encryption key '01234567'. We then used DES to encrypt our input string to yield a cipher text. After that we decryped our cipher text to recover the original data.

IMPORTANT A limitation of DES is that the key must be exactly 8 Bytes longs and the data that is encrypted must be a multiple of 8 Bytes in length. Obviously our data will seldom be a multiple of 8 Bytes so we often need to pad the data up to the next multiple of 8 Bytes.

Now let's try a different DES mode, DES CFB, to encrypt and descrypt some data. Of interest with CFB mode is that instead of each block being encrypted individually to form the cipher text each block in CFB mode is combined with the previously encrypted block. This time we shall use a slightly longer plain text to give a more interesting message, but still stick to the multiples of 8 Bytes rule. This is because we need more than one block for the combination step to work.

```
>>> from Crypto.Cipher import DES
>>> from Crypto import Random
>>> feedback_value = Random.get_random_bytes(8)
>>> des_enc = DES.new(b'secret!!', DES.MODE_CFB, feedback_value)
>>> txt = b"Hello World From Napier"
>>> cipher_txt = des_enc.encrypt(txt)
>>> cipher_txt
b"\xea\xa2\x81/\x9f\xc6\x80\xbf\xd8\xeef\x81\x89M\x8f'\xe7\x9eB\xa7\xb8\xd7\xd6"
>>> des_dec = DES.new(b'secret!!', DES.MODE_CFB, feedback_value)
>>> des_dec.decrypt(cipher_txt)
b'Hello World From Napier'
```

Notice that this is fairly similar to the ECB version earlier. The are two main points to notice. The first is that the DES object now takes a third argument, an 8 Byte random string and we use the Random function of the Crypto package to supply us with random data[4]. The main difference is that we have created two DES objects this time, one to encrypt and one to decrypt. This is necessary because of the combination step, that occurs after each block is encrypted which alters the feedback value.

2.5 Encryption with Stream Cyphers

Stream cyphers differ from Block cyphers by working on byte-by-byte on their input data, treating the data as a stream instead of discrete blocks. Stream cyphers are essentially block cyphers in which the block size is 1 Byte in length. PyCryptodome only supports two stream cyphers, ARC4 and XOR, in ECB mode. Let's look at an example of using the ARC4 algorithm:

```
>>> from Crypto.Cipher import ARC4
>>> arc4_enc = ARC4.new(b'01234567')
>>> arc4_dec = ARC4.new(b'01234567')
>>> txt = b"Hello World from Edinburgh Napier University"
>>> cyphertxt = arc4_enc.encrypt(txt)
>>> cyphertxt
"\xd9\xb0\x9fs)\x07r_B?\xdfz\x94\xdc\x8c='\x8b.9@\x1e\xe3@\xc68K\x1c\x18:\xad\xc3:~\xf9\
    x95wa\xbcB\xa0U\x08\xe9"
>>> arc4_dec.decrypt(cyphertxt)
'Hello World from Edinburgh Napier University'
```

2.6 Public Key Encryption

Correctly selected and used instances of stream and block cyphers are acceptably secure and performant. However they have a fundamental flaw, in certain contexts, that is common to all security systems. People. These cyphers require both the people encrypting and the people decrypting to share the same key. Key management is a big problem, if not the major problem, that makes things like email encryption difficult for the average computer user. As a result, the average email might as well be written on a postcard.

[4]It is worth noting that often the quality of encryption depends upon the quality of the randomness that is supplied to the algorithm. It is actually quite hard to get truly random numbers from a computer and even small statistical regularities in the randomness can be sufficient to break the encryption

Public-key encryption uses two keys, known as a key-pair to encrypt and decrypt data. The two keys in a key pair are created together using an algorithm than ensure that they are mathermatically related to each other. One key is designated the public key and the other key is designated the private key. The public key can be shared with anyone whereas the private key is kept private and known only to the person who owns it. The public key is then used to encrypt some data and the private key is used to decrypt it. Because of the mathematical relation between the public and private key, data that is encrypted with one key cannot be decrypted with the same key but most be decryped using the key's partner. One way to think about this is like using a padlock and key. You give some one a box and padlock. They put a message in the box, then they lock the box with your padlock and only you can unlock it with your key. Things are actually more complicated than this in practise but this is a good place to start from. Keys are actually very long numbers, very long prime numbers to be specific and generally, the longer the key the more secure the key. PyCryptodome provides functions for generating keys, e.g.

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(2048)
>>> private_key = key.export_key()
>>> file_out = open("private.pem", "wb")
>>> file_out.write(private_key)
1674
>>> file_out.close()
>>>
>>>
>>> public_key = key.publickey().export_key()
>>> file_out = open("receiver.pem", "wb")
>>> file_out.write(public_key)
450
>>> file_out.close()
```

If you look in the folder that you ran the previous Python code in then you should notice a pair of new files. Theses are the public and private keys we just created. Having created a key-pair we can now use the public key to encrypt some data and then the private key to decrypt it. Let's start with using the public key to encrypt something, e.g.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Random import get_random_bytes
>>> from Crypto.Cipher import AES, PKCS1_OAEP
>>> data = "Soylent Green is people".encode("utf-8")
>>> file_out = open("encrypted_data.bin", "wb")
>>> recipient_key = RSA.import_key(open("receiver.pem").read())
>>> session_key = get_random_bytes(16)
>>> cipher_rsa = PKCS1_OAEP.new(recipient_key)
>>> enc_session_key = cipher_rsa.encrypt(session_key)
>>> cipher_aes = AES.new(session_key, AES.MODE_EAX)
>>> ciphertext, tag = cipher_aes.encrypt_and_digest(data)
>>> [ file_out.write(x) for x in (enc_session_key, cipher_aes.nonce, tag, ciphertext) ]
[256, 16, 16, 37]
>>> file_out.close()
```

Having encrypted some data with the public key (*receiver.pem*), we can now go ahead a decrypt the file we just created *encrypted_data.bin* using our private key (*private.pem*).

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import AES, PKCS1_OAEP
>>> file_in = open("encrypted_data.bin", "rb")
>>> private_key = RSA.import_key(open("private.pem").read())
>>> enc_session_key, nonce, tag, ciphertext =
    [ file_in.read(x) for x in (private_key.size_in_bytes(), 16, 16, -1) ]
>>> cipher_rsa = PKCS1_OAEP.new(private_key)
>>> session_key = cipher_rsa.decrypt(enc_session_key)
>>> cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
>>> data = cipher_aes.decrypt_and_verify(ciphertext, tag)
>>> print(data.decode("utf-8"))
Soylent Green is people
```

A couple of other useful things that we can do with the RSA algorithm are to sign and verify messages. We can sign a message using a public key and a hash algorithm in order to establish two things (1) that the message hasn't changed during transmission, and (2) that the origin, meaning the person who sent the message, can be trusted. Notice that this only holds if the key pair has not been compromised. This is the main reason why effective secure communications is difficult, because managing and sharing keys on a large scale is difficult, even for experienced computer users, and small mistakes can easily leave the entire system nearly as insecure as it would be without encryption. In fact some might say that a system that uses insecure encryption is worse,

because you might believe it to be secure whereas within an un-encrypted system you already know not to trust it.

So let's sign a message, first we import a few libraries. Then we prepare a message and load in an existing key (*receiver.pem*). We then calculate a hash-value for the message and sign it using our RSA key, e.g.

```
>>> from Crypto.Signature import pkcs1_15
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>>
>>> message = b'My namie is Inigo Montoya. You killed my father. Prepare to die.'
>>> key = RSA.import_key(open('private.pem').read())
>>> h = SHA256.new(message)
>>> signature = pkcs1_15.new(key).sign(h)
```

Given the plain text, the signature, and the public key (but *not* the private key) the recipient of a message can now check that the message that was sent both came from the person who owns the private key but also that the messages hasn't been maliciously altered during transmission, e.g.

```
>>> key = RSA.import_key(open('receiver.pem').read())
>>> h = SHA256.new(message)
>>> try:
...     pkcs1_15.new(key).verify(h, signature)
...     print("The signature is valid.")
... except(ValueError, TypeError):
...     print ("The signature is not valid.")
...
The signature is valid.
```

Public Key cryptography is an important tool in the fight to ensure that we can communicate in ways that are secure from eavesdropping, non-repudiable (meaning the person who sent it can be verified as such), and unaltered.

2.7 Using py-bcrypt Library for Password Hashing

Hash functions can be used in password management and storage. Web sites should only store the hash of a password and not the raw password itself. This way only the user knows the real password. When the user logs in, the hash of the password input is generated and compared to the hash value stored in the database. If it matches, the user is granted access[5]. Whilst you could use the hashes available in PyCryptodome for user passwords, this is no longer the most secure approach. Rather you should use a type of password hash that has been specifically designed for use to store password hash-values and thus mitigates many of the drawbacks of generic cyrptographic hash functions in this context. Such hashes are known as *key derivation functions* and are designed to significantly slow down the process of hashing a value so that brute force attacks, where you try to calculate all possible hashes, become significantly expensive in terms of time to calculate. This contrasts with hashes used for checksumming, which obviously need to run as fast as possible.

BCrypt is the OpenBSD Blowfish password hashing algorithm that is described in a paper by Niels Provos and David Mazieres called "A Future-Adaptable Password Scheme"[6]. There is a Python wrapper for this algorithm called py-bcrypt[7] that you should use to hash password in your web-apps, at least until this algorithm is demonstrated to be flawed or a stronger system is proposed.

The BCrypt library isn't yet installed so you'll need to install it as follows:

```
$ pip3 install --user bcrypt
$
```

[5] If you ever use a website that can help you to *recover* your password then they probably aren't hashing passwords. In this case you should probably question the security of that site and their ability to keep your data safe.

[6] http://www.openbsd.org/papers/bcrypt-paper.ps

[7] http://www.mindrot.org/projects/py-bcrypt/

Now check that it's installed properly with the following command. Again, if you get any output other than the return of the prompt '$' then you should contact the module leader for a fix:

```
$ python3 -c "import bcrypt"
$
```

As well as implementing a password derivation algorithm, bcrypt can also salt the supplied password so that the hashed-value is more resistant to rainbow-table based attacks. When you use the hashpw function, you can either pass in the pre-generated salt or else call gensalt() directly and the salt is stored within the generated hash-value. To hash a password we do the following:

```
>>> import bcrypt
>>> hash = bcrypt.hashpw('secretpassword'.encode('utf-8'), bcrypt.gensalt())
>>> hash
b'$2b$12$5KkwUyRUDEooMWAMBOLdnuiOIJPvG2dYmP5nNyhSs1hW1xdovn0Ni'
```

First we import bcrypt, then we create a new hashed value using the supplied password and asking bcrypt to generate a salt. We only need to store the hash-value after that as the salt is stored within the hash-value. For example, you could now store the hash-value in your user database. When we need to verify a user's password we would do the following, assuming that you have retrieved the hash-value from your user DB as 'hash' and that the supplied password is stored in 'pass':

```
>>> hash == bcrypt.hashpw('secretpassword'.encode('utf-8'), hash)
True
```

and if the supplied password is incorrect we will see something like this:

```
>>> hash == bcrypt.hashpw('badpassword'.encode('utf-8'), hash)
False
```

This should be enough to get you started in storing your user's data securely. The rule of thumb is to keep your security arrangements as simple as possible because unwarranted complexity can hide weaknesses. You should also not implment your own security arrangements if there are already well tested alternatives. Encryption is a game for specialists and the liklihood is that you will not crate anything as good as what is already available.

3 Secure login with BCrypt & Flask

Now let's combine our use of bcrypt for password hashing with some Flask decorators and Python functions so that we can protect specified routes from being accessed by users who have not supplied the correct credentials. We'll start by breaking down what we need in our Flask app. Firstly we need some routes, a default '/' route, a '/secret/' route, and a '/logout/' route. Secret is the route that we will protect from access by unauthorised users. The default route will present a login form then check the login credentials and will either reshow the login form, if authorisation fails, or else redirect to the secret route if our login is successful. Finally our logout route is used to reset the authorisation within our app as otherwise we would have to delete our cookies each time we wanted to log out.

We'll start by putting together our simple login form, which should be stored in a *templates* folder under our main python app file:

```
1  <html>
2  <head>
3  </head>
4  <body>
5
6  <form name="login_form" action="" method="post">
7
8      <input type="email" placeholder="Email" name="email">
9      <input type="password" placeholder="Password" name="password">
10     <button type="submit" class="btn" name="button" value="login">
11         Sign In
12     </button>
```

```
13
14 </form>
15
16 </body>
17 <html>
```

As you can see this merely present two input boxes, one each for the email and password respectively, and a button. When the button is pressed the form is POSTed to the route that is listening for it, in our case the *root* route.

Now let's see the flask app itself which is stored in *login.py*

```
1  import bcrypt
2  from functools import wraps
3  from flask import Flask, redirect, render_template, request, session, url_for
4
5  app = Flask(__name__)
6  app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
7
8  valid_email = 'person@napier.ac.uk'
9  valid_pwhash = bcrypt.hashpw('secretpass'.encode('utf-8'), bcrypt.gensalt())
10
11 def check_auth(email, password):
12     if(email == valid_email and
13         valid_pwhash == bcrypt.hashpw(password.encode('utf-8'), valid_pwhash)):
14             return True
15     return False
16
17 def requires_login(f):
18     @wraps(f)
19     def decorated(*args, **kwargs):
20         status = session.get('logged_in', False)
21         if not status:
22             return redirect(url_for('.root'))
23         return f(*args, **kwargs)
24     return decorated
25
26 @app.route('/logout/')
27 def logout():
28     session['logged_in'] = False
29     return redirect(url_for('.root'))
30
31 @app.route("/secret/")
32 @requires_login
33 def secret():
34     return "Secret Page"
35
36 @app.route("/", methods=['GET', 'POST'])
37 def root():
38     if request.method == 'POST':
39         user = request.form['email']
40         pw = request.form['password']
41
42         if check_auth(request.form['email'], request.form['password']):
43             session['logged_in'] = True
44             return redirect(url_for('.secret'))
45     return render_template('login.html')
```

Because we are keeping this login example as simple as possible, you will notice that the user credentials are hard-coded into the example using the *valid_email* and *valid_pwhash* variables. Notice also that valid_pwhash doesn't store the password but the hashed and salted version of the password. This means that we cannot easily recover the user's password if they forget it. In a real app we would store these credentials in a user database then retrieve them as required, however we have not included that here in order to simplify the example.

We also have some utility functions, for example the check_auth function that takes in an email and password, then compares them to the stored user and hashed password, returning True or False depending upon whether they match or not.

Next we have a decorator function called 'requires_login' that we use to *decorate* any Flask route that can only be viewed by a logged in user. If the user is not logged in then they are redirected to the '/' route. The only check that we are doing within this decorator is seeing whether there is a 'logged_in' session variable. If this check evaluates to True then the user is considered to be logged in, and not otherwise.

The 'logout' route merely removes the session variable that determines whether a user is logged in or not then redirects to the '/' route.

The 'secret' route just displays a string indicating that it is secret. Notice the additional decorator on this route. Not only do we have the '@app.route' decorator that tells Flask to treat this function as a web-app route, but we also have the '@requires_login' route which causes the 'requires_login' function to be run *before* the 'secret' route is executed.

Finally, the 'root' route displays our login template. If a POST is received then the check_auth route is called with the supplied credentials to determine whether the user is logged in.

This is a relatively low-security way to implement a login mechanism for a Flask app. It is susceptible to various attacks, for example, a replay-attack; if an attacker were to copy a logged in user's cookie into their own browser then they would be treated as logged in, even though they had made no changes to the cookie. They wouldn't however be able to log in again as they would not have valid credentials to do so, so this attack can be mitigated to some degree by setting a timeout on the length of the user's authentication period. Another defence is to use encrypted tokens that increment for each request a user makes so that there is a strict sequence of interactions between the user and the app. If any interaction supplies an incorrect (or reused) token then the user is automatically logged out and asked to supply their credentials. Achieving a secure login that is not in any way vulnerable to attacks is a difficult task but as a developer and designer we should also evaluate the likelihood of various attacks, for example, the replay attack described above would require an attacker to gain access to your users secure cookie stored in their browser which would mean that the user is likely already heavily compromised.

However all is not lost. If we securely store our user's data and hash passwords and other sensitive data, and if we also follow a privacy-by-design methodology and only store data that is necessary for the effective functionality of our app, we can at least reduce the effects of a successful attack in terms of our user's personal data. If we are also active in logging the behaviour of our app, checking those logs, and reviewing our codebase for known vulnerabilities or attack vectors then we will also be reducing the likelihood of a successful attack occuring.

4 Activities: Challenges

Taking any of the Flask apps that you've created so far, including those you made during earlier labs, as a starting point, or else creating something from scratch:

1. Extend the secure login example so that it uses a simple database to store accounts for individual users rather than hardcoded email and password credentials.
2. Extend the last challenge so that you have two user roles, a regular user and an admin user. Now implement an admin route so that only a user in the admin role can access the admin route.
3. Extend the last challenge so that the secret page that regular users see is personalised and greets the logged in user.
4. Extend the last challenge so that the admin page can be used to administrate the user accounts for the site. This might include listing signed up users and being able to delete them.
5. Extend the last challenge so that users can set a new password whilst they are logged in.
6. Extend the last challenge so that users can start a new password process if they forget their credentials.

If you're still not confident with creating a new Flask web-app from scratch[8] don't just edit and re-use your existing ones. The setting up aspect is still part of the practise. For each of the exercises, create a new project folder in your set09103 folder. You will need to create a new virtual env each time and install pip. Each project folder should have a unique name but don't call it app or flask as this will cause errors. Doing this will give you practise in setting up new Flask projects so that it becomes straightforward. We'll be doing that a lot over the coming weeks so making it less of a chore is a good idea. One tactic that can help is to create yourself a crib sheet containing the *invariant* commands that you use each time you set up a new Flask app for development. Over a short period of time, and a few repetitions of practise, you **will** get past this.

If you are confident in creating a new Flask app as needed then it is fine to reuse your existing web-apps as a skeleton for new activities and challenges. Note also that once your have set up a virtualenv it can be reused for different Flask apps, you just need to pass the name of the flask app python file to the "*uv run ...*" invocation to tell uv that you want to run a different flask app.

5 Finally...

If you've forgotten what you can do with HTML and want a little more practise or to look up the details of some tags then see the following:

- HTML Tutorial: `https://www.w3schools.com/html/`
- HTML Exercises: `https://www.w3schools.com/html/html_exercises.asp`
- HTML Element Reference: `https://www.w3schools.com/tags/`

Similarly, if you aren't yet entirely comfortable with Python then see the following:

- Python Tutorial: `https://www.w3schools.com/python/default.asp`
- Python Language Reference: `https://www.w3schools.com/python/python_reference.asp`

Use the following to find out more about HTTP verbs and status codes:

- HTTP Verbs: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods`
- HTTP Status Codes: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status`

Remember that the best way to become really familiar with a technology is to just use it. The more you use it the easier it gets. Building lots of "throw away" experimental apps, scripts, and sites is a good way to get to achieve this so let your imagination have free reign.

[8] you should be able to do the entire pprocess from the first lab topic in less than five minutes by now.