An introduction to authentication
Users, passwords, and hash functions; with flask.

Alessandro Tomasi*[1]

[1]Security and Trust, Fondazione Bruno Kessler

October 20, 2018

Abstract

Installing flask and running your first web app; logging in users; working with hash functions.

1 Introduction

In this workshop you will be asked to create a web app providing login functionality to registered users. A set of exercises is provided in Section 3 to focus your work and sharpen your understanding of the topic. Feel free to browse ahead and decide for yourself if you can complete them without guidance; otherwise, the rest of this document is designed as an introduction to the main topics and a quickstart guide.

In particular, there are many topics you will encounter along the way that are not the focus of this course, such as jinja html templates and static resources like css files and logos. These are quite useful to build a functional app and are intended for you to practice writing an app in flask, but please resist the temptation to spend too much time on anything that is not strictly related to security issues. Your goal is to provide login functionality and practice with hash functions.

Finally, this document is a selection of topics designed to guide you to the material specific to this course, and leave out a great deal of information that is not strictly necessary; however, if you think something important is missing, please do let your workshop tutor know.

1.1 IP address and TCP ports

Your device has an address on the network you are currently on, almost surely an IP (Internet Protocol) address of 32 bits, commonly written as four bytes (in decimal notation) separated by a full stop. For example:

*altomasi@fbk.eu

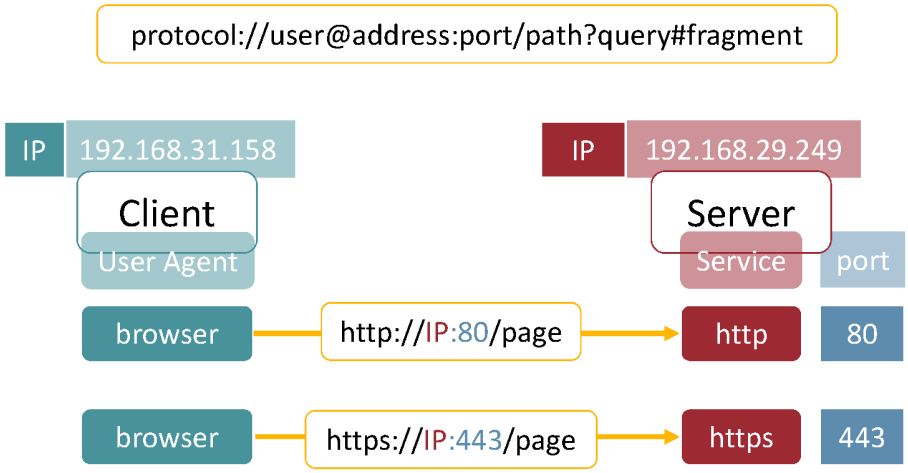

Figure 1: URL structure and examples.

`127.0.0.1` This is called the `loopback` or `localhost` address. Any request sent to this address will be re-routed to the local machine.

`192.168.1.1` This is usually your home router address.

IP addresses for devices that come and go from a network are usually assigned dynamically. You can determine your machine's current IP address from a terminal using `ifconfig` on linux or `ipconfig` on windows. You can also do this on your mobile phone if it is for instance connected to the local wifi; you can try a terminal emulator or use a full network scanner like fing.

In addition to having a network address, services running on devices listen on specific ports; for instance, http traffic is usually addressed to port 80, or sometimes alternatively port 8000; https uses port 443; and ssh uses port 22. There are TCP UDP port lists if you want to see which service makes use of which port.

You are free to use a different port, as long as there is a service listening on it. The full range of available ports is 2^{16} (0 to 65535); the first 2^{10} are reserved for privileged or well-known services, and it is generally good practice to choose a port above 5000 for unconventional purposes.

If you are using virtualbox to run a linux virtual machine, you can use port forwarding to re-route requests from your host machine to your guest machine. For instance, forwarding `127.0.0.1` port `5002` to guest port `5001` will allow you to use your host browser with address `localhost:5002` to access whatever service is listening on port `5001` on your guest.

You can find a summary representation of URLs and examples for specific protocols in Figure 1.

1.2 html forms and POST

GET and POST are http methods and can be used in html forms. GET indicates a request for a resource; POST indicates a request to process the body of a request, and is generally most appropriate for sending completed forms.

An html form allows users to enter data in a structured container - for instance, their username and password in a text field. There are many good tutorials online on how to write html forms, for instance this one. For the purposes of this course, you are unlikely to need more than the most basic elements; for instance, if you understand the following you should be good to go:

```
<form method="POST">
    <label for="AIRSPEED">What is the airspeed
                of an unladen swallow?</label>
    <input type="TEXT" id="AIRSPEED"
            name="ANSWER" value="AFRICAN OR EUROPEAN?" />
   <input type="SUBMIT" value="WHAT DO YOU MEAN?">
</form>
```

`label` is a string displayed on screen for the user to understand what they are expected to enter in a specific `input`; `for` links a label to an `id`; `value` is the default text for that input.

2 Set-up

I assume you will be using linux mint, ubuntu, or an equivalent distribution with apt package manager and repositories.

2.1 flask: hello world

There are extensive tutorials on flask - see the official documentation, in particular the quickstart guide.

Install flask, either using apt or pip:

```
sudo apt-get install python-flask
```

or the python3 version, if you prefer.

Test your installlation with the default hello world app: adapt the default code

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'HELLO, WORLD!'

if __name__ == '__MAIN__':
```

```
    app.run(    )
```

and run it with

```
python app.py
```

Setting the environment to development will allow flask to automatically reload your app code every time the source files change, and produce debug output in the browser.

2.2 HTML templates and jinja

Create a folder called `templates`. Your html files go here. For instance:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():

    return render_template('HOME.HTML')
```

You can write re-usable html templates using Jinja if you so wish.

2.3 Static resources and `url_for`

Create a folder called `static`. All resources that do not change, such as css files or images, go here and are referred to using `url_for`. For instance, using templates:

```
from flask import Flask, render_template, url_for

app = Flask(__name__)

@app.route('/')
def home():

    css = url_for('STATIC', filename='STYLE.CSS')
    return render_template('HOME.HTML', css_url=css)
```

3 Exercises

3.1 Practice with flask

1. Ensure you are running a current LTS linux distribution, e.g. mint or ubuntu.

2. Install flask.

3. Create a web app listening on localhost port 5001.

4. Create two pages for the app serving html from a template and using the same css static resource. The first page (login) should let the user specify a string, e.g. a username, and the second page (welcome) should display that name.

5. Write a web app handling user login and storing their hashed salted passwords. In particular, it should include:

   - a registration form,
   - a login form,
   - a route displaying different content depending on whether the user is logged in or not

You will need to store user data locally. You may handle this as you prefer; if you use a text file, please make it tab separated to be readable.

You may choose to use the flask-login extension, or not; you can handle the session object and cookies directly without the login extension (see e.g. here), it's just more work.

When generating the salt, and other random quantities for the purpose of security, it is important to use a cryptographically strong random number generator. In python you may wish to use os.urandom(). If you do, note that this will generate binary data (bytes); when displaying this or saving it to disk, please make it readable with some encoding intelligible to humans, such as base64 or hex. You may wish to use the binascii module, or not. If you do, note that finalized digests have a *newline* symbol appended; you can remove this using `rstrip()` (in python 2.7), or otherwise.

3.2 Practice with passwords and hash functions

You may use the cryptography or hashlib libraries.

1. Check that hash function digests are highly sensitive to changes in the message: if m changes only a little, the digest should be completely different.

   (a) Take an arbitrary string m as message, and compute $h(m)$ using SHA-256. You may randomly generate m.

   (b) Change the last character in m, and recompute the hash.

2. Let's say passwords can be written using letters ($\#L = 2 \cdot 26$), digits ($\#D = 10$), or special characters ($\#S = 32$). Assume passwords can be 6, 7, or 8 characters long. Calculate by hand the number of all possible passwords for the policies:

(a) No restriction; uniformly random

(b) Only letters

(c) At least one digit or special character

3. Assume each password is ASCII-encoded. What is the storage requirement for a table of all possible passwords and their corresponding hash?

4. Now consider adding a 16-bit random salt. What is the new storage requirement?

5. Verify that salting is not enough if passwords are too short, e.g. using hashcat and performing a mask attack.

6. A GeForce GTX 1080 Ti has a reported hashrate of 31.3 MH/s [source]. Since these are measured as double sha-256 for mining purposes, let's count it as $62.6 \cdot 10^6$ sha-256 operations per second. Assume that passwords are uniformly random in the available space for simplicity, and assume that an attacker has to try on average half of all possible passwords to find a match with a hash. Now calculate how much time it takes for an attacker to find the password in each of the three policies above.

7. A GeForce GTX 1080 Ti consumes 180 W of power. Assume an average electricity retail price of 0.08€ per kW/h.

   (a) Calculate how much money it costs to recover a single password on average for each of the three policies. Consider the cost of the card (about 800€), but ignore any other equipment and cooling costs.

   (b) Calculate how much it costs to compute every possible password for each policy.

   (c) Now consider adding a 16-bit random salt. How does the cost change?

   (d) Now consider using a double hash for each password, i.e. computing the hash twice. How does the cost change?

8. Now, go to haveibeenpwned.com and check when and how many times your hashed and salted passwords have been stolen from databases; make a note of how long ago that happened.

9. Hash functions as spam filter (see hashcash).

   Suppose you receive an email with content message m at your UniTN address a, and consider the following spam filter: if the message is preceded by a header with a 256-bit nonce n such that the first b bits of $h(a||n||m)$ are equal to zero, you read the message; otherwise it is considered spam.

   Using an arbitrary message m and an email address a, try to find an n such that the filter is satisfied for $b = 4, 8, 12, 16$ etc. How long does it take?

10. *Advanced: collision attacks* are specific to the hash function. You can find an example of how to exploit vulnerabilities in MD5 to create two different executables with identical hash here.

11. *Advanced: length extension attacks* also affect specific hash functions and can be dangerous if they are used as MACs. You may choose to attempt to replicate them, following instructions here and perhaps using a tool found here.