

INF-2310 - Computer Security

Assignment 4: Web Authentication

By Isak Kjerstad

ikj023@post.wit.no

1 Introduction

In this assignment we will implement a web authentication system where human users can register and log in to their corresponding accounts [1]. The system must fulfill the following requirements:

1. Authenticated users must be greeted with their username (and possibly some other hidden information) on the sites landing page [1], if and only if they are currently logged in to their own account.
2. New users should be able to register with a username and password combination. The password should be confirmed by using two input fields [1].
3. Existing users should be able to log in by using their username and password credentials. A successful login should redirect the user to the sites landing page. All input should originate from HTML forms [1].
4. User credentials must be persistently stored on disk, and should be resilient to dictionary and (fast) brute-force attacks [1].
5. The credentials must be protected from network snooping and replay attacks [1] during the client-server communication process.
6. Appropriate error messages should be displayed to the users [1], e.g. when log in fails or when new users attempt to register with weak passwords.
7. The system should be deployed on a server [1].

In short, the system should authenticate users and protect against eavesdropping, man-in-the-middle and replay attacks. Under normal operation only authenticated users should have access to their account. In case of a successful attack, leaking user information from the servers disk storage, no password should be easily readable [1]. This is to prevent attackers from compromising several services belonging to the average security-unconcerned user re-using the exact same password [2] on multiple sites.

2 Technical background

In order to communicate securely in-between the client and the server, the HTTP is insufficient to use. This is because all information sent over the HTTP is un-encrypted and simply sent in plain text [2]. The consequences of using the HTTP when exchanging credentials, no matter how the server behaves and stores data, is that a man-in-the-middle attacker can read and misuse all transmitted information. If the authentication system relies on a username and password, both these fields would be visible and even modifiable to an attacker, compromising both the confidentiality and integrity of the service and user. The HTTPS uses a symmetric cryptosystem in combination with MAC's [2] to obtain both confidentiality and integrity when sending and receiving requests and responses over any network. In the process of establishing a HTTPS connection (using TLS) [2], the server must send its SSL certificate [2] containing information like for example:

- The servers public key.
- Domain name to the server.
- Name of the certificate authority (CA) that issued the certificate.
- A unique serial number set by the CA.
- A digital signature of all the fields in the certificate.

The certificate enables the client to validate the servers authenticity by a chain of trust [2]. This is done by reaching out to a higher authority and check if the certificate is valid. The highest authority issues root certificates set to be trusted by the device/browser belonging to the client [2]. Of course, validating the expiration date, the digital signature and so on also goes into the process of verifying a certificate [2].

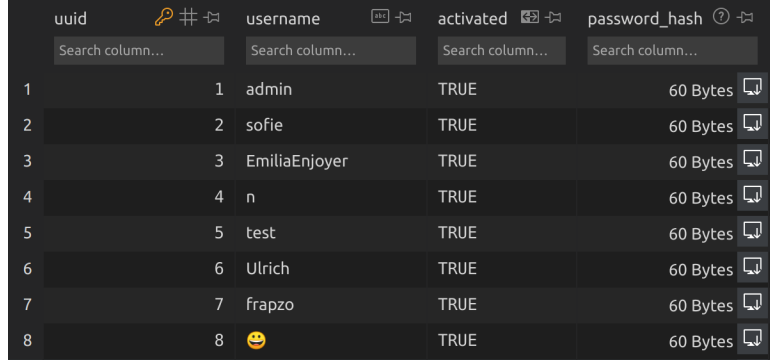
3 Design and implementation

By using Certbot to get a Let's Encrypt certificate [4] on the deployment server we immediately solve problem five mentioned in the introduction. The server is set to handle incoming HTTPS requests with NGINX [5], and then proxy the traffic to the Flask server. The certificate is linked to NGINX, so all incoming and outgoing web traffic can be encrypted as discussed in the technical background section. This solves both the concern with network snooping and replay attacks.

We use Flask [5] to create views for the landing, log in and registration page. The Jinja template language [5] is used throughout the project. A base template is utilized for a consistent styling on the entire site. The base template is simply a HTML file with a linked site-wide CSS file and a navigation bar. The base template also implements the flashed message feature in Flask, making it very easy to display information, warnings and errors to the users from the back-end

without much effort later on.

The server is set up to store user credentials and all other instance data in a SQLite database [6]. Flask’s SQLAlchemy [6] is chosen to create the models using the built-in Object Relational Mapper (ORM) [6]. The same mechanisms are later used to create and query users, and modify other data. The image below describes the user table and the storage of the needed credentials:



uuid	username	activated	password_hash
1	admin	TRUE	60 Bytes
2	sofie	TRUE	60 Bytes
3	EmiliaEnjoyer	TRUE	60 Bytes
4	n	TRUE	60 Bytes
5	test	TRUE	60 Bytes
6	Ulrich	TRUE	60 Bytes
7	frapzo	TRUE	60 Bytes
8	😊	TRUE	60 Bytes

Figure 1: *Instance of the user table.*

As seen on the image each user has a unique username and a password hash. The other fields are not very relevant or important, and can be ignored for simplicity reasons. When a new user is created, the plain text password is hashed using Bcrypt [3], and only the hash is ever stored in the database. Since Py-Cryptodome’s Bcrypt function has a maximum password length of 72 bytes, the recommended pre-hash step is taken [3] to allow a password of any arbitrary length.



Figure 2: *One of the Bcrypt password hashes*

Even though Bcrypt is a one-way cryptographic hash function [3] that salts the input to prevent rainbow table attacks [2], we still have a security challenge. A short password is easy to brute-force and common passwords can be vulnerable to dictionary attacks [2]. To fix this we must enforce a few restrictions on users chosen passwords. We allow all Unicode characters and require the password to be at least eight characters long. In addition, we restrict users from using the top 100 000 common compromised passwords. No other restrictions are enforced, in order to keep the key-space as large as possible [2]. Encouraging users to create strong passwords is hard. Too many restrictions, the key-space shrinks and users choose dangerous shortcuts. Too few restrictions, users create bad passwords as well [2]. However, this system prohibits the most common mistakes and allows for good passwords of any length.

When logging in, a query is done on the provided username. Then the inputted

password is hashed and compared to the stored hash. On a match, the user is logged in. This is done by setting a Flask session cookie [5]. The cookie is stored with the client, and is not confidential. However, the cookie is signed with the servers secret key [5]. Without access to this key, the client can view any cookie in plain text but not modify any of them. This preserves the needed integrity, so malicious users can not falsely gain access to other accounts. On a successful login, the user is redirected to the landing page.

The landing page has logic to slightly change behaviour based on if a user is logged in or not. Exactly what happens can easier be read in the source code, or observed on a running instance than described with words here in the report. However, the mechanism for verifying a logged in user is based around reading the session cookie.

4 Discussion and conclusion

All requirements are implemented and working. The server manages credentials in a way that follows good security practices. The authentication system accepts both user registration and the ability to log in. An improvement might be to hash the password on the client-side. At the moment the passwords might be compromised if an attacker gets physical access [2] to the server, and is able to read the contents of the main memory. The same goes if an attacker installs listening software in-between the NGINX server and the Flask server, or else is able to capture and log un-encrypted traffic within the server itself. This could be avoided if the server never has knowledge about the actual password at all. However, since all of the requirements are fulfilled, we can conclude this assignment as completed. The web authentication system is temporary deployed on the course server with a few additional and interesting features to be discovered.

References

- [1] Precept, `Web-Authentication/*`
- [2] Michael Goodrich, Roberto Tamassia *Introduction to Computer Security*. 1. Edition by Pearson
- [3] PyCryptodome, <https://pycryptodome.readthedocs.io/en/latest/>
- [4] Certbot, <https://certbot.eff.org/>
- [5] Flask, <https://flask.palletsprojects.com/en/2.2.x/>
- [6] SQLAlchemy, <https://flask-sqlalchemy.palletsprojects.com/en/3.0.x/>