Overview

Our solution is a video sharing platform with login verification security. That security is the main focus of our project. When you are online you are effectively anonymous. While it is true that you can have a social media account, there are very few ways for someone to verify that every post made by you is truly you. Every few weeks there are stories of politicians getting hacked and posting unprofessional messages to the public. For this issue, we have developed a system of multi factor authentication (MFA) to ensure user security so that you will not be impersonated online. Currently, our solution is two processes, one is a login and encrypted database while the other is a multi factor authentication system. Our end goal is to integrate the two so that the login can require a MFA.

Password verification

We have created a two tier security system to allow you to browse videos freely when you are logged into your account but you are required to sign in through MFA to be able to post videos of your own. We devised this solution so that the most potentially damaging actions a person could take are kept behind tight security while the most common and low stakes usages of the product are convenient to access.

For this, our website has two login steps. MFA authorization and a standard sign in. When a standard sign in is performed, a new window pops up prompting you to enter your password and email. From there, our application checks its RDS database to see if the identification info is correct. Due to our database utilizing set semantics, it is impossible for two users to have the same username. In the event that it is correct we log them in at the basic level. Should the user wish to log in at the more advanced level, they would be required to sign in through MFA. Upon doing so, they are asked to enter a code which was sent to the user through their email. This system works to increase user security as any hacker would need access to both your account and your authentication tool.

Encrypting data

To ensure the security and privacy of users, our program utilizes both client side encryption and server side encryption to hide user data in the event of a data breach. The server side encryption is handled by Amazon Web Services(AWS), a service that is trusted by thousands of organizations to handle their data while the client side encryption is performed by us through a Blowfish Encryption algorithm(BEA). This serves is to our benefit as it ensures that even if us or AWS suffer a data breach the hackers will only receive encrypted data. We chose to use BEA because it is the algorithm that is more resilient to brute force attacks, and most likely to stand up to Moore's law as the years go on. As an example, a modern server could crack an MD5

hash for a 6 character long password in under 40 seconds. Compare this to BEA where that same server would take almost 12 years. This helps to ensure that even if data is intercepted or stolen it is functionally useless. This system we have created helps to protect user privacy and limit the effectiveness of malicious attacks against our system such as SQL injections. For example, a SQL query such as “SELECT * FROM Users WHERE UserId = 105 OR 1=1;” would print out all tuples in a table where 1=1 which is always true but our code has been tested and verified to bypass this issue.

AWS Multilayered-Security (Encryption Cont’d)

How Does It Work?
Essentially, the user first enters their username and password (shown in the form in the login.html file), and after hitting the Signup button, their information is first encrypted via the BEA algorithm, and then sent to the relational database (RDS) and stored on the ‘users’ table. The information present on the ‘users’ table can be seen on the connect_to_db.py script and after signing in, the user can login (form below sign in one) and their information is cross checked with the database via basic SQL queries. It should also be noted that having a passwords table ensures that different identities can only view the users and not the passwords and vice versa.

Due to our reliance on an online service it is also crucial that we set the proper inbound and outbound rules for our database. As it currently stands, our program functions to only allow access to the database instance to certain IP addresses. This is to prevent people online from accessing the data even if they managed to steal our login information.
To keep our access credentials safe, we use AWS Secrets Manager. This service both encrypts our data at rest making it harder to steal through unauthorized access and changes the credentials ensuring that any leaks that do happen have no long term risks on our organization. AWS Secrets Manager in conjunction with the default AWS Key Management Service keys we created also ensures that the RDS itself has an additional layer of security.

Future considerations

As we move forward with this project we have a number of things we intend to change.

Firstly, we will provide more IAM roles in AWS to further open access to more people but limit the data they can access. We also plan to move encryption to AES algorithms (AES256 is standard) instead of BEA as that is more modern and can handle newer attacks. After that we will move from HTTP requests to HTTPS as they are more secure. Having HTTP requests in our Flask functions was simply done in the interest of time despite the server.crt and server.key openssl keys being created. When all that is done we will further restrain and curate our database to make it more resilient to SQL injections. This involves removing the database's ability to contain HTML tags and deny the input of extended URLS.