Networked Password Manager

Matthew Sterrett, Aaron Frost, and Karl Adriano

CS427: Cryptography

Mike Rosulek

March 16, 2022

| Abstract

For our final project, we designed and implemented a networked password manager that is provably secure. In our application, users create accounts on a server by supplying a username and password or alternatively their Google account with single-sign-on(SSO). Once logged in, the user is prompted for a master encryption-decryption password. The master key is then derived by salting and hashing this master password. The user can then add login details for multiple websites which are stored and encrypted in their local logins file. When the user presses the save button, this logins file is synced to the server which allows clients to later request this remote file from different machines. An eavesdropper who sees this logins file during a sync will gain no information about the original plaintext since it is securely encrypted, making the transfer cryptographically secure. In a similar manner, if the server is compromised, there will be no password leakage –only the usernames for registered accounts. We built this password manager with two web application frameworks: Express and Electron. We also used the Google Authentication Library to implement SSO. This password manager is a great project to show on a portfolio as it demonstrates the core principles of a usable and secure password storage scheme with modern web frameworks and security libraries. Our password manager may also benefit a wider community by exposing the inner workings of password management.

| Introduction

As many websites require users to register with new accounts, users are tasked with memorizing numerous login details. Some users find it difficult to keep track of which websites correspond to which login details. This leads users to reuse passwords across different websites which is not advisable. With password reuse, a compromise of one website’s passwords exposes a user's password across multiple platforms. For this reason, it is recommended to use different passwords for each website which leads users back to the original problem of remembering multiple login-to-website associations. A solution to this problem is to use a password manager. A password manager assists the user by managing an encrypted file that stores user login credentials for each website. This logins file can only be read by decrypting under the user’s master password. A password manager is a great solution to our original problem as it minimizes the human memory required to just a single credential. Here, we’ve created a password manager that provides this standard functionality and also allows for remote access to this logins file using a client server architecture.

| Usage

Application Setup

First, create a folder in the root directory called “certs”. In this folder, create a key/certificate pair named “key.pem” and “cert.pem”. This can be done using *openssl* with the following command in the “certs” folder:

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem
```

When creating this key pair, set the “Common Name” field to “localhost”. Assuming *npm* is already installed, Windows users can run *install_requirements.bat* whereas Mac users can run *install_requirements.sh*. Running *chmod +x install_requirements.sh* is required if permission is denied on a Mac. This file will install the needed dependencies: Express, Electron, and the Google Authentication Library. Then, start the server and client by executing *start_server.bat* and *start_client.bat* for Windows or *start_server.sh* and *start_client.sh* for Mac.

Functionality

If the client is started successfully, a window will appear presenting two options: “Login to server” and “Offline login”. If this is the user’s first time, they should press “Login to server” and then manually sign up for an account, or use Google SSO, which opens a browser window and deep links back to the Electron app. After signing up and

logging in, the user is prompted for a master decryption password. This master password is salted and hashed to derive the user's master encryption decryption key, which is not stored anywhere. The user can now add, edit, delete, and save/upload their login credentials as desired. This file is saved locally in *"Client/data/<username>.json"* and remotely in "*Server/data.json*".

| Security

Encryption

The user's login file is encrypted in a provably secure way. We use node.js's built-in secure *crypto* module to securely handle encrypted data.[1] Both client and server data files store the salt and ciphertext of their logins file. The salt is randomly generated using `crypto.randomBytes(16)`. This method generates 16 bytes of highly random data. We also chose to use Argon2id for password-hashing as it is the winner of the Password Hashing Competition.[2] The Password Hashing Competition was organized by cryptography and security experts as an open competition to raise awareness for the need of strong password hashing algorithms and their standardization. Argon2 won the competition due to its high time and memory complexity which makes brute forcing the space of possible passwords non-viable. We are confident in our decision to use Argon2 to hash master passwords.

Generating a key from a password is far from ideal, as the space of reasonable passwords is much smaller than the space of possible 256-bit keys. Still, deriving our master encryption decryption key with Argon2 is more secure than general hash functions not designed for low-entropy inputs like passwords, and is obviously safer than using the password directly as a key. Our program generates an IV with `crypto.randomBytes(16)`. Then, the program creates the ciphertext by calling *`crypto.createCipheriv("aes-256-cbc", master key, IV)`*. We have chosen to use the AES-CBC algorithm which is a cipher block chaining mode and one of the most used symmetric key encryption algorithms. AES is NIST-certified and, like Argon2, has also been adopted as the standard symmetric key cipher of choice[3]. Note that we do not need anything more than CPA security here since the ciphertext is transferred over TLS, so it should not be possible to capture the ciphertexts as they are transmitted. Further, if the attacker gets the encrypted password file, there are no online attacks available and thus nothing except CPA security is needed. These properties create ciphertexts that are indistinguishable from random.

Decryption

To decrypt, the program derives the master key by hashing the entered master password with the stored salt. The IV and ciphertext are then extracted from the local

data store. The program then uses `crypto.createDecipheriv((“aes-256-cbc”, key, IV).update(ctxt)` to decrypt and obtain the plaintext logins file.

| Conclusion

Creating this networked password manager taught us how to communicate sensitive data in a client server architecture. We learned how to implement different cryptographic building blocks to support one major design, which provides a simple method for average website users to securely store and recall their passwords. Our password manager showcases our learned knowledge of cryptography and its application to modern web frameworks.

| References

1. https://nodejs.org/api/crypto.html#crypto-module-methods-and-properties
2. https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf
3. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf