Mitchell Radford
Michael Carris
Nour Rahal-Arabi
Rosulek
CS 427
7 March 2022

PassVault Password Manager
Assessment of Security

1- Hashing with Argon2

1.1- Overview

There are a few hashing algorithms built for passwords specifically, with Argon2 being one of them. Argon2 is one of the more recent ones and was built to replace the previous standards of bcrypt and scrypt. Argon2 provides the ability to generate salted hashed passwords and then store and verify them. These are the basic functionalities of any password hasher which is required for our project to function properly. The hashing in our project is used in two facets: user account password storage and as an encryption key.

As discussed in Section 3, when the user makes an account, it generates and stores a salted hash in a users file that holds all users with their usernames and passwords. This is the only place where hashes are stored at all. The only other place hashes are used are to generate the key for the encryption of the user's service passwords. When the user's service information is stored, it's encrypted instead of hashed so that the password manager can send the user their usernames and passwords for each service/software. We opted to use the hashed master password that the user enters to log into the password manager itself as the key for encryption. The reason this is secure is discussed in Section 3.4.

1.2- Pros and Cons

Argon2 provides much more security parameters in comparison to other password hashing algorithms. One of the main ideas behind Argon2 is the idea that it protects against a variety of attacks that are more prevalent with greater computing power, such as GPU-based attacks. Because of this, Argon2 was built to be a memory-hard function that includes memory requirements as a parameter [1]. This provides more security because it means that in order to calculate many hashes, it requires more memory, not just CPU or GPU power. It also helps prevent side-channel attacks because it uses data-depending memory access [2]. Side channel attacks rely on the fact that data is stored in a way that allows access to information or affects the system instead of the program itself [3]. Finally, it's more resistant to time-memory tradeoff attacks which are also, as the name states, memory based attacks that are solved by the memory-hard functionality provided by Argon2.

The reason that these are important is because some of these attacks are not as well prevented by bcrypt, which has been an industry standard password hashing algorithm. However, it's still used because of Argon2's main downside, which comes from the fact that it's somewhat resource intensive. Since we are implementing and checking passwords in a file locally and therefore 'offline', the memory issues are not as much of a problem. Despite this, these parameters are all individually configurable meaning that regardless of the system this is implemented, security can be tailored to fit system specifications, such as if a system has more RAM than CPU power.

The other comparison with Argon2 is why use it over traditional hash algorithms such as SHA-256 or similar/better. In order to implement them in a way that is secure for our use case, we would need to create an iterative process among other features to allow for the computational power to generate and verify hashes [5]. This leaves a lot of room for insecure structures when instead there are verifiably secure alternatives, such as Argon2. The reason we want iterative, or more applications of the hash algorithm over the same input, is because it allows us to create a hash that would require more resources for an attacker to break it.

1.3- Overall Security

There are plenty of papers which analyze and prove the security of Argon2 based on the number of features and the way it implements its algorithm. It's recommended specifically by OWASP due to these abilities along with many newer reports published on the topic [4]. It's given such praise because it won the 2015 Password Hashing Competition which contained a variety of specifications to compare password hashing algorithms. So far the site still recommends Argon2 as the premier password hashing algorithm [6]. Any of the various features of Argon2 and Argon2 itself that prove that it's secure against a variety of attacks, as discussed in the previous section, can be found in Section 5 of the paper written and updated by the developers of Argon2 [1].

1.4- Selected Security Parameters and Justification

Parameter values are separated by numerical values and system scalable values. System scalable values are those that are respective to each system such as the amount of CPU or RAM power dedicated to hashing whereas numerical values are salt or hash length sizes. Since Argon2 is scalable to a variety of systems based on resources available, the numerical parameter settings recommended by Argon2's developers are what we used for our project here. Otherwise, a reasonable default value is inserted for system scalable values. For reference, all the lengths are in bytes and the memory use setting is in KB.

For our hash and salt lengths, we set 32 byte (256 bit) and 16 byte (128 bit) lengths respectively. Both are above or at recommended defaults but the hash length is provided specifically to match the AES encryption key, which is used for when the user logs in. The default value for time cost, which is the number of iterations over memory the hashing algorithm runs, is 3 because over 3 passes of the algorithm would get rid of them from memory entirely [1]. We chose to double that to 6 for more security since it didn't affect our time too heavily with the other settings selected. We selected 1 GB of RAM and 4 threads for parallelism in order to meet recommendations and also ensure that it isn't too taxing for most systems or take too long for the user experience. With a better computer, one could easily upgrade it to 2 or even 4 GBs along with more threads for parallelization and thus speed up this process and make it more secure. It should be noted that a program not run on a command line could potentially run this in the background. The document proving Argon2's security states that for both the amount of memory and amount of time, there is no specific "insecure" value that can be inserted into those parameters, but obviously more is better [1].

Finally, the last important note about Argon2 is that it’s stated to be optimized for more dangerous settings where access to the same machine doesn’t mean that an attacker could break the hashes or access them from memory [1].

2- Encryption using PyCryptodome

2.1- Overview

Advanced Encryption Standard (AES) is a secure block cipher with a fixed block length of 16 bytes. AES is desirable in modern encryption schemes due to its efficiency and staunch security, and has remained the NIST standard for protecting electronic data and general encryption since 2001 [8]. The scheme was introduced as a replacement for Data Encryption Standard (DES), the previous NIST standard. AES is heavily utilized by governmental agencies and military entities for protecting classified information, as well as commercial services such as password managers and social media platforms. The encryption scheme provides encryption and decryption functionalities using a single key (symmetric encryption) of size 128 bits, 192 bits, or 256 bits. The PyCryptodome implementation of AES is used in the project to encrypt and decrypt the user's service information, as well as verify the identity of the user based on their credentials [7].

2.2 - Security

AES is computationally secure against brute force attacks as well as all other known attacks, regardless of the chosen key size. No attack on AES has been deemed feasible in modern cryptography. While there have been attacks against AES that have been shown to succeed faster than brute force attacks, none have been successful in practical terms. Certain known attacks on AES - such as biclique attacks - have claimed an advantage of a 10% reduction in computational expense when compared to traditional brute force attacks [9]. Even so, it is expected for such an attack to take billions of years to break even a 128 bit key. For these reasons - in addition to its mathematical efficiency and variable key size - AES is maintained as the encryption standard in the United States and around the world, and is used to secure most government transcripts classified as top-secret. Almost all modes of AES encryption/decryption are considered secure in all circumstances, with the exception of Electronic CodeBook (ECB) which has been shown to be semantically insecure; i.e., simply observing an ECB ciphertext can leak information about the plaintext, in some contexts. CBC mode, however, is most commonly used and leaks no information about the plaintext. The project will utilize CBC mode as part of its secure implementation.

2.3 - Implementation Parameters and Justification

The project's encryption and decryption functionalities are implemented using a PyCryptodome AES object with Ciphertext Block Chaining (CBC) mode. The encryption scheme implementation makes use of a randomized initialization vector (IV) and a hashed master password as the key for encryption. The IV is stored publicly in a text file along with the user's encrypted credentials. The IV is made public and is not encrypted or hashed itself - this aligns with AES security principles, as it is assumed that an adversary may have access to the IV of an AES encryption. The master password is hashed using Argon2, a provably secure hash algorithm by modern standards. An adversary would have to find a collision in Argon2 in order to acquire the key, a task that is computationally and mathematically infeasible. The plaintext is padded to achieve a length of a multiple of the block size (16 bytes) then encrypted using CBC mode. These parameters were chosen in alignment with common CBC practices for secure encryption/decryption.

3- Architecture and Design Choices

3.1- Overview

In a real world application, data would be stored on a secured server. Due to constraints, however, our system simply runs in the command window with local files. Because of this, our system was designed to keep the stored data secure, despite the fact that it is stored locally. Anything referencing hashing in this section refers to salted hashes from Argon2. Likewise, encryption and decryption mentioned here refers to encryption using AES-CBC.

3.2- User Login Storage

User login data consists of a **username**, **an IV,** and a **salted hashed password**. When a user signs up, their username, and salted password hash are stored into ***users.txt***. The username is stored here as a plaintext. To keep the rest of the user's data secure, a user's master password is stored as the salted hash of the password rather than a plaintext. Argon2 generates a secure hash, so it is secure to store the salted password hash directly, even with the salt being public. Additionally, the IV is updated every time data is encrypted.

If an attacker opens ***users.txt***, they will be able to determine which user has which salt and which hashed password is theirs. However, this information by itself is useless to an attacker. Since the hash is secure, the attacker cannot learn a user's password from this data. As a result, the user login storage is secure. Additionally, the salt for the hash is stored as part of the hash, which can be retrieved by Argon2.

3.3- User Specific Filenames

Originally, each user would have had a seemingly random filename to hold their data. This would have worked as described below.

The filename would be generated by encrypting the corresponding username with the statically salted (always the same for every input) hash of their password. Because Argon2 generates secure hashes, the salted and statically hashes of any message are unrelated, an attacker cannot know the hash without knowing the password. Therefore, this would be a secure and random encryption key.

As a secure encryption key, the corresponding ciphertext would be seemingly random, providing an attacker no useful information to determine the owner of a specific storage file. For these reasons, the names of user files are secure.

However, this idea was scrapped as a result of changing encryption IVs and time constraints. Instead, each user has a file following the naming convention of 'username.txt'. If this were to be used, we would use a secure encryption algorithm that does not require an IV so that the filename may be the same every time the username is encrypted.

Despite not having this feature, our application would remain secure in a real life scenario because all the files would be on a secure database. This functionality only helps by applying an extra layer of secrecy.

3.4- User Data Storage

As a plaintext, a user's data file their information in the following format:

```
SERVICE NAME 1
USERNAME
PASSWORD
SERVICE NAME 2
......
```

However, all of this data is encrypted upon storing. When any part of the file is changed, the entire ciphertext content is decrypted and the plaintext is re-encrypted together. Since the encryption key is a statically salted hash of the user's master password (which cannot be known without knowing the master password), it will create a secure ciphertext. This is because the master password isn't stored or printed anywhere therefore there's no way for an attacker to get the password without brute forcing. Additionally, knowing the layout of the plaintext does not help an attacker distinguish the actual content of any plaintext line since the ciphertext does not necessarily distinguish where a line starts or stops. For these reasons, the data within a user's file is secure.

3.5- Limitations

In general, this system was designed so that the storage of a user's login information, corresponding filename, and file contents are secure from attackers that do not know the user's password. However, just like in a real application, knowing both a username and password allows one to bypass all of this security.

Unfortunately, there are few good ways to counter this, since this is how a regular user would also login. A possible way to combat this would be to implement two-factor authentication, which likely would be done if this were a real application. However for the sake of simplicity, we did not implement this feature.

References

1. Biryukov, A., Khovratovich, D., & Dinu, D. (2017). (rep.). Argon2: the memory-hard function for password hashing and other applications. Retrieved March 15, 2022, from https://www.cryptolux.org/images/0/0d/Argon2.pdf.
2. Chen, B. (2019). (rep.). Memory-Hard Functions: When Theory Meets Practice. Retrieved March 15, 2022, from https://escholarship.org/uc/item/7x4630qv.
3. Wright, G., &amp; Gillis, A. S. (2021, April 6). What is a side-channel attack? SearchSecurity. Retrieved March 15, 2022, from https://www.techtarget.com/searchsecurity/definition/side-channel-attack
4. OWASP. (n.d.). Password storage cheat sheet. Password Storage - OWASP Cheat Sheet Series. Retrieved March 15, 2022, from https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
5. Nidecki, T. (2019, July 2). Insecure default password hashing in CMSS. Security Boulevard. Retrieved March 15, 2022, from https://securityboulevard.com/2019/07/insecure-default-password-hashing-in-cmss/#:~:text=Password%20Hash%20Security%20Considerations&amp;text=The%20SHA1%2C%20SHA256%2C%20and%20SHA512,should%20always%20use%20a%20salt
6. Password hashing competition. Password Hashing Competition. (n.d.). Retrieved March 15, 2022, from https://www.password-hashing.net/
7. Classic modes of operation for symmetric block ciphers. (2021, October 19). In *PyCryptodome API Documentation*. Retrieved from https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html#cbc-mode
8. Federal Information Processing Standards. (2001, November 26). ADVANCED ENCRYPTION STANDARD (AES). Retrieved from https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf
9. Gstir, D. (2012, October 15). Analysis of Recent Attacks on AES. Retrieved from https://diglib.tugraz.at/download.php?id=576a78078c529&location=browse