Password Manager Desktop App

CY 4740/6740: Network Security

Professor William Robertson

Riddhi Adhiya, Anthony Gavazzi, Justin Raynor, and Mobolaji Rotibi

18 December 2020

Introduction

Earlier this year, NordPass analyzed the most common passwords in 2020 using data from security breaches, and found that the top five most commonly exposed passwords were '0123456', '123456', '0123456789', '123456789', and 'qwerty'.[1] Each of these passwords took less than one second to be cracked, and exposed more than 60 million times in total.[1] According to LastPass' third annual global password security report, it was found that employees reuse a password an average of thirteen times.[2] Remembering so many different passwords can prove to be difficult for the average person, considering the number of different accounts that each individual has for online banking, shopping, education or work, as well as other categories. The combination of both the overuse of common passwords, as well as the reusing of passwords such a large number of times, leaves many individuals vulnerable to data compromise. However, using a password manager can help in mitigating data compromise, as it takes away the burden of remembering passwords and can encourage the use of stronger passwords. Password managers can also make the process of changing a password quicker and easier in the event there is a security breach.

Rationale and Requirements

Many password managers currently exist on the market, allowing consumers to weigh the pros and cons of each before choosing one to use. However, creating a password manager allows for a greater understanding of both the methodology behind how the software works, as well as the different ways in which a password manager can be attacked. For this project, a desktop password manager was created in Java, using a model-view-controller design construct. The manager encourages the use of strong passwords by allowing users to select a password length and generating random passwords of this length, using a variety of letters, numbers, and special characters. A single master password for the manager, which is set by the user, is used to decrypt the entire password database. This master password is stored in a hashed format rather than plaintext in order to increase security. The source code for the password manager can be found at https://github.com/brotibi/PasswordManager.

Process:

I. Password Generation

Passwords are often the first line of defense in maintaining controlled access to applications and, when used correctly, provide a mechanism for ensuring confidentiality availability and authenticity. Several research studies have looked at user behavior, password entropy calculations and how users form, remember and store passwords and conclude that user defined passwords are often weak, re-used or guessable.[3,4] From looking at commonalities in captured password lists, given platform password rules and social engineering, attackers have a large plethora of tactics to conduct password attacks.

Proper password formation techniques and password quality are vital to minimizing these potential attacks and password compromise.[5] A properly formed password manager will allow users to create more complex, less guessable and uniquely strong passwords without having to remember, write down or compromise the password itself. For these reasons, we first focused on creating an algorithm to form a randomly generated password based on user defined rules and preferences.

Because each platform has slightly different rules for password strength including number of characters, upper and lower case letters, numbers, special characters and what passwords can and cannot start with, we wrote our password generation tool to incorporate all of these rules as options that can be selected by the user. Additionally, the user can select the number of passwords to be generated, which follow the user selected rules. With the ease of producing randomly generated and strong passwords quickly, users can avoid the common password forming mistakes cited in the literature, minimize common password attacks (such as dictionary and rainbow table attacks) as well as maintain a quick mechanism for password revocation if a compromise occurs. Once we created the ability to generate strong and unique passwords that follow platform rules, we then focused on building mechanisms for passing, hashing and securely storing the passwords.

II. Hashing, Encryption and Decryption

Although password manager applications help users create strong passwords and avoid reusing passwords, they offer attackers a promising target. If an attacker compromises a password manager, the attacker gains access to every single one of the user's accounts. To protect against this, a password manager must encrypt its contents using a master password which only the owner knows. The encryption must be sufficiently strong to prevent the attacker from ever decrypting the data, and the master key must not be stored anywhere on the system.

When a user creates a new password database, they provide a master password for it. Our password manager immediately generates a random 16-byte salt and computes the SHA-1 hash of the master password with that salt. The password is hashed rather than encrypted because the password manager only needs to know when a user has entered the correct password, and never needs to retrieve the original value. This hash is stored in the password database. In addition, another random 16-byte salt is generated and is stored in the database to be used in a key-derivation function during encryption and decryption.

All passwords managed by the app are identified by a user-provided tag. Both the tag and the password are encrypted based on PKCS #5, a widely-used password-based

cryptography standard.[6] Once the provided master password is validated against the hash, the aforementioned random 16-byte salt is used to compute the SHA-1 hash of the master password. Note that because of the salt, this hash is not the same as the one used to verify that master password. This hash is used to encrypt and decrypt the tags and passwords. Without knowing the master password, an attacker cannot obtain the key and thus cannot decrypt the database. In addition, because the password is hashed using a random salt, precomputation attacks are impossible, even if an attacker knows that a weak password is used somewhere in the database.

Worth noting is that SHA-1 does not provide collision resistance, meaning that two distinct passwords may hash to the same value, regardless of salt or iterations.[7] However, this fact does not impact the security of the password manager because collisions only occur when the text being hashed is longer than 64 characters. A 64-character password is overkill, especially if it is sufficiently entropic, and it is assumed that a user would not choose passwords that long, especially not for the master password. Even if a password over 64-characters were chosen, an attacker would still have to use brute force to find a collision with that password's hash, and no password manager is truly safe against brute force attacks in the first place.

III. File Manager for Password Storage and Retrieval

Local Passwords Managers store passwords locally in a file. These files are an obvious target for attackers, for they give them the potential to gain access to all of a user's passwords. To protect against attacks it is essential that not only are the passwords hashed but all other information such as usernames and urls are also hashed.

The only thing the password manager inherently knows is the directory of the password database file. When a user creates a master password, this also yields a database file encrypted with an AES Block Cipher (HMAC-SHA1 Hash) derived from the master password. An initialization vector can be generated randomly each time a database is saved, allowing for multiple database files to be encrypted using the same master password. To both read and write contents of the database file while maintaining data integrity the AES Cipher is needed.

If someone was to obtain the encrypted database file they would need to also obtain the master key to be able to access the information stored on there. Without the master key, the file would be useless, any efforts made to change the file would make the file unreadable, and since the hash used to encrypt the file is salted it provides some level of protection against precomputation attacks.

It is worth mentioning that encrypting the database file does not necessarily protect against specialized attacks on the password manager, it is at its core just an additional layer of security, the user must also ensure that no specialized malware is downloaded on their device. If an attacker was able to get hold of your master password somehow, the entire database would be compromised. The primary goal is to ensure the authenticity and integrity of the data in the file via the use of a HMAC-SHA1 Hash of the plaintext.

Mitigating Attacks

To assess the security of our password manager, we consider three high-level threats and how our application protects against them. The first attack we consider is the existence of a malicious co-resident application that can invoke any interfaces exposed by the password manager and spoof the password manager UI. The password manager does not expose any interfaces invokable by other applications, so the first threat is handled by the very nature of the application. However, there is no way for the password manager to protect against a spoofed UI. There are no elements of the UI that cannot be spoofed perfectly by an attacker and there is no app behavior that cannot be implemented by the attacker as well. The user of the password manager must ensure that nobody has been able to install such an app, and must double check that the application they are launching is indeed the real password manager.

A second high-level threat we considered is what happens if an attacker obtains the encrypted database itself. As discussed in the section on hashing, encryption, and decryption, the master password is hashed with a random salt before being stored in the database, so the attacker cannot trivially obtain the master password, and cannot use precomputation attacks because of the salt. The master password is hashed again using a different salt to obtain a key used to decrypt the remainder of the file. The tags and passwords stored in the database are encrypted with PKCS #5, which is known to be secure. Thus, an attacker cannot decrypt the tags and password without the key, which they cannot compute without the master password.

The third high-level threat we consider is when an adversary obtains a forensic dump of storage and memory. The password manager cannot perfectly protect against this because the master password and decrypted passwords must be in memory for at least some time. Otherwise, the master password could not be verified and the user would not be able to see or edit their passwords. In addition, because the application is written in Java, any memory allocated persists in memory until the garbage collector removes it, so it is not enough to allow sensitive data to go out of scope to protect it. To protect against this threat as much as possible, the managed passwords are kept encrypted in memory until they absolutely have to be decrypted, and are re-encrypted as soon as they can be. Further, sensitive data is never stored as a string object, since strings in Java are immutable. By using char or byte arrays instead, the password manager can explicitly overwrite the sensitive data when it is no longer needed

Conclusion

The password manager implemented in this project includes password generation based on a user-provided length, allowing users to choose between unique passwords that are hard to guess. A master password, which the manager will hash, is used to encrypt and decrypt all other passwords that are stored in the database. All passwords are stored locally in an encrypted file, ensuring that an adversary cannot do anything with it unless the master password is also known. As such, three attacks were considered when developing the password manager: a malicious application that could spoof the UI or invoke any exposed interfaces, an adversary obtaining the encrypted database, and an adversary obtaining a forensic dump of storage and memory. While these three attacks were considered when the manager was developed, in an effort to mitigate future attacks, the application was written in agile to allow for greater flexibility.

Works Cited

[1] "Top 200 most common passwords of the year 2020." *NordPass,* https://nordpass.com/most-common-passwords-list/.

[2] Ng, Abigail. "Everything you know about passwords could be wrong." *CNBC,* 26 December 2019, https://www.cnbc.com/2019/12/26/everything-you-know-about-passwords-could-be-wrong.html.

[3] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman. Of passwords and people: Measuring the effect of password-composition policies. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11, p. 2595–2604. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/1978942.1979321

[4] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. 2010. Testing metrics for password creation policies by attacking large sets of revealed passwords. In Proceedings of the 17th ACM conference on Computer and communications security (CCS '10). Association for Computing Machinery, New York, NY, USA, 162–175. doi: https://doi.org/10.1145/1866307.1866327

[5] W. Ma, J. Campbell, D. Tran and D. Kleeman, "Password Entropy and Password Quality," *2010 Fourth International Conference on Network and System Security*, Melbourne, VIC, 2010, pp. 583-587, doi: 10.1109/NSS.2010.18.

[6] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017

[7] Bynens, M. "PBKDF2+HMAC hash collisions explained." 25 March 2014, https://mathiasbynens.be/notes/pbkdf2-hmac