Modernizing Password Usage in Computing

by

Adam Coffee

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2017

APPROVED BY:

Cindy Norris, Ph.D., Thesis Project Director

Larry Bridges, Second Reader

Dee Parks, Ph.D., Departmental Honors Director

James Wilkes, Ph.D., Chair, Computer Science

ABSTRACT

Modernizing Password Usage in Computing.

(May 2017)

Adam Coffee, Appalachian State University

Appalachian State University

Thesis Chairperson: Cindy Norris, Ph.D.

Even as advances in cryptography have greatly improved users' account security, the concept of the password has endured as the primary means of authentication. Furthermore, users are still largely responsible for choosing and managing their own ever-growing collection of account credentials. Due to the nature of users (and the inherent difficulty in remembering a variety of unique and strong passwords), there exists a propensity for weak passwords to be chosen, for identical passwords to be used on more than one account, and for passwords to be stored insecurely. This thesis explores password strength theory, the YubiKey authentication device, modern cryptographic algorithms, and secure programming techniques that may be leveraged to modernize password usage by abstracting the user away from the continual process of creating, managing, and storing their passwords and other account information.

This work culminated in the development of PassMan, a two-factor encrypted password manager program that offers a suite of tools to mitigate the aforementioned vulnerabilities: A customizable high-entropy password generator, password strength calculator, and auto-type function effectively free the user from the task of creating and remembering their account information, including usernames, passwords, and associated notes.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Cryptography has become ubiquitous in the computing world, where it is often relied on to secure the systems and online accounts of users by relating them to some cryptographic key. Perhaps the most common use-case is analogous to the login procedure for many online services, where users provide a username and associated password which is subsequently run through a cryptographic hash function; if the result matches the password hash for that username in an authentication database, access is granted.

While that is a simple example, the general notion of using a password for authentication has been expanded to many other use-cases in which it is required to derive a key for cryptographic purposes. All modern operating systems support physical access control with passwords, as well as full-disk encryption, which uses such a password-derived key along with an encryption algorithm to provide confidentiality (and optionally, integrity assurance) of data in storage.

Despite these advances in security and privacy, the concept of the password has nevertheless endured as the primary means of authentication in most systems and services. Because users are typically in charge of choosing and managing their own passwords, there is potential for several security issues to arise:

1. Password commonality: Users may use the same password across many accounts, or only slightly alter them for each account (such as changing "BadPassword123" to "BadPassw0rd456").

2. Weak passwords: Because strong passwords are inherently difficult for users to remember, they may self-generate passwords that are statistically weak (such as those containing their birthdate or common English words and number sequences).

3. Insecure storage: Users may keep their passwords in plaintext where others could access them (such as writing them on notes near their workspace).

One comprehensive solution to these problems is the concept of a password manager. This thesis describes PassMan, a password manager program with a simplistic graphical interface that remembers user account information (including account names, usernames, passwords, and notes) and stores them confidentially for later retrieval. In order to further enhance the security of the user's data at-rest, PassMan encrypts it with two authentication factors: the user's master password (something they *know*), and a cryptographically-unique response from a hardware authentication device known as a YubiKey (something they *have*). While these features greatly minimize problem (3), problems (1) and (2) are resolved by removing the user from the password selection process. PassMan includes a password strength calculator and customizable password generator which, when used together, will ensure the usage of high-strength passwords that are unique to every account. PassMan can also autotype usernames and passwords into most other user applications, such as a web browser showing a login form.

Ultimately, PassMan modernizes password usage in computing by abstracting the user away from the continual processes of password creation, management, and storage. This thesis explores the background of techniques and algorithms used to create PassMan, describes similar works, and explains its usage and implementation in detail.

Chapter 2

Background

PassMan uses several modern cryptographic tools and some specialized hardware to improve the user's password security. Ultimately, it should guarantee the following to be an effective password manager:

1. Strong confidentiality of all data relating to stored accounts, requiring *two* authentication factors for access
2. Assured integrity of all aforementioned data, regardless of threat (like storage medium corruption or malicious manipulation)
3. Strong password generation, with unpredictable high-entropy output

Each of these goals requires application of one or more algorithms to ensure reliability of these claims. Firstly, because the data does not need to be shareable, a symmetric encryption algorithm fits well. The Advanced Encryption Standard (AES) was chosen to uphold (1), with the cipher key being comprised of two factors: the user's master password, and a unique cryptographic response from additional hardware called the YubiKey. The latter depends on another algorithm, known as Hash-based Message Authentication Codes (HMAC), which itself relies on part of the Secure Hash Algorithms (SHA) family. These two factors are combined to produce a key of the proper size for the cipher via yet another algorithm called Password-based Key Derivation (PBKDF). Figure 2.1 shows the context in which each of these tools is utilized:

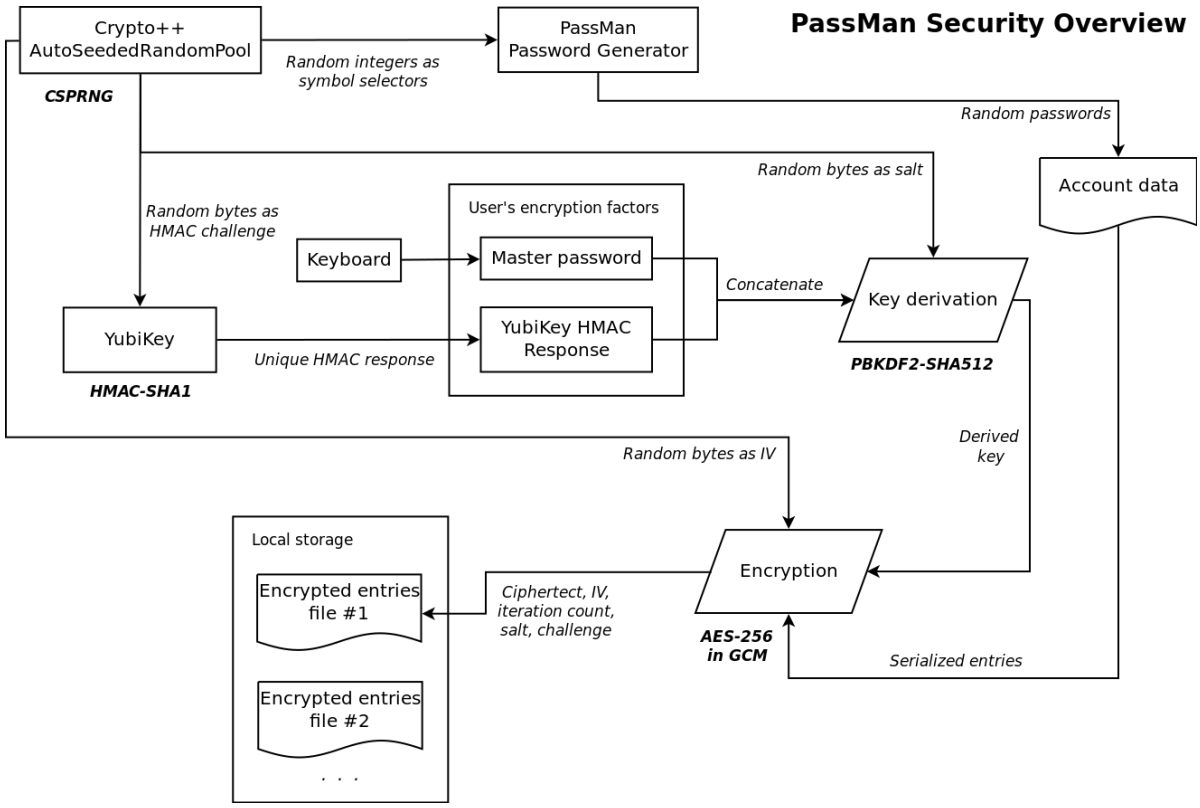


Figure 2.1: PassMan security overview

Guarantee (2) requires using AES in a "mode of operation" that offers extra information about the integrity of the encrypted data; several modes are available, but Galois/Counter Mode was chosen due to efficiency and availability. Strong password generation (3) requires knowledge of password strength theory, and (as with many of these algorithms) a Cryptographically Secure Pseudo-random Number Generator (CSPRNG). Finally, knowledge of secure programming rules is essential to the reliability of these claims when the software is executing.

2.1 Advanced Encryption Standard

The algorithm used by PassMan to store account information confidentially is the Advanced Encryption Standard (AES). Originally named "Rijndael" after its designers Vincent Rijmen and Joan Daemen, AES is a symmetric-key encryption algorithm published by the National Institute of Standards and Technology (NIST) in 2001. It was defined in Federal Information Processing Standards (FIPS) Publication 197, after a selection process narrowed to ciphers with 128-bit block sizes and 128, 192, or 256-bit key lengths [20]. AES has become available in a variety of software cryptography packages, and has been approved by the National Security Agency (NSA) to provide confidentiality of documents up to top-secret classification, provided 192 or 256-bit keys are used [25].

In addition to its broad approval, AES has been significantly accelerated in many modern Intel and AMD processors with the Advanced Encryption Standard New Instructions (AES-NI) extension to the x86 instruction set architecture (ISA). Using these instructions on a Pentium 4 processor, the Crypto++ security library executes AES in Galois/Counter mode at 3.5 cycles per byte of data, versus 28 cycles without [4].

Cipher Security

AES is designed around a common cryptographic primitive known as a "Substitution-permutation network," in which layers of substitution and permutation are repeated for some number of rounds on cleartext data and a key, to produce a transformation to ciphertext. Within this system, plaintext data and the encryption key are passed through substitution boxes (known as S-boxes), and permutation boxes (P-boxes). Figure 2.2 shows an example of a three-round network that operates on 16-bit blocks, where

- $S_1 \ldots S_4$ are substitution boxes
- $K_0 \ldots K_3$ make up the key
- P is the permutation box
- $\oplus$ denotes an exclusive-or (XOR) operation

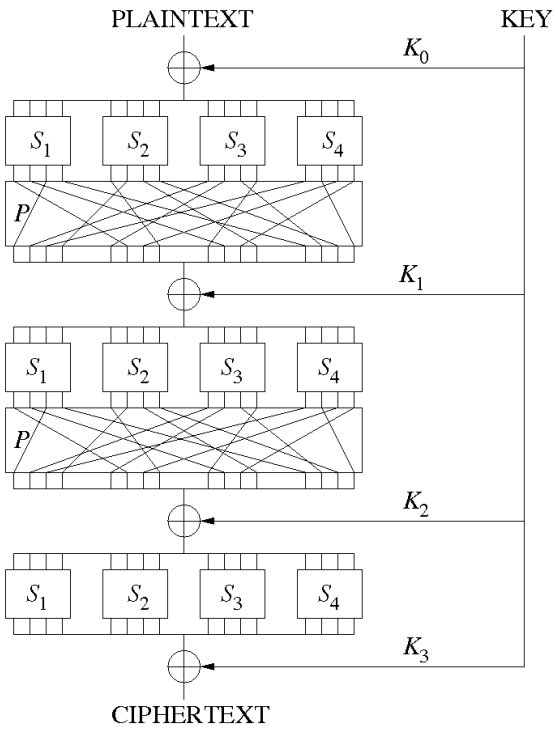


Figure 2.2: Substitution-permutation network example

The S-boxes may be viewed as lookup tables that offer one-to-one substitutions of data, and must be invertible for decryption to be possible. The network is defined to operate on input sizes equal to the cipher's block size. This means that for AES and its 128-bit blocks, the network would be far larger than the one in Figure 2.2, which operates on 16-bit blocks. In software, these tables are often hard-coded two-dimensional arrays; however, on machines with the AES-NI extension, one entire round of AES can instead be executed with the `AESENC` instruction. The goal of this process is to break up the relationship between plaintext, ciphertext, and key, thereby limiting cryptanalysis or statistical inspection.

An abstracted description of the entire algorithm is shown below, where the substitution-permutation network is represented by the four parts of the *Rounds* step. Depending on the key size of 128, 192, or 256 bits, each block of cleartext is subjected to 10, 12, or 14 rounds, respectively. The number of rounds scales with the key size in order to ensure that larger keys fully influence all of the ciphertext [10]. Steps (2) and (4) are included in the round count, so step (3) is effectively repeated 8, 10, or 12 times.

1. *KeyExpansions* — derive round keys from the key using the key schedule
2. *InitialRound*
    (a) *AddRoundKey* — combine each byte of current state with block of round key via $\oplus$
3. *Rounds*
    (a) *SubBytes* — substitute each byte of current state with another via lookup table
    (b) *ShiftRows* — transpose rows of current state a certain number of steps
    (c) *MixColumns* — transform columns of current state with a fixed matrix
    (d) *AddRoundKey*
4. *FinalRound*
    (a) *SubBytes*
    (b) *ShiftRows*
    (c) *AddRoundKey*

The *KeyExpansions* step is an initialization step that occurs prior to computing the rounds. It exists to expand the cipher key into separate round keys. With AES, one key is produced for each round according to its "key schedule." While the details of this step are outside the scope of this description, it suffices to say that this step is necessary to obfuscate relationships between the ciphertext and cipher key. Without this step, cryptanalysis could exploit such a relationship to recover the cleartext.

Galois/Counter Mode

While AES is a strong and efficient block cipher, as its classification denotes, it is only appropriate for encryption of single data blocks. To use it for sequences of data with arbitrary length (such as PassMan's entries file), AES is used in a "mode of operation" that makes it applicable to any data sequence. This is done by splitting the sequence of data into blocks equal to the cipher's block size, and encrypting them iteratively. Unfortunately, problems remain with regards to confidentiality and integrity of the data:

1. Encryption of the same plaintext with the same key will result in identical ciphertext.

2. Manipulation of the ciphertext will result in non-obvious corruption of the plaintext.

Ideally, identical plaintext would always result in different ciphertext, preventing an attacker from relying on statistics to derive the cleartext. This is particularly important in cases where the plaintext contains strings of text, as the probabilities of letters in languages are well-known, aiding statistical analysis. In the case of PassMan, an attacker could also manipulate the ciphertext such that, when decrypted, it produces incorrect account passwords without the knowledge of the user.

PassMan uses AES in Galois/Counter Mode (GCM) to eliminate these issues (with *very* high probability) by using an initialization vector (IV) to influence the initial state of encryption or decryption of a stream of data blocks, and by calculating an authentication tag at the end of the process [22]. In practice, the IV is an array of randomly-generated bytes chosen prior to starting an encryption process, and stored for use in decryption later. It is then used in *addition* to the cleartext and key to produce the ciphertext; since the IV was (ideally) randomly chosen, and now has influence over the resulting ciphertext, problem (1) has been resolved. Note that the IV may remain public information without compromising the ciphertext, as it is ultimately useless without the key.

Cipher modes like GCM which resolve problem (2) provide "authenticated encryption," or the ability to detect any change in the ciphertext. For each block in the sequence of cleartext data, AES is used to encrypt that block, and then the XOR of the resulting ciphertext and a previously-generated authentication code is computed. Because the authentication code of any particular block is now influenced by the content of each prior block, a change in the content of any encrypted block will cause the authentication tag computed at the last block to differ, ultimately indicating a loss of integrity. Software that implements GCM may then throw an exception, allowing the programmer to handle the situation and notify the user.

2.2 Secure Hash Algorithms

The Secure Hash Algorithms (SHA) are a suite of cryptographic hash functions supplied by NIST under FIPS 180-1 and 180-2. While PassMan does not make direct use of any SHA functions, the HMAC and PBKDF2 algorithms used by PassMan do rely on them to accomplish authentication and key derivation. Two groups are in common use today that differ based on the size of their "message digests" or output:

- SHA-1 with 20-byte output, resembling the older, weak Message Digest 5 (MD5) hash
- SHA-2 including SHA-256, SHA-384, and SHA-512, with 32, 48, and 64-byte outputs, respectively

Hash functions deterministically take an arbitrarily-sized data input and produce a fixed-sized output, however *cryptographic* hash functions uphold additional requirements:

1. It is infeasible to produce a message from its digest (non-invertible, preimage attack resistance).
2. A small change in a message dramatically changes its digest (avalanche effect).
3. It is infeasible to produce two differing messages with the same digest (collision resistance).

The function should ideally also be easy to compute for any message, as hash functions are often used as primitive pieces in larger algorithms, such as those for key derivation, password storage, and message authentication.

All members of the SHA family accomplish the above goals by running message data through a compression function for a certain number of rounds. The compression function for SHA-256 is shown in Figure 2.3, where

- $A \ldots H$ are single bits of the hash input (message)
- W_t is the t^{th} word of the message
- K_t is a set of constants used to initialize the hash
- $Ch(E, F, G) = (E \wedge F) \oplus (\neg E \wedge G)$
- $Ma(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$
- $\Sigma_0(A) = (A \gg 2) \oplus (A \gg 13) \oplus (A \gg 22)$
- $\Sigma_1(E) = (E \gg 6) \oplus (E \gg 11) \oplus (E \gg 25)$
- $\boxplus$ is addition, modulo 2^{32}

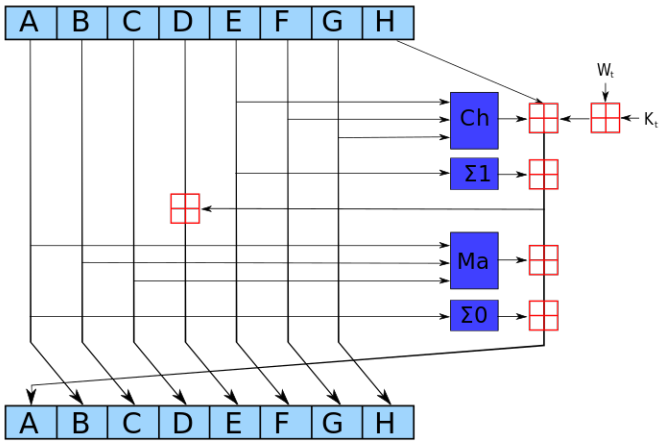


Figure 2.3: SHA-256 compression function

Avalanche Effect

The Avalanche effect describes the desirable behavior of hash functions that are highly sensitive to changes in input. Simply put, a small change in the message (even a single bit-flip) should result in a very different hash. Hash functions that do not exhibit this have weak randomization, and are unsuitable for use in larger cryptographic systems, as an attacker could then make predictions about the input when given the output. Figure 2.4 exemplifies the effect inherent to SHA-1 by flipping only single bits in a short message.

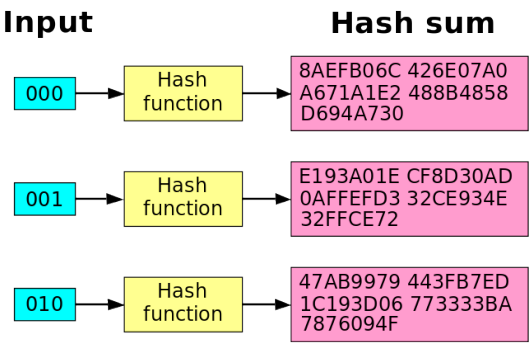


Figure 2.4: SHA-1 avalanche effect

Preimage and Collision Resistance

The preimage resistance quality of a hash shows that it is infeasible to find the function's inverse. Given a hash function with L bits in a given message digest, a brute-force attack would take 2^L computations of that function in the worst case to discover the message input corresponding to that digest. This is part of the reason why SHA-1 (with $L = 160$) is being phased out in favor of SHA-2 varietals; their larger digest lengths $L \in \{224, 256, 384, 512\}$ exponentially raise the amount of work needed. In practice, this means that it is so computationally intensive as to be extremely improbable that (assuming digest d is known), the message m could be found where $H(m) = d$.

By extension, a hash function with strong collision resistance will be highly resistant to finding two messages m_1 and m_2 where $H(m_1) = H(m_2)$. A hash function with L bits in the message digest will require, on average, $2^{L/2}$ computations of that function to find messages m_1 and m_2 with equivalent hashes. It is important to note that collision resistance cannot be absolute; because hash functions inevitably compress their inputs, the pigeonhole principle guarantees that some differing values will map to the same digest. A good hash function will simply ensure that this is exceedingly rare.

2.3 Hash-based Message Authentication Code

Hash-based Message Authentication Codes (HMAC) use a cryptographic hash function and key to provide integrity assurance and authenticity of some message. They are commonly used in cipher modes to help provide authenticated encryption, and may also be used as authentication factors (such as in the YubiKey), assuming the key is unique to the entity being authenticated. PassMan consumes HMACs from the user's YubiKey as the second encryption factor for their entries files. Originally standardized in RFC 2104, HMAC-SHA1 and HMAC-MD5 are used today in the Internet Protocol Security (IPsec) and Transport Layer Security (TLS) protocols [11]. As it is defined there, Equation 2.1 shows the HMAC function.

$$HMAC(K, m) = H((K' \oplus opad) \parallel H((K' \oplus ipad \parallel m))) \tag{2.1}$$

- K is the key
- m is the message for authentication
- H is a cryptographic hash function like SHA-1
- K' is the key after padding or hashing K to fit the input size of H
- $\oplus$ denotes XOR operation
- *opad* is an arbitrary outer padding used to create input with the size expected of H
- $\parallel$ denotes **concatenation**
- *ipad* is an arbitrary inner padding used to create input with the size expected of H

The inner application of the hash function serves to cryptographically intertwine the key and message such that it is computationally infeasible for an adversary to produce the same digest without the key. Naturally, the degree of assurance is tied to the hash function used, but is also reliant on the strength of the secret key.

The outer hash function prevents a length-extension attack, where the message corresponding to a previously-used MAC is extended then re-hashed to produce another valid MAC. Despite the weaknesses discovered in MD5 and SHA-1 regarding collisions, HMAC-MD5 and

HMAC-SHA1 are still reliable for authenticity assurance, as it has been shown that strong collision-resistance characteristics are not strictly required [1].

2.4 Password-based Key Derivation Function

When using passwords as encryption keys, it is common to run them through a "key derivation" process with a hash function that normalizes the key length and ensures some minimal level of security (in the case that a weak password is used). As described in RFC 2898, the Password-based Key Derivation Function (PBKDF) family consists of two standards for producing cryptographic keys from passwords [15]. The second version paired with the SHA-512 hash function (PBKDF2-SHA512) is used by PassMan to derive the key for encrypting the entries file from the user's two authentication factors. Key derivation functions are used to solve two major problems:

1. The computation of cryptographic keys from passwords can be very fast on modern hardware, enabling adversaries to brute-force search the key space for the correct password

2. Given the key derivation algorithm, adversaries can use tables of pre-computed hash outputs (rainbow tables) to quickly search for the correct password

PBKDF2 solves these issues by applying a hash function to the input password along with a "salt" value, then repeating this process some number of times, eventually resulting in the derived key. The salt is a randomly-chosen sequence of bytes that is concatenated to the input password prior to starting the derivation process. By appending this random value, pre-computed tables like in (2) cannot be used, since they do not contain inputs including the salt.

Because the password and salt combination is repeatedly hashed (a process called "key stretching"), the attack in (1) is effectively hindered: the amount of computational work an adversary must do to make a single password guess increases linearly with the specified cycle count. Through this process, some minimal amount of key strength is maintained, as the amount of work and time to discover even a weak password is significantly increased.

2.5 Password Strength

From a purely mathematical perspective, passwords are simple concatenations of symbols, where each symbol has been selected from a predefined set. In the ideal (maximum security) scenario, each symbol element will have been randomly chosen according to a uniform probability distribution, thus guaranteeing that each symbol has equal chance of selection. In this case, the information entropy (measured in bits) contained in a randomly chosen password is given by Equation 2.2 below:

$$H(L, N) = log_2(N^L) = L * log_2(N) \tag{2.2}$$

- N is the size of the set containing all possible symbols
- L is the length of the desired password

This equation is what PassMan uses to estimate password strength. Note how it relates closely to the number of *possible* passwords given by N^L, which is visualized on a logarithmic scale in Figure 2.5 for each symbol set commonly found on keyboards:

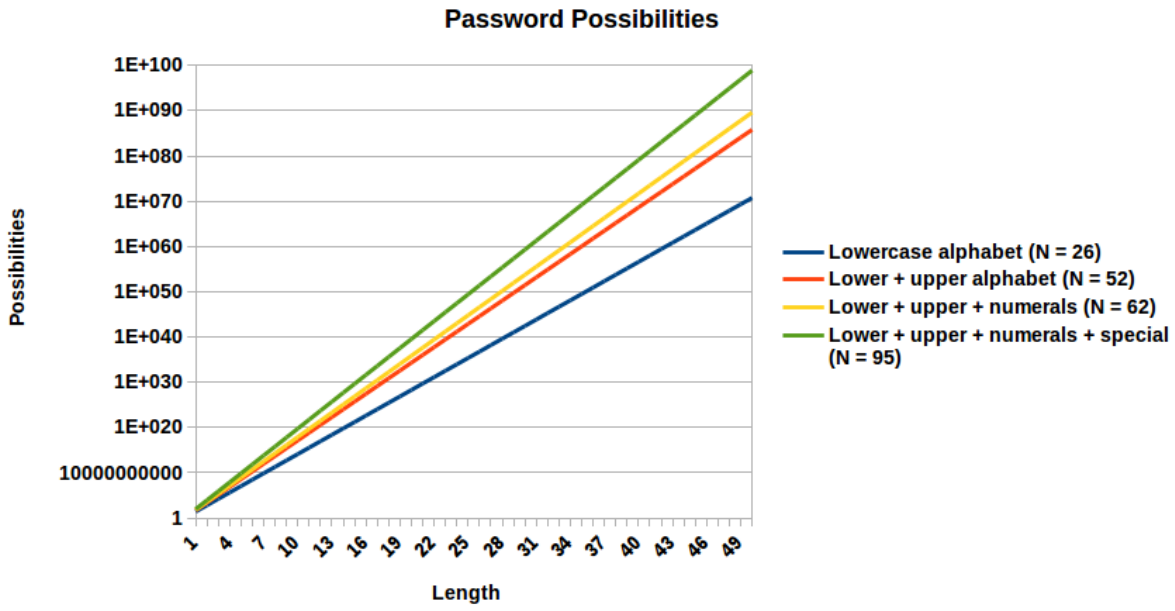


Figure 2.5: Password possibilities with keyboard symbol sets

Unfortunately, this model does not correlate well with how people often select passwords. For example, the password "PassWord" consists of symbols from the lowercase and uppercase alphabet, meaning $N = 26*2 = 52$. The length $L = 8$, so there are $52^8 \approx 5.35*10^{13}$ possibilities. A brute-force search would require checking on average half of the passwords to find the correct one, yet a dictionary attack crafted with knowledge of common password choices would discover this one almost immediately.

This is the core issue in the relationship between people and their passwords: Statistical knowledge of commonly-chosen passwords, and to an extent, knowledge of a specific person's attributes like their birth-date, may greatly increase an attacker's chance of guessing that person's passwords. The inherent difficulty in remembering strong passwords no doubt further aids attackers in this regard.

The solution to this issue is to remove the person from the equation, which is exactly what PassMan accomplishes as a password manager. Generating passwords with the aforementioned uniform distribution (in the form of a built-in password generator driven by a random number generator) ensures that the calculated entropy of a password actually extends to real-world strength.

2.6 YubiKey

The YubiKey, shown in Figure 2.6, is a hardware authentication device designed by Yubico that supports a variety of standardized authentication methods [30]. HMAC responses from this device are used by PassMan as the second encryption factor. It operates over the Universal Serial Bus (USB), and comes in a variety of models suited for differing use-cases, from generating Universal Second Factor (U2F) tokens to managing OpenPGP private keys. The YubiKey may be used as a second authentication factor with password managers like LastPass and KeePass [14, 17]. Organizations including Facebook and Google use it to enhance employee security, and also offer authentication compatibility with their services [27]. Several recent models are in the process of being validated for compliance with the Federal Information Processing Standard (FIPS) Publication 140-2 at the highest assurance level, and many earlier models have already been validated [24].

Figure 2.6: YubiKey USB device

Features

As previously mentioned, YubiKeys come in a range of models with increasing capabilities. The YubiKey 4 is Yubico's full-featured model, while the YubiKey NEO also supports Near Field Communication (NFC) to extend many capabilities to mobile computing devices. Regardless of these differences, any YubiKey may be used with PassMan as a second encryption factor, as every model supports the Challenge-Response mechanism.

YubiKeys have two "configuration slots" that may be programmed via Yubico's YubiKey Personalization Tool with the user's choices. Several of the features produce some form of One-Time Password (OTP); these ephemeral passwords are only valid for a single use or over a short time window, thereby limiting an adversary's ability to reuse a previously-obtained password to gain access to some service. The Yubico OTP feature generates an encrypted password only usable once, often to authenticate with an online service such as Dropbox or Google. While this feature requires coordination with Yubico servers, the Initiative For Open Authentication Time-based One-time Password (OATH-TOTP) option will generate a short one-time password for any supporting service. Hash-based Message Authentication Code One-time Password (OATH-HOTP) authentication is also offered, which relies on a secret key rather than time-period validity.

The feature leveraged by PassMan as a second factor in entries file encryption is Challenge-Response, which uses a Hash-based Message Authentication Code with Secure Hash

Algorithm 1 (HMAC-SHA1). As this method requires no Internet connectivity, it is ideal for offline authentication. This method relies on a 20-byte secret key being used in the HMAC computation, a value which is set by the user in the personalization software, and stored solely inside the device's secure memory in order to cryptographically identify the YubiKey as unique to that user.

Security

Yubico has taken several steps to prevent attempts at retrieving sensitive data (such as the HMAC key). The hardware and firmware is designed to disallow attempts to retrieve such write-only data via communication with the device. Furthermore, the cryptographic routines have been written to minimize physical information leakage, reducing the viability of side-channel attacks like power and electromagnetic analysis. Finally, while YubiKeys were not specifically designed to resist physical attack, it is improbable that data may be retrieved by physical deconstruction of the hardware without also destroying the memory [28].

Software Interaction

Regardless of the feature used, client applications need not talk directly with the YubiKey for authentication data. Yubico's YubiKey Authenticator program is available on their website (and in standard system repositories). Together with their YubiKey Personalization package, any software can query YubiKeys for OTP, OATH, and HMAC Challenge-Response tokens by executing the included `ykchalresp` binary. All of this software is available for the Windows, Mac OS X, and Linux operating systems.

2.7 Secure Programming

Randomness

Many cryptographic operations require some amount of random input. Common variables like keys, IVs, and salts should be unpredictable to an adversary; if they are not, then any security guarantees for the algorithm in use are no longer valid. PassMan always randomly generates such data prior to encrypting the user's entries file to avoid this pitfall. A naive solution may be

to use a pseudo-random number generation algorithm that has been seeded with some dynamic value, such as the current system time. This is far from ideal however, as an adversary could potentially calculate the number generator's state when it was used to produce sensitive data, then recover that data. Even without knowledge of the generator's state, many pseudo-random algorithms are vulnerable to statistical prediction, meaning that keys generated with them are also (to an extent) predictable.

The solution to these problems is a cryptographically secure pseudo-random number generator (CSPRNG). These algorithms differ from other number generators in that they possess additional mathematical qualities, including

- The "next-bit" test: No polynomial-time algorithm should exist that, when given the first k bits of a random sequence, can predict the $k+1$ bit with probability $p > 0.5$.
- Withstanding "state compromise": All of the generator's state should be revealable without it being possible to calculate the random numbers leading up to the compromise.

Many well-vetted CSPRNGs are available in cryptographic software packages, including Crypto++. In practice, these generators consume entropy (highly-random data) provided by the operating system. Under Linux, entropy is commonly sourced from the kernel via the special files `/dev/random` (for blocking calls) or `/dev/urandom` (non-blocking calls); because the OS is limited in the amount of high-quality entropy it can retrieve from the actual machine hardware, many CSPRNGs "stretch" this entropy into longer bit sequences [5]. With this process, it is considerably harder for an adversary to manipulate the number generator (or deduce sensitive data sourced from it) because the entropy is sourced from a combination of complex and ideally nondeterministic machine factors. In fact, the Linux kernel sources entropy from inputs like keyboard key-press timings, time between hardware interrupts, and disk seek times [18].

Sensitive Data

Within most operating systems, memory owned by any process may be reused later by another process with the old data still resident in memory. In this scenario, the offending process has

likely allocated space on the heap with a call to `malloc()`, or instantiated objects there with the `new` operator. If the process exits without cleansing (by setting each byte of allocated memory to `0`), then sensitive variables like keys or cleartext could be recovered by another process [26]. Assuming an adversary had normal user-level privileges, a naive attack could involve sending a well-timed kill signal to the target process, followed by memory allocation calls to inspect memory for sensitive data. While signal handlers could be used to ensure cleansing of memory following a `SIGSTOP` or `SIGINT`, `SIGKILL` cannot be prevented.

To mitigate this issue, all sensitive variables should be cleansed prior to going out of scope. Before any call to `free()` or deconstruction with `delete`, loops may be used to zero-out the memory range. Ultimately, sensitive variables should remain in memory only while they are needed.

Crypto++ Library

The Crypto++ library (also known as `libcryptopp` or `libcrypto++`) is a free and open source library containing common cryptographic tools and primitives, including the algorithms mentioned before. It is written in C++, supports cross-platform usage, and is released under the Boost Software License (although much of the code is in the public domain). Some of the complete implementations offered include:

- Cryptographically secure pseudo-random number generators
- Symmetric ciphers like AES, Serpent, Twofish
- Block cipher modes of operations like ECB, CBC, GCM
- Message authentication codes like HMAC, CMAC
- Cryptographic hash functions like SHA, RIPEMD
- Key derivation functions like PBKDF2
- Asymmetric ciphers like RSA
- Elliptic-curve algorithms like ECDSA

Many algorithms in the library have been favorably benchmarked, although Crypto++ is far from the fastest library overall [4]. Crypto++ does offer accelerated AES with assembly routines using AES-NI. The NIST has validated the library under FIPS 140-2; no security issues were discovered [23].

Chapter 3

Related Work

3.1 KeePass and KeePassX

KeePass is an open-source GUI-based password manager written in C# (C++ for earlier versions) for the Windows operating system. It is released under the GNU General Public License (GPL) 2+. Much like PassMan, KeePass stores user account names, usernames, passwords, and notes in an encrypted file, albeit with different file extensions (`.kdb` or `.kdbx`). As opposed to cloud-based services like LastPass, PassMan and KeePass simply store these files on the user's local storage, bringing increased security at the expense of portability.

A myriad of plugins have been written for KeePass, all of which expand on the manager's interoperability with browsers and other password manager files, among other things. As such, KeePass can import and export account data from over 30 different password managers [19]. KeePass offers hierarchical organization of accounts, and tracks metadata such as password creation and expiration dates. Auto-type functionality is included, using the standard Windows clipboard. A password generator is built-in that can be further randomized by adding entropy from keyboard and mouse inputs. Interestingly, this added-entropy feature is built into the program, meaning that it manages the sourcing of additional entropy by itself, instead of wholly relying on the operating system to provide it.

In terms of security, KeePass ensures confidentiality with the same AES-256 cipher that PassMan relies on, although the Twofish and ChaCha20 ciphers are also available [13]. While PassMan uses Galois/Counter Mode (GCM) to assure integrity with authenticated encryption

(AE) of the file, KeePass uses Cipher Block Chaining (CBC) mode, which by itself, does *not* provide AE. To add this assurance, it uses an Encrypt-then-MAC scheme with HMAC-SHA256.

The YubiKey is supported for use as a second authentication factor as well, although PassMan and KeePass accomplish this differently: Using the YubiKey's Challenge-Response feature, PassMan relies on the cryptographically unique HMAC-SHA1 response to a random challenge as the second factor. LastPass supports the YubiKey's OATH-HOTP feature to produce the second factor [14]. PassMan uses PBKDF2-SHA512 for key derivation, while KeePass uses Argon2; both solutions suitably increase the work required to brute-force a master key and stifle rainbow-table attacks, although Argon2 is arguably better due to its enhanced memory requirements.

KeePassX started as a Linux port of KeePass, offering many of the same features. Modern versions are now cross-platform compatible with use of the Qt 4.8 libraries, making KeePassX available on Linux, Mac OS X, and Windows. Functionally, KeePassX does not differ much from KeePass, although the user interface has diverged significantly, which can be seen in Figure 3.1. Another key exception is how YubiKeys are utilized as a second authentication factor, where KeePassX uses the HMAC-SHA1 Challenge-Response feature instead of OATH-HOTP.

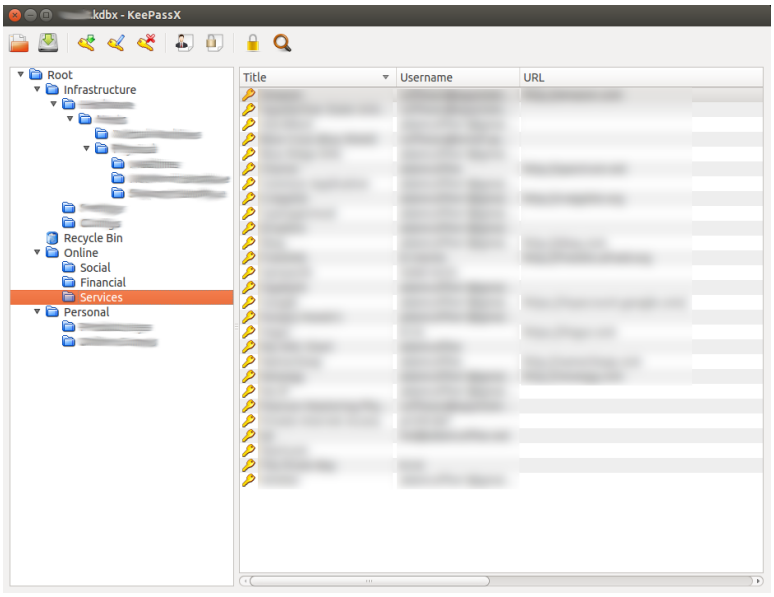


Figure 3.1: KeePassX main window

3.2 LastPass

LastPass is a closed-source password management service owned by LogMeIn that extends the functionality offered by password manager programs like KeePass and PassMan to the cloud. It operates as a "freemium" service, meaning that entry-level users can use it for free, but must pay for the full feature set. Instead of providing an executable program to users, LastPass operates solely as a web application accessible from the user's browser, shown in Figure 3.2. Naturally, this requires Internet access, unlike more traditional password managers. Account information is encrypted with AES-256, with the master key being derived through PBKDF-SHA256; the derivation and encryption process is done on the user's local machine (as opposed to within LastPass's servers) so that LastPass is not privy to user data [16]. As with KeePass and PassMan, LastPass supports the YubiKey as a second authentication factor with the YubiKey's OATH-HOTP feature [17].

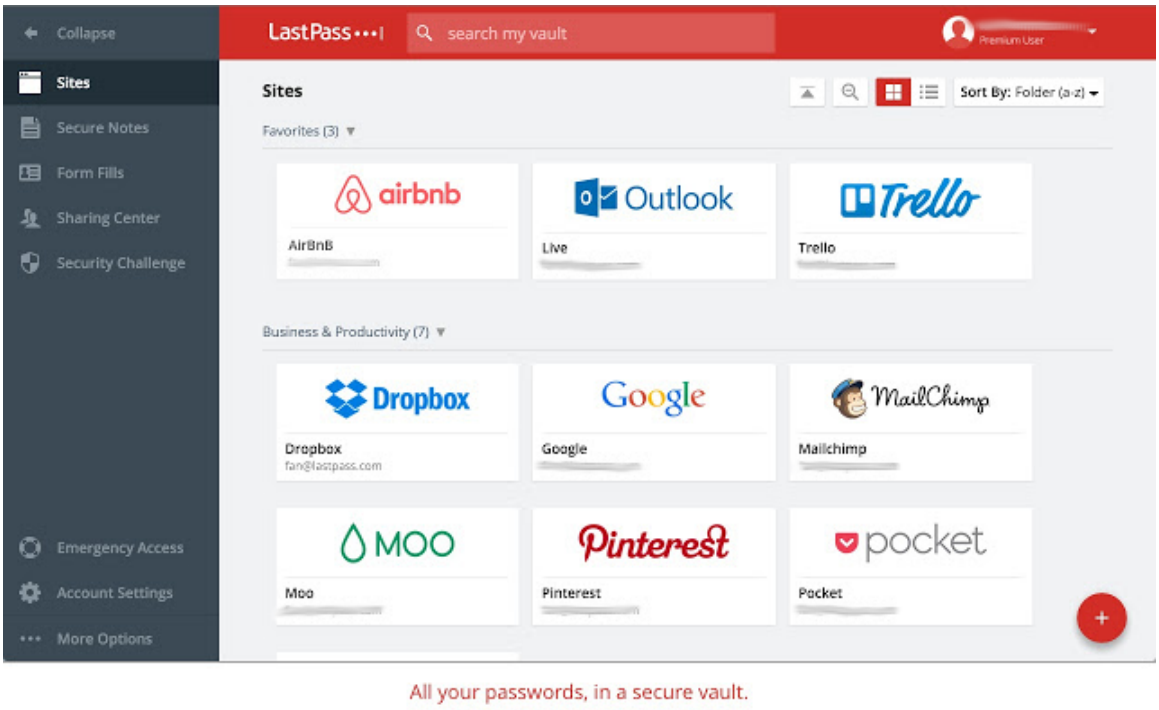


Figure 3.2: LastPass main interface

Chapter 4

How To Use PassMan

4.1 Installation

PassMan must be installed from source, which is available as a compressed archive at `https://github.com/adamcoffee1/PassMan/`. The archive holds an installation script along with the source code for compilation. PassMan is currently compatible with any 32 or 64-bit Linux machine for which Qt, a cross-platform application framework, can be installed. Installation within a Linux environment involves executing the included `install-passman.sh` shell script. All required dependencies will be automatically installed (the root password may be required), after which the program will be compiled using `qmake` and `make` for the user's machine architecture. Dependencies include:

- `xdotool` for auto-type functionality
- `libcrypto++` for cryptographic routines
- `yubikey-personalization` for YubiKey interaction
- `build-essential` and `qt5-default` for program compilation

The resulting binary is written to `/usr/bin/`, and an application configuration file is read by `desktop-file-install` so that the application shows up in the user's desktop manager. Currently, PassMan is configured to compile with Qt 5 which is available on many Linux distributions. After installation, PassMan can be executed from the terminal by typing `PassMan` or selected in the list of applications from the desktop manager. It is identifiable by the icon in Figure 4.1.

Figure 4.1: PassMan icon

4.2 Security

PassMan stores account information including usernames, passwords, and associated notes in an encrypted entries file on local storage. It uses two authentication factors to encrypt the file:

- Master password (something the user *knows*)
- YubiKey HMAC response (something the user *has*)

Because the master password is now the only password that must be remembered to access all accounts, users should make this password strong. For maximal security it should be very long and consist of lowercase and uppercase letters, numbers, and other keyboard symbols [21].

YubiKey Configuration

The YubiKey comes with two configuration slots, where each may be set up to respond in a certain manner. One of these slots needs to be set for Challenge-Response mode via Yubico's YubiKey Personalization Tool. This program is available in the `yubikey-personalization-gui` package, found in most standard repositories. The secret key should be generated as in Figure 4.2 to uniquely identify the YubiKey [31].

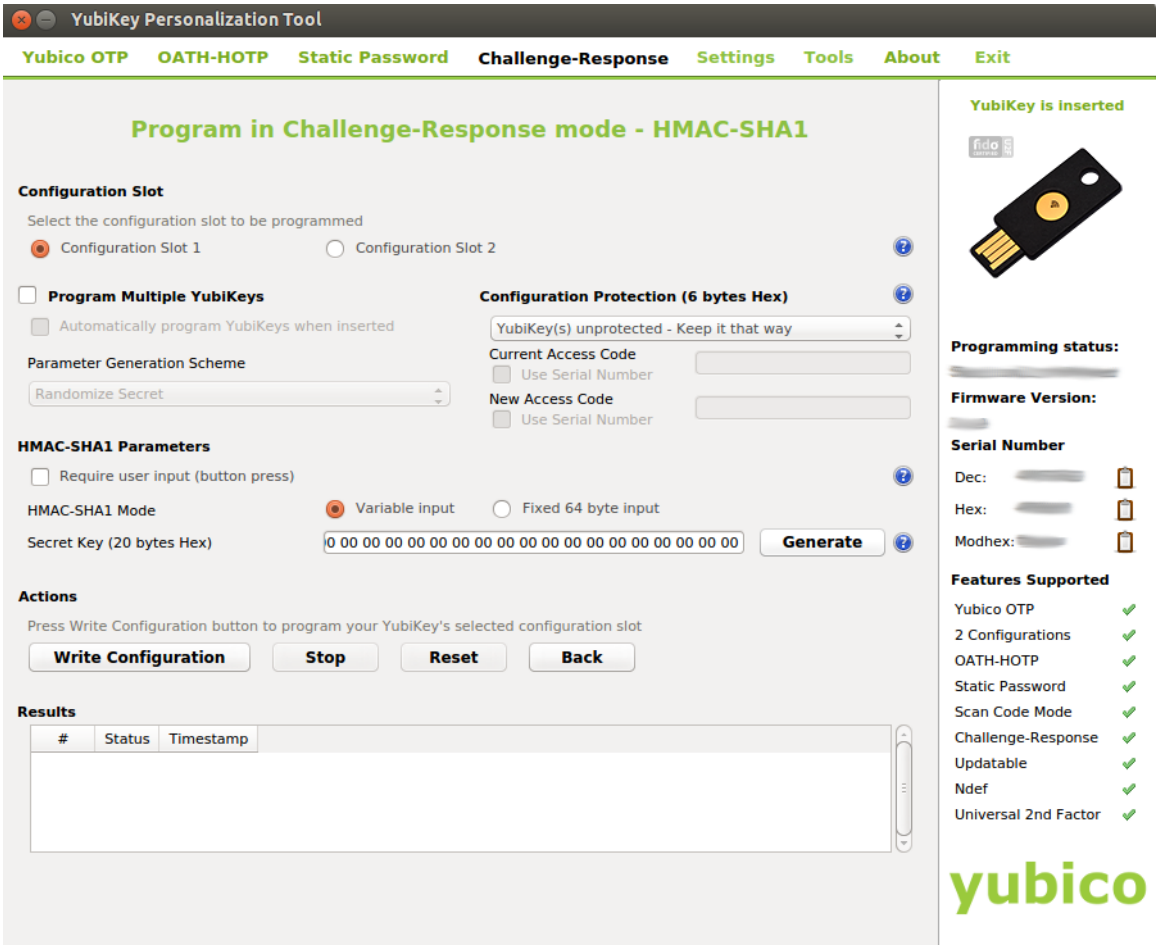


Figure 4.2: YubiKey slot personalization for challenge-response

Once this configuration is written, the YubiKey is ready to be used as a second factor unique to the user. Be sure to remember which configuration slot was programmed (1 or 2), as this needs to be selected during authentication (when an entries file is being decrypted or encrypted).

4.3 Entries File Creation

The first task at hand after installation is to import existing accounts into a PassMan entries file. First, an empty entries file must be created by selecting *File* → *New Database* in the menu bar. For each account to be entered into the file, selecting *Entries* → *Add Entry* causes a new empty entry to be created. The username, password, and notes textboxes correspond with the currently selected account in the entry list. In Figure 4.3, a new entry is being edited.

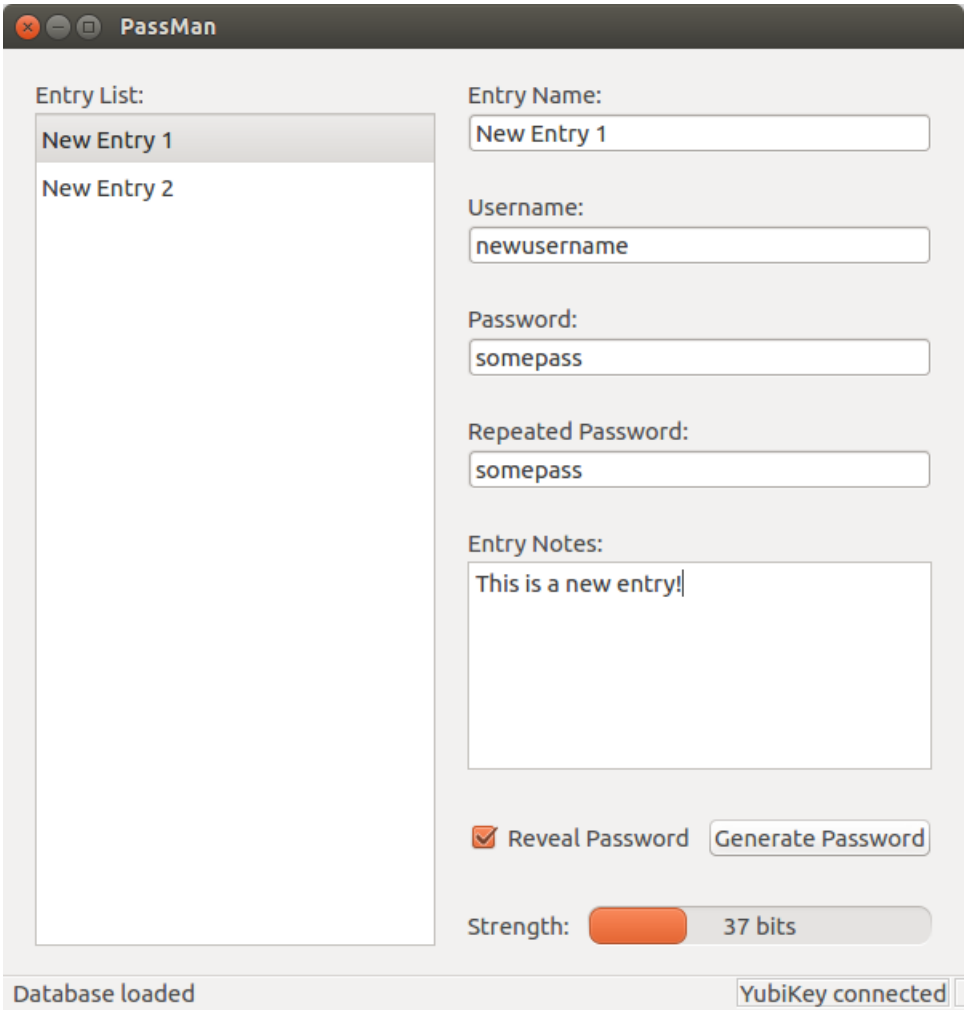


Figure 4.3: PassMan database editing interface

Saving

To save the entries file, select *File* → *Save Database* to initiate the encryption process. An authentication window will show as in Figure 4.4, where the desired master password is entered. Ensure that the YubiKey is connected to the machine (its status is shown in the lower-right corner of many PassMan windows). After entering the password, pressing *Return* or the *Initiate Challenge* button will kick off an HMAC-SHA1 challenge of the YubiKey to obtain the second encryption factor. The file is then encrypted with the derived master key and committed to storage.

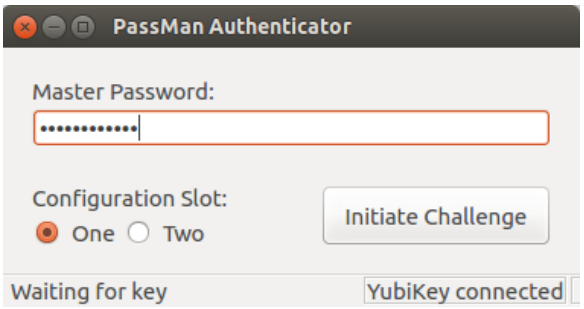


Figure 4.4: Authenticator interface

4.4 Password Generation

As the responsibility for generating and remembering account passwords is now PassMan's alone, the user is free to utilize strong and unique passwords for each and every account. The *Generate Password* button in the main window will bring up a dialog for password generation; it is also accessible by selecting *Tools* → *Password Generator*.

The generator is customizable to improve conformity with the widely varying password policies of various services. For example, many antiquated services disallow special symbols to be used in passwords (thereby reducing password strength potential). In this case the user would simply deselect the symbol character type, causing the generation of a fresh random password without special symbols. Sliding the length bar as in Figure 4.5 to the right will increase the number of randomly generated characters, and clicking the *Generate* button will trigger the creation of a new password if the user is not satisfied with the current one.

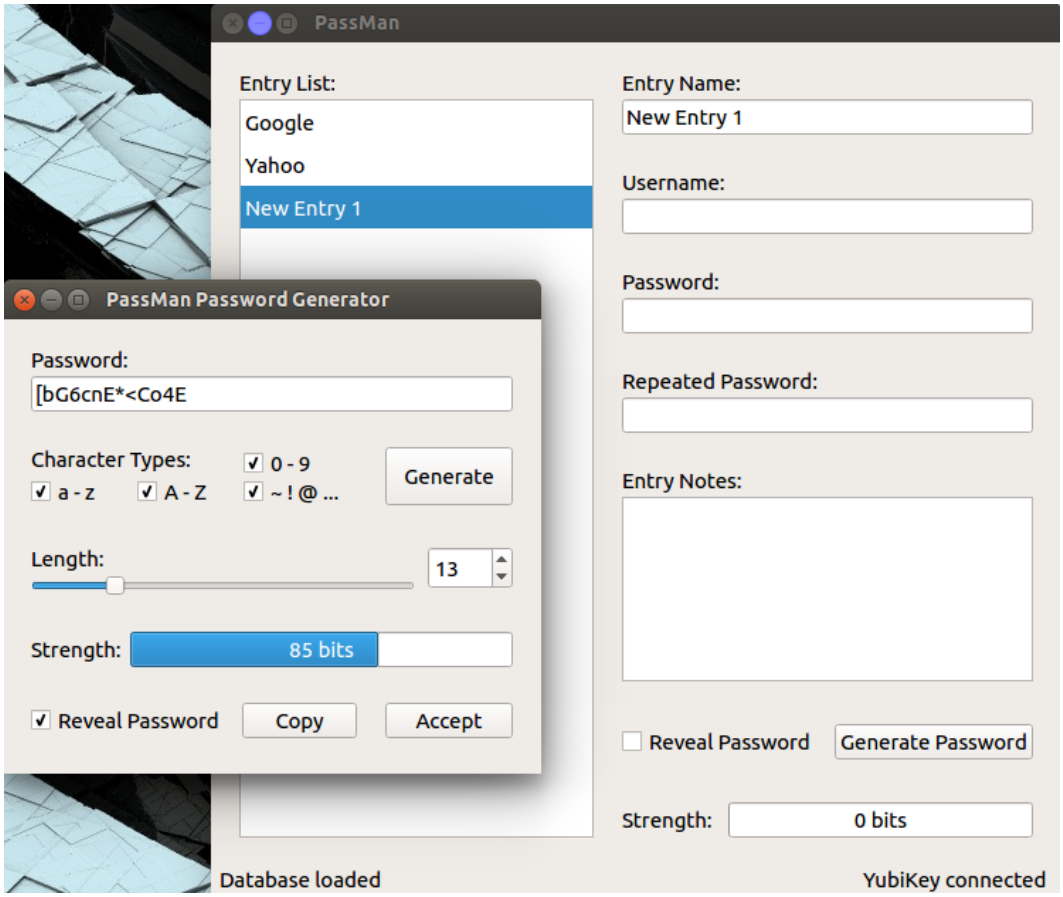


Figure 4.5: PassMan password generation interface

4.5 Account Management

With an account entry selected, the associated username and password may be auto-typed into a login form in another window (such as a web browser) by selecting *Entries* → *Auto-Type Entry*. Window focus will automatically switch to the previously selected application, and the information typed in. Finally, a *Return* keypress is simulated to finish the process.

Entry usernames and passwords can also be manually copied from the *Entries* menu bar option. Many menu bar operations are also associated with keyboard shortcuts for user convenience. For example, pressing *Delete* will cause the currently selected entry to be removed from the database.

Password strength is always shown in the bottom right of the main window to identify any accounts with weak passwords. Note that PassMan requires an account password to be typed twice to avoid the saving of mistyped passwords. If the password and repeated-password textboxes differ in content, the latter textbox will be highlighted in yellow until the mismatch is corrected.

4.6 Diagnostics

There are a couple diagnostic windows included that are not encountered with normal usage of the program. The first, shown in Figure 4.6, is the YubiKey Tester, available from *Tools* → *YubiKey Tester*. This dialog is useful for testing the HMAC-SHA1 challenge-response mechanism. After the configuration slot programmed for HMAC is selected, a challenge may be entered in the textbox. By pressing *Return* or clicking *Initiate Challenge*, PassMan will communicate with the YubiKey and ask for a response to the challenge. The response is shown in hexadecimal, along with the serial number and version of the YubiKey. Any errors are shown in the lower left corner of the dialog.

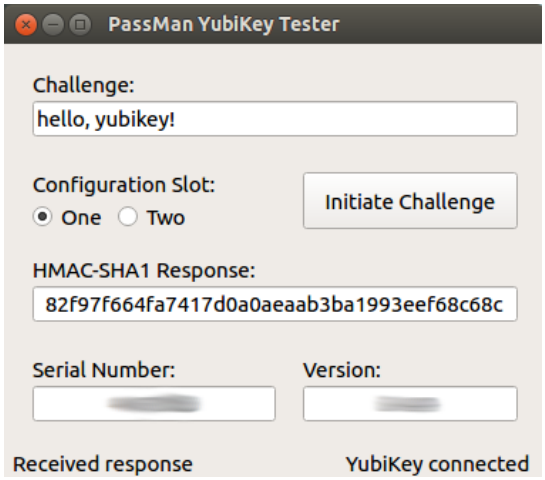
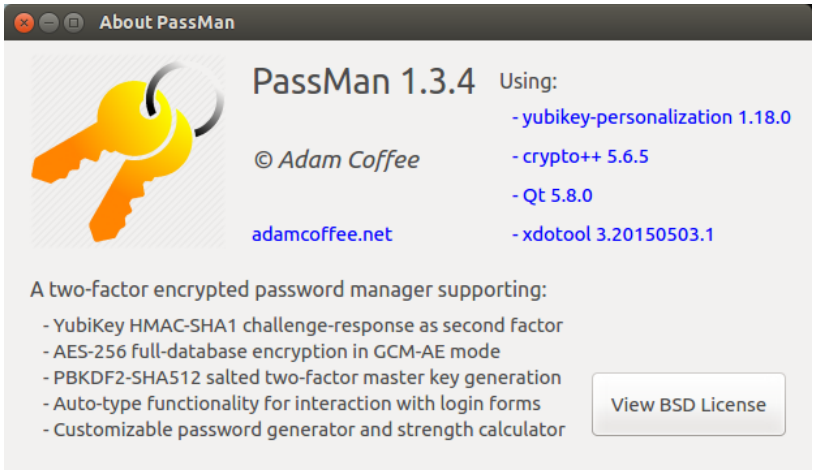


Figure 4.6: PassMan YubiKey testing interface

Additionally, a simple dialog to test password strength is available from *Tools* → *Password Strength Calculator*. There, the user can type in passwords that they come up with to learn their strength.

Chapter 5

PassMan Implementation Details



5.1 Development Framework

Qt (pronounced "cute") is an application framework for cross-platform development that uses standard C++ or Python programming languages [2]. Currently developed by The Qt Company, it is available under both commercial and open-source GNU Lesser General Public License (LGPL) 3.0 licenses. In the case of PassMan, the C++ version was used due to familiarity with the language, efficiency, and interoperability with external libraries. Figure 5.1 shows where Qt sits in a typical Linux ecosystem.

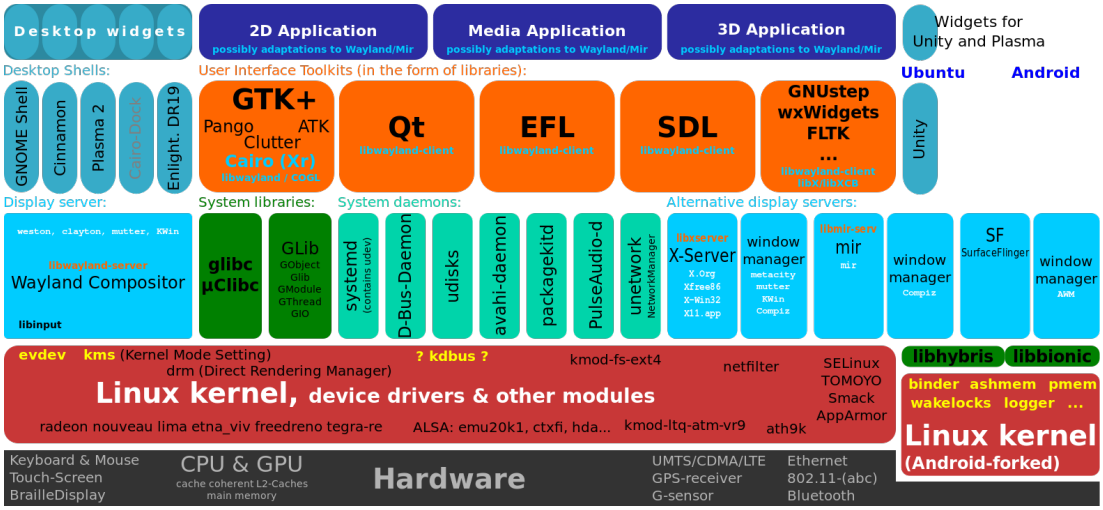


Figure 5.1: Qt framework in Linux-based systems

Some keys traits of Qt include its strong abstraction of the graphical user interface (GUI) and straightforward development environment called Qt Creator, which can be seen in Figure 5.2. It fully integrates debugging, project building, and GUI design. This significantly accelerated the development process and led to an uncomplicated graphical design.

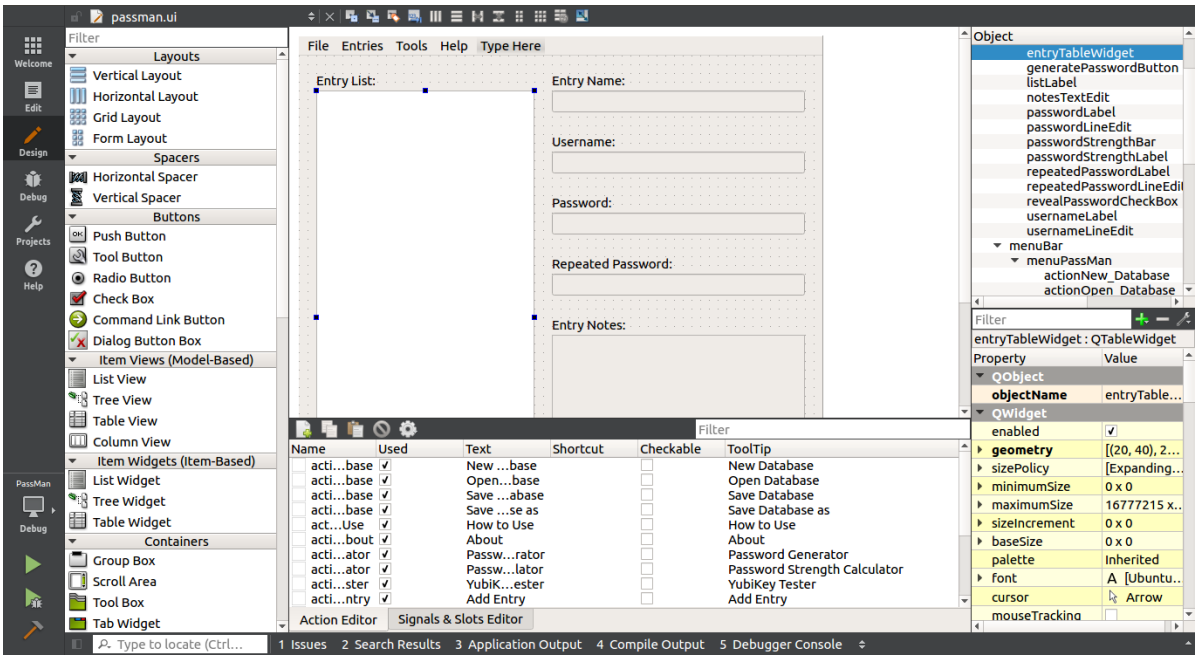


Figure 5.2: Qt Creator editing a user interface

5.2 Software Architecture

Object-oriented design was used to segregate program functionality into various classes. While initial execution starts in a `main()` function as usual, there is actually very little code within that function; the GUI and other peripheral objects are initialized in the `PassMan` class constructor, called in `main()`. This class responds to all user interaction and manages high-level operations and objects.

In general, every class is instantiated once by `PassMan`, and exists on the heap for the life of the program. The `Database` object serves to encapsulate the internal structure of account entries, with some convenience functions for common operations. One exception is `Entry` objects, which may be created and deleted often by the `Database` as the user manipulates their account list.

In the case of user interface classes such as the password `Generator` that may be opened or closed often, their `show()` function is called as needed instead of reinstantiating. This saves memory while improving program performance.

The `Authenticator` class encapsulates all routines involving cryptographic operations, while the `YubiKey` class handles communication and diagnostics with the YubiKey device.

The UML diagram in Figure 5.3 shows the relationships between classes used in PassMan.

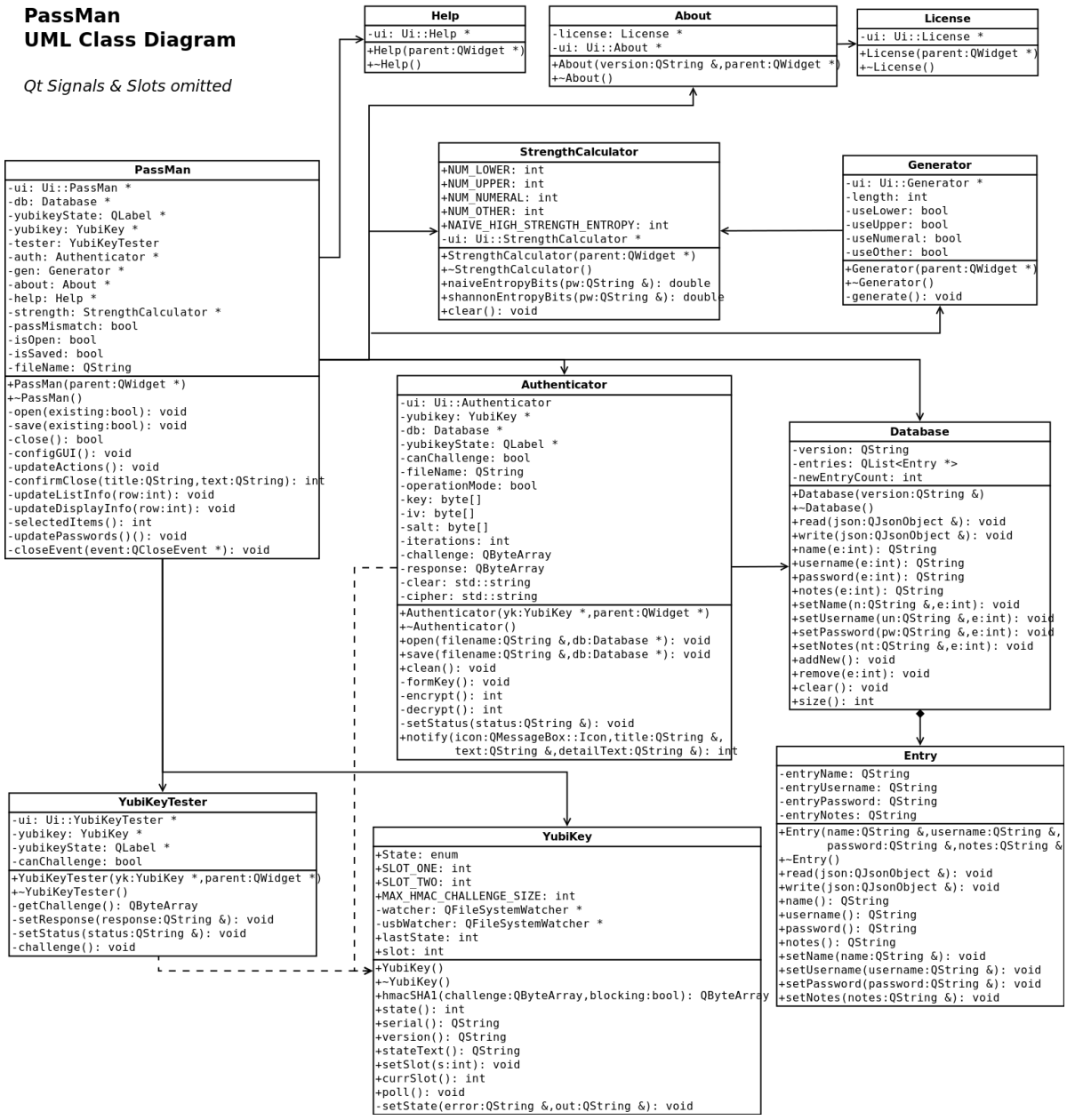


Figure 5.3: PassMan UML class diagram

Observer Pattern Design

Qt offers a language construct known as "Signals and Slots" to facilitate communication of events and data between GUI elements and the core implementation [3]. This observer pattern design is used ubiquitously in PassMan to handle asynchronous events, and avoids excessive boilerplate code. Most commonly, an observable element such as a textbox (a QLineEdit object) can trigger execution of the slot function of its corresponding class, as in Figure 5.4 and Listing 5.1. In this example, the `textEdited()` slot of the password textbox is being chosen in Qt Creator. This textbox resides within the `PassMan` main window class, thus the code to be executed when that textbox is edited is stored there as well.

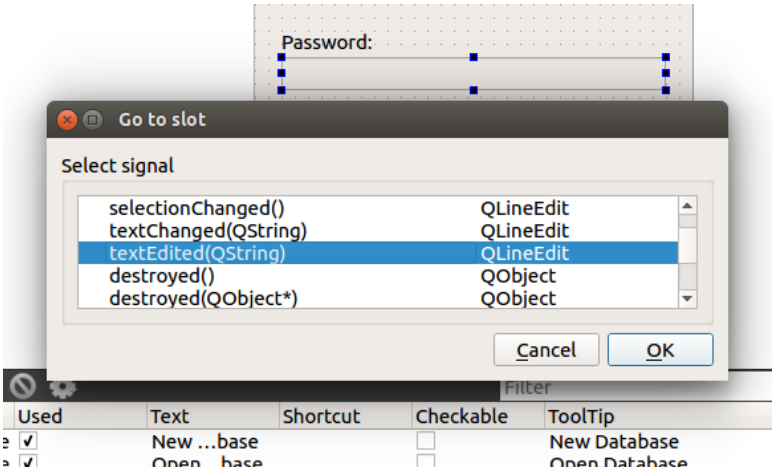


Figure 5.4: Slot selection in Qt Creator for an observable textbox

Listing 5.1: Slot function corresponding to the password textbox

```
1 void PassMan::on_passwordLineEdit_textEdited(const QString &arg1)   // Update entry
      password if changed
2 {
3     int strength = StrengthCalculator::naiveEntropyBits(ui->passwordLineEdit->text
          ());
4     if (ui->passwordStrengthBar->maximum() < strength) ui->passwordStrengthBar->
          setMaximum(strength);
5     ui->passwordStrengthBar->setValue(strength);
6     updatePasswords();
7 }
```

This approach is also used to connect custom classes with custom signals. For example, line 6 of Listing 5.2 shows how the main `PassMan` class is configured in its constructor to receive notice of the insertion or removal of YubiKeys through Qt's `connect()` function. Here, the first argument is a pointer to the object that offers the desired signal, in this case an object of the `YubiKey` class. The second argument specifies which signal of that class to choose; the `yubiKeyChanged()` signal is desired. The third and fourth arguments are a pointer to the receiving object (the `PassMan` object itself), and the member function to be called when the signal is emitted.

Listing 5.2: PassMan class constructor textbox

```
1  PassMan::PassMan(QWidget *parent) : QMainWindow(parent), ui(new Ui::PassMan)
2  {
3      db = new Database(VERSION);
4      yubikey = new YubiKey();
5      gen = new Generator();
6      connect(yubikey, SIGNAL(yubiKeyChanged()), this, SLOT(updateStatusInfo()));
7      connect(db, SIGNAL(readNewData()), this, SLOT(fileReadDone()));
8      connect(db, SIGNAL(writeNewData()), this, SLOT(fileWriteDone()));
9      connect(gen, SIGNAL(passwordGenerated()), this, SLOT(passGenDone()));
10     tester = new YubiKeyTester(yubikey);
11     auth = new Authenticator(yubikey);
12     about = new About(VERSION);
13     help = new Help();
14     strength = new StrengthCalculator();
15     passMismatch = false;
16     isSaved = true;
17     isOpen = false;
18     configGUI();
19 }
```

5.3 Database Management

The `Database` class holds a `QList` (Qt's linked-list class) of `Entry` pointers in order to keep track of each account; these entries each hold a `QString` (Qt's native string class) for the name, username, password, and potential notes corresponding to the account. To facilitate the confidential storage of account data, the `write()` and `read()` functions are used to serialize or deserialize, respectively, the `Entry` objects via JavaScript Object Notation (JSON). In the case of writing during a save operation, the resulting JSON string would be passed through the encryption process prior to being written out to storage. For reading during an open operation,

the file is decrypted and then interpreted into actual objects. Listing 5.3 and 5.4 show the read and write functions for both.

Listing 5.3: Database serialization functions

```
1  void Database::read(const QJsonObject &json) // Extracts entry information from
       JSON object
2  {
3      entries.clear();
4      QJsonArray entryArray = json.value(ENTRIES_KEY).toArray();
5      for (int i = 0; i < entryArray.size(); i++)
6      {
7          QJsonObject entryObj = entryArray.at(i).toObject();
8          entries.append(new Entry(entryObj.value(NAME_KEY).toString(), entryObj.
               value(USERNAME_KEY).toString(),
9                                   entryObj.value(PASSWORD_KEY).toString(), entryObj.
                                       value(NOTES_KEY).toString()));
10     }
11     version = json.value(VERSION_KEY).toString();
12     emit readNewData(); // Notify watchers that database is loaded
13 }
14
15 void Database::write(QJsonObject& json) // Serialize entry information to JSON
       object
16 {
17     QJsonArray entryArray;
18     foreach (Entry* e, entries)
19     {
20         QJsonObject entryObj;
21         e->write(entryObj);
22         entryArray.append(entryObj);
23     }
24     json.insert(ENTRIES_KEY, entryArray);
25     json.insert(VERSION_KEY, version);
26     emit writeNewData();    // Notify watchers that database saved
27 }
```

The `QJsonObject` argument is a dictionary structure holding serialized objects; it contains an array of all entries, and a version string. During a read operation, this object will have been read from the entries file after decryption by the `Authenticator` class, then passed off to the `Database` class. Each value in the `entryArray` is converted back into an actual `Entry` object, and inserted into the account list for display. Other classes are then notified of the change via a call to `emit readNewData()`. During a write operation, all entries are handled in a similar but reversed manner. `Authenticator` will call the `write()` function of `Database` with an object reference; each entry is serialized, then appended to the `QJsonObject`.

Listing 5.4: Entry serialization functions

```
1  void Entry::read(const QJsonObject& json)
2  {
3      entryName = json.value("name").toString();
4      entryUsername = json.value("username").toString();
5      entryPassword = json.value("password").toString();
6      entryNotes = json.value("notes").toString();
7  }
8
9  void Entry::write(QJsonObject& json) const
10 {
11     json.insert("name", entryName);
12     json.insert("username", entryUsername);
13     json.insert("password", entryPassword);
14     json.insert("notes", entryNotes);
15
16 }
```

5.4 YubiKey Communication

As the second factor in the master key is an HMAC-SHA1 response from the user's YubiKey, this response must be acquired from the YubiKey prior to deriving the key. The `YubiKey` class takes care of these lower-level operations such that the `Authenticator` and `YubiKeyTester` classes can query it when needed.

Instead of directly talking with the YubiKey (a nontrivial, platform-dependent process), binaries from the `yubikey-personalization` package (available in standard system repositories) are used to initiate HMAC challenges and obtain diagnostic information. Specifically, the `ykchalresp` and `ykinfo` programs are executed within separate processes, and communicated with via their standard input (STDIN), output (STDOUT), and error (STDERR) streams.

Listing 5.5 shows how a `QProcess` is used to accomplish this programmatically. Starting at line 4, pre-formed commands are selected based on the user's slot choice. The HMAC challenge is then written to the standard input of the newly formed process on line 5. A blocking call is made to wait while `ykchalresp` queries the YubiKey. This is important because the YubiKey may be programmed to require a button-press prior to transmitting an HMAC response, something that could take the user a few seconds to complete.

Listing 5.5: YubiKey HMAC-SHA1 challenge function

```
1  QByteArray YubiKey::hmacSHA1(const QByteArray& challenge, bool blocking)    //
       Complete an HMAC-SHA1 challenge-response
2  {
3      QProcess* proc = new QProcess();    // Will run Yubico software in separate
           process
4      proc->start(slot == 1 ? HMAC_SLOT_1_COMMAND : HMAC_SLOT_2_COMMAND, QIODevice::
           ReadWrite);
5      proc->write(challenge.toHex()); // Send challenge via standard input
6      proc->closeWriteChannel();
7      if (blocking) proc->waitForFinished(-1);    // YubiKey may require button-press
           , wait if caller desired
8      QString error(proc->readAllStandardError());
9      QString out(proc->readAllStandardOutput());
10     setState(error, out);
11     proc->close();
12     delete proc;
13     return out.left(out.length() - 1).toUtf8(); // Strip newline
14 }
```

Because the YubiKey is a USB device that may be removed or inserted at any time, it is important for PassMan to be aware of these changing states. A `QFileSystemWatcher` is used to signal changes to the `/dev/usb/` directory, which lists certain connected Universal Serial Bus (USB) devices. Upon any change, `lsusb` is executed in a separate process to search for YubiKeys, shown in Listing 5.6. The output is checked to see if it contains the proper string identifier for a YubiKey on line 7. If this is the case, the YubiKey is queried for diagnostic information with `poll()`, to further confirm that it exists. A signal is emitted to notify other classes of the change.

Listing 5.6: YubiKey detection function

```
1  void YubiKey::usbChange()   // Check if USB change was a YubiKey change
2  {
3      QProcess* proc = new QProcess();    // Will run listing program in separate
           process
4      proc->start(GET_USB_COMMAND, QIODevice::ReadWrite);
5      proc->waitForFinished(-1);
6      QString out(proc->readAllStandardOutput());
7      if (out.contains(USB_NAME)) this->poll();
8      else lastState = NOT_PRESENT;
9      proc->close();
10     delete proc;
11     emit yubiKeyChanged();  // Notify watchers that a change has occured
12 }
```

Ultimately, this brings increased convenience for the user, as they may be notified of the YubiKey's status in program windows. It should be noted that this is a Linux-specific solution to detection, and would need to be generalized for cross-platform compatibility.

5.5 Authentication and Confidentiality

To ensure the secure storage of user account information, the entire serialized entries file is passed through an encryption process prior to being committed to storage. All secure operations take place within the `Authenticator` class, which only provides `open()`, `save()`, and `clean()` functions for `PassMan` to call. All variables relevant to cryptographic operations are held within this class, including

- 256-bit key for the block cipher (unsigned character array)
- 2048-bit initialization vector (IV) for the block cipher mode (unsigned character array)
- 128-bit salt for key derivation (unsigned character array)
- Iteration count for key derivation (signed integer)
- Challenge and response arrays for the YubiKey HMAC second authentication factor (QByteArray)
- Cleartext and ciphertext strings (standard library string)

While all of these variables are treated as sensitive during the program's execution, in reality only the cipher key, YubiKey HMAC response, and cleartext string are *critically* sensitive. The IV, salt, iteration count, YubiKey HMAC challenge, and ciphertext may be treated as common knowledge, something an adversary could possess while still being unable to reproduce the database cleartext. Thus, these values are stored in the database file for later decryption as in Figure 5.5.

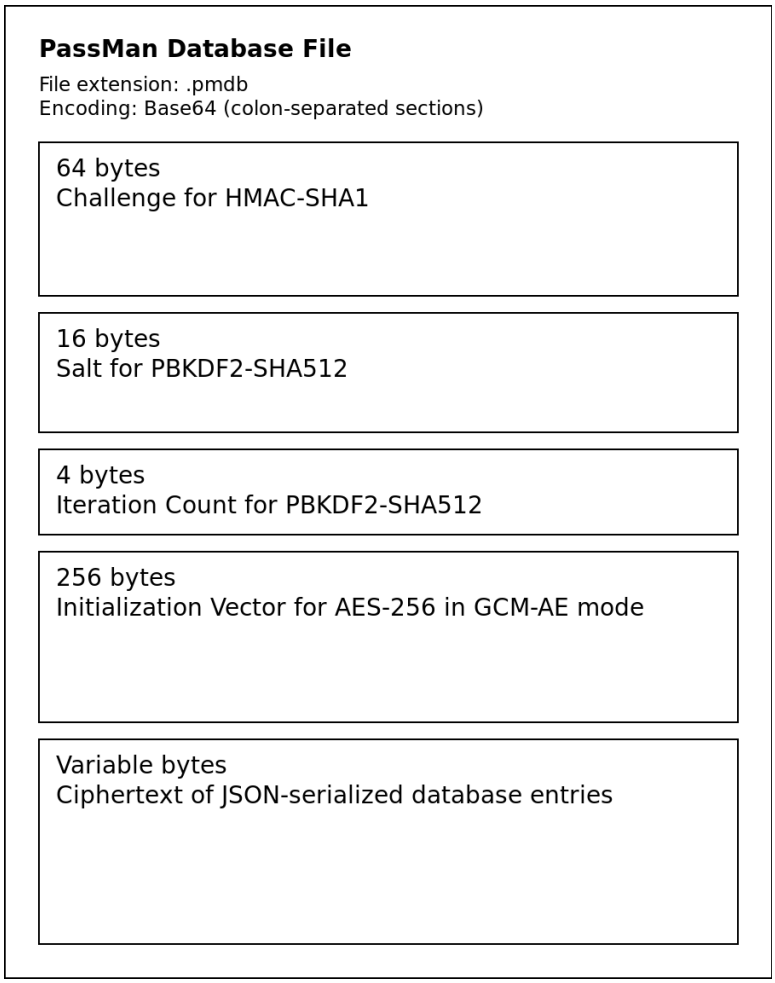


Figure 5.5: PassMan database file specification

All data is encoded in Base64, with each section colon-separated. While Base64 is less space-efficient than storage in binary form, the user benefits from increased portability, as the file is communicable as simple text.

The `clean()` function exists to zero-out these variables after a database file is closed, or prior to program exit; they remain in memory while a file is open for the user's convenience. If at any point during encryption or decryption an exception occurs, these variables are always cleaned to prevent the data from existing in memory after the program dies. Listing 5.7 shows the simple zeroing function.

Listing 5.7: Authenticator clean function

```
1  void Authenticator::clean() // Reset authenticator and wipe any sensitive data
2  {
3      for (int i = 0; i < CryptoPP::AES::MAX_KEYLENGTH; i++) key[i] = 0;
4      for (int i = 0; i < IV_SIZE; i++) iv[i] = 0;
5      for (int i = 0; i < SALT_SIZE; i++) salt[i] = 0;
6      challenge.fill(0);
7      response.fill(0);
8      clear.assign(clear.length(), 0);
9      cipher.assign(cipher.length(), 0);
10 }
```

Cryptographic Tools

The specific block cipher chosen for confidentiality is the Advanced Encryption Standard (AES) specified in FIPS 197, operating with a 256-bit key (the largest typically used). In addition to large key options, AES was chosen because it is a well-researched cipher and is often accelerated in modern processors via an extension to the x86 instruction set [6]. In the case of PassMan however, confidentiality alone is not sufficient: an adversary could still manipulate the entries file by changing portions of it; decryption of this file would likely succeed, but with incorrect cleartext data [7]. To prevent this, AES is used in Galois/Counter Mode (GCM) to provide Authenticated Encryption (GCM-AE). In the previous scenario, decryption would instead fail with an authentication exception, allowing PassMan to then notify the user of corruption [8].

The master key is generated with Password-based Key Derivation Function 2 (PBKDF2), using the SHA-512 hash function (PBKDF2-SHA512). This process is known as "key stretching," and slows the process of password cracking [9]. The degree of stretching is governed by the number of hash iterations. PassMan uses a non-fixed iteration count, instead aiming for the derivation operation to take 0.5 seconds. Obviously the iteration count will vary depending on many machine factors, but it is guaranteed to be large, effectively raising the amount of

work needed to guess a single password. Since key derivation only occasionally occurs over the lifetime of the program, this delay is afforded in exchange for improved key strength.

Listing 5.8 shows the corresponding code. First, the presence of a YubiKey is confirmed by checking the `canChallenge` boolean value. An attempt is made to retrieve an HMAC response on line 5, then the user is notified of any errors. If none occurred, the response is concatenated with the user's master password on line 15.

From this point forward, key derivation differs depending on whether encryption or decryption is desired. For encryption, a new key must always be used. Starting on line 24, the concatenated key factors, previously-generated salt, and desired derivation time `MIN_PBKDF_TIME` are used to derive the key. Note that the iteration count is returned by the call; this value simply represents the number of SHA-512 iterations that were needed to obtain the desired derivation time. A call to `encrypt()` accomplishes the actual encryption, leaving the ciphertext in the `cipher` member variable.

For decryption, remember that a key has been derived in the past and used to encrypt the file, thus it is necessary to exactly recreate that key for decryption to succeed. On line 19, the two concatenated key factors in the `response` variable, recovered salt, and recovered iteration count are used to derive the key. The call to `decrypt()` on line 20 completes the actual decryption process, leaving the cleartext in the `clear` member variable.

Listing 5.8: Authenticator key derivation function

```
1  void Authenticator::formKey()   // Create master key
2  {
3      if (canChallenge) {
4          setStatus(BUSY_YUBIKEY);
5          response = yubikey->hmacSHA1(challenge, true);
6          yubikeyState->setText(yubikey->stateText());
7          if (yubikey->state() == YubiKey::NOT_PRESENT) {
8              notify(QMessageBox::Warning, ERROR_TITLE, YUBIKEY_ERROR,
                   YUBIKEY_PRESENT_ERROR);
9              return;
10         }
11         else if (yubikey->state() == YubiKey::TIMEOUT) {
12             notify(QMessageBox::Warning, ERROR_TITLE, YUBIKEY_ERROR,
                   YUBIKEY_HMAC_ERROR);
13             return;
14         }
15         response.append(ui->masterPasswordLineEdit->text());
16         setStatus(BUSY_KEY);
17         CryptoPP::PKCS5_PBKDF2_HMAC<CryptoPP::SHA512> kdf;  // Derive master key
               from concatenation of user password and YubiKey response
18         if (operationMode == DECRYPT_MODE) {
```

```
19              kdf.DeriveKey(this->key, sizeof(key), 0, (byte*) response.data(),
                    response.length(), this->salt, sizeof(salt), iterations, 0);  //
                    Use recovered iteration count to derive key
20              if (!decrypt()) return;
21              db->read(QJsonDocument::fromJson(QByteArray::fromStdString(clear)).
                    object());
22          }
23          else {
24              iterations =  kdf.DeriveKey(this->key, sizeof(key), 0, (byte*) response
                    .data(), response.length(), this->salt, sizeof(salt), iterations,
                    MIN_PBKDF_TIME);
25              if (!encrypt()) return;
26              QFile file(fileName);
27              QByteArray iv;
28              for (int i = 0; i < IV_SIZE; i++) iv.append(this->iv[i]);
29              QByteArray salt;
30              for (int i = 0; i < SALT_SIZE; i++) salt.append(this->salt[i]);
31              file.open(QIODevice::WriteOnly);
32              file.write(challenge.toBase64());
33              file.write(":");
34              file.write(salt.toBase64());
35              file.write(":");
36              file.write(QByteArray::number(iterations).toBase64());
37              file.write(":");
38              file.write(iv.toBase64());
39              file.write(":");
40              file.write(QByteArray::fromStdString(cipher).toBase64());
41              file.close();
42          }
43          this->hide(); }
44  }
```

Encryption Process

In Figure 5.6 the complete encryption process can be seen. Note that the HMAC challenge, IV, and salt are randomly generated for each encryption operation. This is critical to the security of the entries file, as it ensures that the same entries cleartext always results in drastically different ciphertext. Furthermore, using a new HMAC challenge each time prevents an adversary with knowledge of a prior HMAC response from reusing it as one of the factors in key derivation.

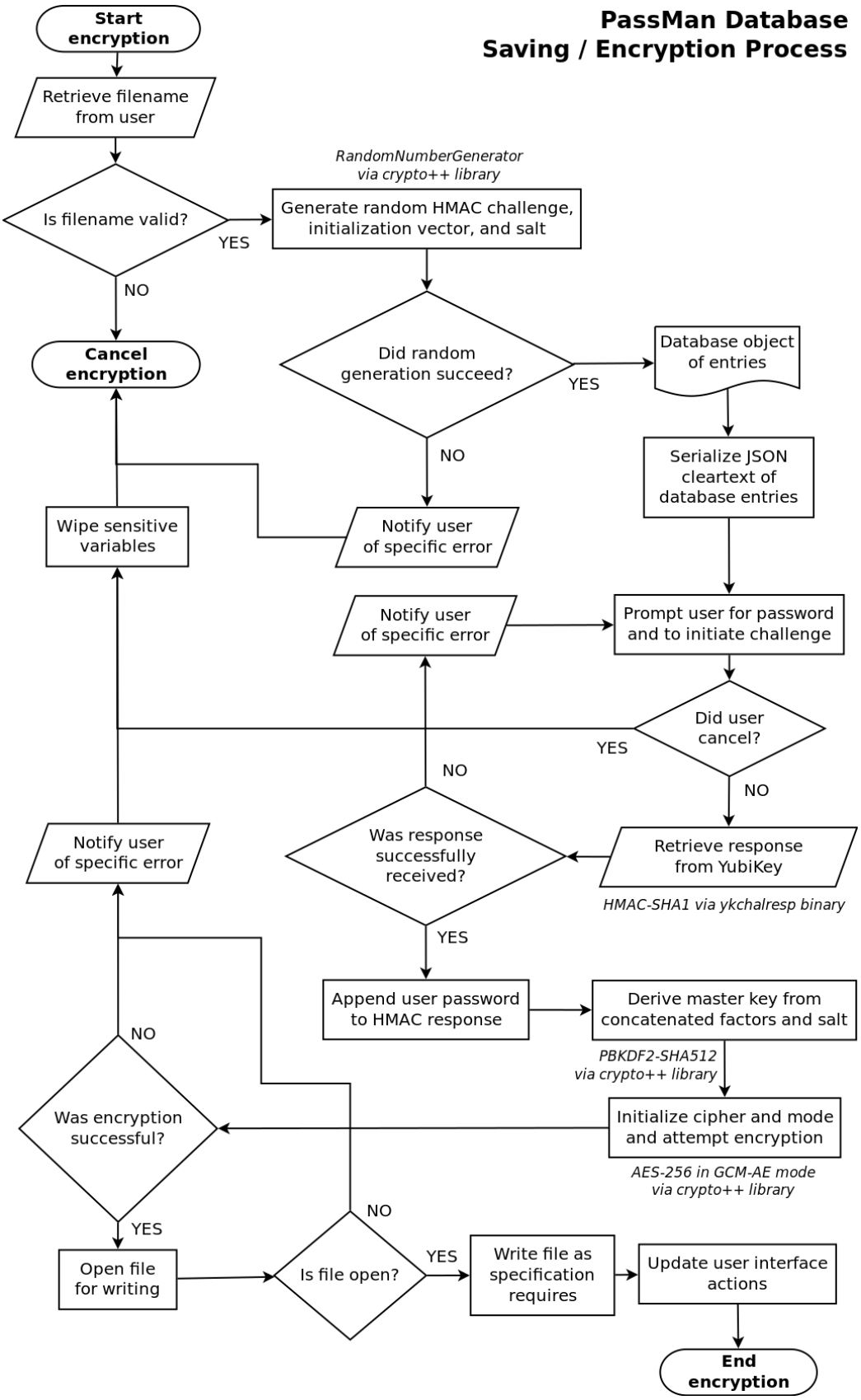


Figure 5.6: PassMan encryption process flowchart

The code for these steps can be seen in Listing 5.9, where the `save()` function receives as arguments the filename and entries list to encrypt. An `AutoSeededRandomPool` is used to source random data offered from the operating system for each variable in lines 8 through 14. Then, the entries are serialized on line 25, and stored in the `clear` string for encryption. After this step, the user is prompted via a dialog (on line 28) to present their master password, which then triggers an HMAC challenge to the YubiKey.

Listing 5.9: Authenticator save function

```
1  void Authenticator::save(const QString& fileName, Database* db) // Encrypt a file
2  {
3      try
4      {
5          operationMode = ENCRYPT_MODE;
6          this->fileName = fileName;
7          this->db = db;
8          byte challenge[YubiKey::MAX_HMAC_CHALLENGE_SIZE];
9          CryptoPP::AutoSeededRandomPool prng;
10         prng.GenerateBlock(challenge, sizeof(challenge));   // Generate new random
               HMAC challenge each time!
11         this->challenge.clear();
12         for (int i = 0; i < YubiKey::MAX_HMAC_CHALLENGE_SIZE; i++) this->challenge.
               append(challenge[i]);
13         prng.GenerateBlock(iv, sizeof(iv)); // Generate new random IV each time!
14         prng.GenerateBlock(salt, sizeof(salt)); // Generate new random salt each
               time!
15     }
16     catch (CryptoPP::Exception& ex) //Catch if challenge and iv generation fail
17     {
18         setStatus(FAILED);
19         notify(QMessageBox::Critical, ERROR_TITLE, ENCRYPT_ERROR, QString(ex.what()
               ));
20         this->clean();
21         this->hide();
22         return;
23     }
24     QJsonObject obj;
25     db->write(obj);
26     QJsonDocument doc(obj);
27     clear = doc.toJson().toStdString();
28     this->show();   // Continue process after user supplies password
29 }
```

Actual encryption does not occur until the user presents their password and initiates a challenge, which triggers the execution of the aforementioned `formKey()` function, which itself calls `encrypt()` (shown in Listing 5.10) after it has derived the key. GCM mode is initialized on line 6 and 7, and actual encryption is attempted on line 8. Here, an `AuthenticatedEncryptionFilter` is used to run the cipher in GCM mode. Upon successful encryption, the execution returns to `formKey()`, where the file is written out to storage, and variables cleaned.

Listing 5.10: Authenticator encryption function

```
1  int Authenticator::encrypt()    // Perform authenticated AES-256 encryption in GCM-
       AE mode
2  {
3      try
4      {
5          cipher.clear();
6          CryptoPP::GCM<CryptoPP::AES>::Encryption enc;
7          enc.SetKeyWithIV(key, sizeof(key), iv, sizeof(iv)); // Initialize cipher
8          CryptoPP::StringSource src(clear, true, new CryptoPP::
               AuthenticatedEncryptionFilter(enc, new CryptoPP::StringSink(cipher),
               false, TAG_SIZE));   // Run cleartext through
9      }
10     catch (CryptoPP::Exception& ex)
11     {
12         setStatus(FAILED);
13         notify(QMessageBox::Critical, ERROR_TITLE, ENCRYPT_ERROR, QString(ex.what()
               ));
14         this->clean();
15         this->hide();
16         return false;
17     }
18     setStatus(COMPLETE);
19     return true;
20 }
```

Decryption Process

Figure 5.7 shows how the decryption process is much the same, except that there are extra checks to ensure that the entries file is valid prior to attempting decryption. This is done to avoid errors during the deserialization process of entries file variables.

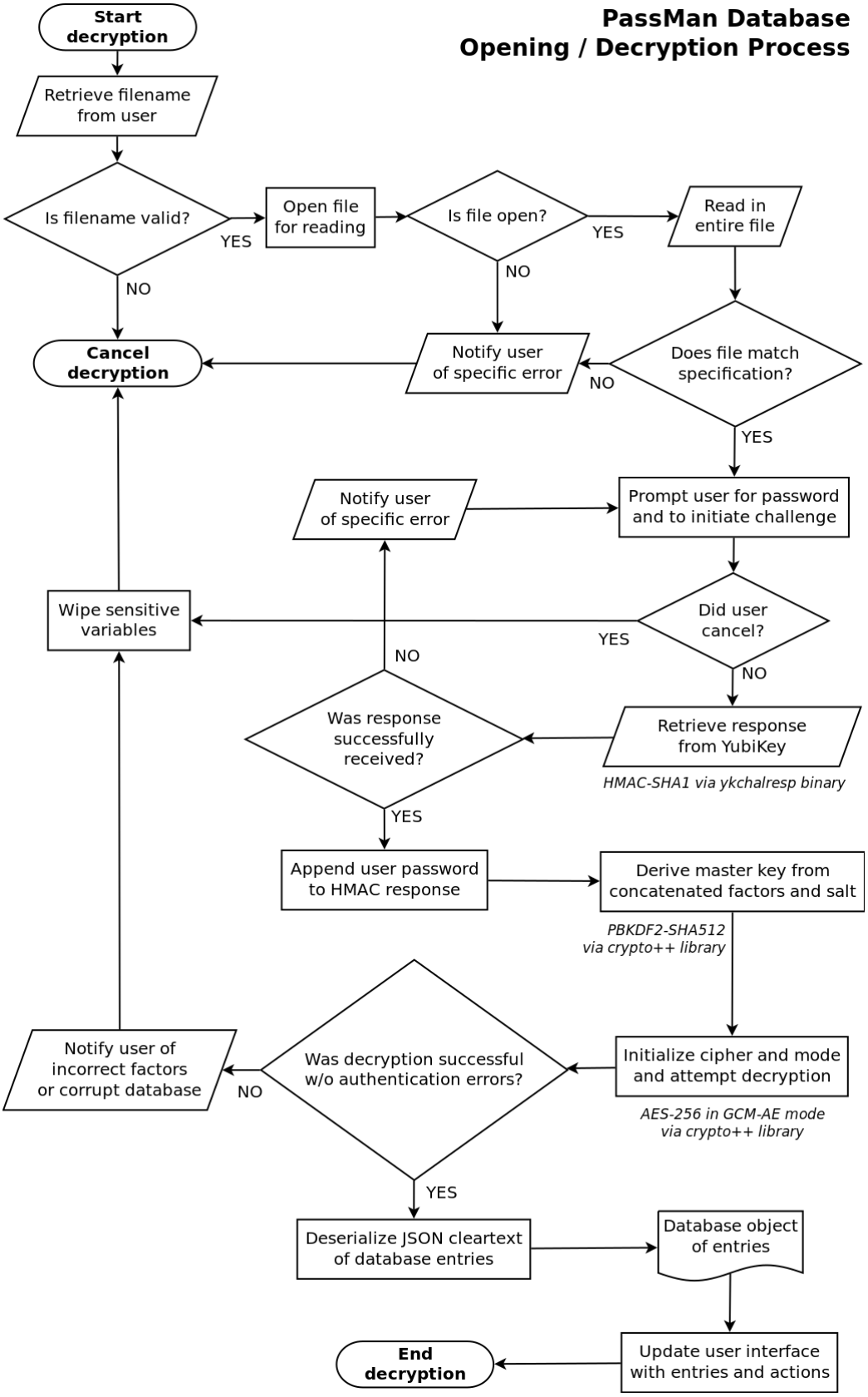


Figure 5.7: PassMan decryption process flowchart

In Listing 5.11, the `open()` function reads in the file, breaking it into pieces on line 15 via `split()`. This follows the colon-separated file specification, which is further confirmed in the call on line 17. If there are no errors, the user is prompted via a dialog to present their master password, then triggering the HMAC challenge.

Listing 5.11: Authenticator open function

```
1  void Authenticator::open(const QString& fileName, Database* db) // Decrypt a file
2  {
3      ui->masterPasswordLineEdit->clear();
4      operationMode = DECRYPT_MODE;
5      this->fileName = fileName;
6      this->db = db;
7      QFile file(fileName);
8      if (!file.open(QIODevice::ReadOnly))    // Failed to open file
9      {
10         notify(QMessageBox::Critical, ERROR_TITLE, DB_ERROR, FILE_ERROR);
11         this->clean();
12         this->hide();
13         return;
14     }
15     QByteArrayList parts = file.readAll().split(FILE_PORTION_SEPARATOR);
16     file.close();
17     if (!ensureParts(parts)) return;
18     this->show();   // Show interface to user
19 }
```

Actual decryption does not occur until the user presents their password and initiates a challenge. Similar to encryption, this causes `formKey()` to be called, which itself calls `decrypt()` (shown in Listing 5.12) after it has derived the key. GCM mode is initialized, and the ciphertext fed through the `AuthenticatedDecryptionFilter`. In this stage, an exception could indicate that the file has been corrupted. In this case, the user would be notified, and the operation aborted.

Listing 5.12: Authenticator decryption function

```
1  int Authenticator::decrypt()    // Perform authenticated AES-256 decryption in GCM-
       AE mode
2  {
3      try
4      {
5          clear.clear();
6          CryptoPP::GCM<CryptoPP::AES>::Decryption dec;
7          dec.SetKeyWithIV(key, sizeof(key), iv, sizeof(iv)); // Initialize cipher
8          CryptoPP::AuthenticatedDecryptionFilter adf(dec, new CryptoPP::StringSink(
               clear), CryptoPP::AuthenticatedDecryptionFilter::DEFAULT_FLAGS,
               TAG_SIZE);    // Initialize authentication filter
9          CryptoPP::StringSource src(cipher, true, new CryptoPP::Redirector(adf));
                  // Redirector feeds cipher into authenticator
10     }
11     catch (CryptoPP::Exception& ex) // Will catch if integrity check fails, or
           other issue
12     {
13         setStatus(FAILED);
14         if (ex.GetErrorType() == CryptoPP::Exception::DATA_INTEGRITY_CHECK_FAILED)
               notify(QMessageBox::Critical, ERROR_TITLE, DECRYPT_ERROR,
               INTEGRITY_ERROR);
15         else    // Some other odd exception
16         {
17             notify(QMessageBox::Warning, ERROR_TITLE, DECRYPT_ERROR, QString(ex.
                   what()));
18             this->clean();
19             this->hide();
20         }
21         return false;
22     }
23     setStatus(COMPLETE);
24     return true;
25 }
```

5.6 Password Generation

When using PassMan, the user need only remember their master password and their YubiKey to access all account passwords. Therefore, they can use unique, high-strength passwords for each account. The `Generator` class provides a customizable interface for creating passwords with desired entropy levels. Listing 5.13 contains the simple generation function.

Listing 5.13: Password generation function

```
1  void Generator::generate()  // Formulate a new password, given set constraints
2  {
3      QString pass;
4      CryptoPP::AutoSeededRandomPool prng;    // Initialize CSPRNG
5      QList<int> selectedTypes;
6      bool containsAllTypes, containsLower, containsUpper, containsNumeral,
           containsOther;
7      containsAllTypes = false;
8      if (useLower) selectedTypes.append(0);
9      if (useUpper) selectedTypes.append(1);
10     if (useNumeral) selectedTypes.append(2);
11     if (useOther) selectedTypes.append(3);
12     if (selectedTypes.length() < 1) return;
13     while (!containsAllTypes)
14     {
15         pass.clear();
16         containsAllTypes = containsLower = containsUpper = containsNumeral =
               containsOther = false;
17         for (int i = 0; i < length; i++)
18         {   // Select a random type, then a random value within that type
19             switch (selectedTypes.at(prng.GenerateByte() % selectedTypes.length()))
20             {
21                 case 0:
22                     pass.append(LOWER.at(prng.GenerateByte() % StrengthCalculator::
                           NUM_LOWER));
23                     break;
24                 case 1:
25                     pass.append(UPPER.at(prng.GenerateByte() % StrengthCalculator::
                           NUM_UPPER));
26                     break;
27                 case 2:
28                     pass.append(NUMERAL.at(prng.GenerateByte() % StrengthCalculator
                           ::NUM_NUMERAL));
29                     break;
30                 case 3:
31                     pass.append(OTHER.at(prng.GenerateByte() % StrengthCalculator::
                           NUM_OTHER));
32                     break;
33             }
34         }
35         for (int i = 0; i < length; i++)    // Ensure all chosen types are
               somewhere in the password
36         {
37             if (LOWER.contains(pass.at(i))) containsLower = true;
38             if (UPPER.contains(pass.at(i))) containsUpper = true;
39             if (NUMERAL.contains(pass.at(i))) containsNumeral = true;
40             if (OTHER.contains(pass.at(i))) containsOther = true;
41         }
42         containsAllTypes = (containsLower == useLower) && (containsUpper ==
               useUpper) && (containsNumeral == useNumeral) && (containsOther ==
               useOther);
43     }
44     ui->passwordLineEdit->setText(pass);
45 }
```

Individual characters are randomly selected from each of the enabled symbol groups available on a standard keyboard. Specifically, the `generate()` function checks to see which symbol groups the user has enabled, then starts a loop to choose a random symbol for each position in the password. An `AutoSeededRandomPool` from the Crypto++ library is used to ensure the output is highly unpredictable. Within the loop, a random symbol group is chosen, then a random symbol inside of the chosen group is appended to the password. Once a candidate password is generated, a check is done to confirm that at least one symbol from each enabled symbol group exists in the password; new passwords will be repeatedly generated until this condition is satisfied.

Passwords generated here are automatically inserted into the password field of the currently selected account in the main program, but the generator can also be accessed via the menu for convenience.

Strength Calculation

Password strength is determined using the naive calculation of entropy mentioned in Chapter 2. Code for this is in the `StrengthCalculator` class, and is called by the main and `Generator` classes to visualize strength via a `QProgressBar`. In Listing 5.14, a password string is passed in, and looped through, checking for the presence of various character groups. Ultimately, the length of the password, and the number of possible symbols must be known to estimate the entropy.

Listing 5.14: Password strength calculation function

```
1  double StrengthCalculator::naiveEntropyBits(const QString &pw)  // Calculate raw
       bits of entropy (assuming password made with uniform probability distribution)
2  {
3      bool hasLower, hasUpper, hasNumeral, hasOther;
4      hasLower = hasUpper = hasNumeral = hasOther = false;
5      int len = pw.length();
6      for (int i = 0; i < len; i++)
7      {
8          QChar ch = pw.at(i);
9          if (ch.isLower()) hasLower = true;
10         if (ch.isUpper()) hasUpper = true;
11         if (ch.isDigit()) hasNumeral = true;
12         if (ch.isSymbol() || ch.isSpace()) hasOther = true;
13     }
14     int possibleSymbols = 0;
15     if (hasLower) possibleSymbols += NUM_LOWER;
16     if (hasUpper) possibleSymbols += NUM_UPPER;
17     if (hasNumeral) possibleSymbols += NUM_NUMERAL;
18     if (hasOther) possibleSymbols += NUM_OTHER;
19     return (possibleSymbols > 0) ? ((double) len) * log2((double) possibleSymbols)
           : 0.0;   // Avoid undefined log_2(0)
20 }
```

The naive formula is utilized on line 19, where L corresponds to `len`, and N corresponds to `possibleSymbols`.

5.7 Auto-Type

PassMan is able to automatically simulate keyboard entry via running the external binary `xdotool` (available in standard system repositories) in a separate process. The username and password are written to the standard input of this process, after which `xdotool` will simulate keypresses of the data into the desired login form. Typically, the user would select the username field of the login form they wish to authenticate with, then select *Auto-Type Entry* from the PassMan window after selecting the correct account. Listing 5.15 shows how this is accomplished.

Listing 5.15: Password auto-type function

```
1  void PassMan::on_actionAuto_Type_Entry_triggered()  // Perform auto-type
2  {
3      QString command("xdotool -");
4      QString tabCommand = "key Tab";
5      QString returnCommand = "key Return";
6
7      this->setWindowState(Qt::WindowMinimized);
8      QProcess* proc = new QProcess();    // Send username
9      proc->start(command, QIODevice::ReadWrite);
10     proc->write(ui->usernameLineEdit->text().prepend("type ").toUtf8());
11     proc->closeWriteChannel();
12     proc->waitForFinished(-1);
13     proc->close();
14
15     proc->start(command, QIODevice::ReadWrite);
16     proc->write(tabCommand.toUtf8());  // Send tab key
17     proc->closeWriteChannel();
18     proc->waitForFinished(-1);
19     proc->close();
20
21     proc->start(command, QIODevice::ReadWrite);
22     proc->write(ui->passwordLineEdit->text().prepend("type ").toUtf8());  // Send
           password
23     proc->closeWriteChannel();
24     proc->waitForFinished(-1);
25     proc->close();
26
27     proc->start(command, QIODevice::ReadWrite);
28     proc->write(returnCommand.toUtf8());    // Send return key
29     proc->closeWriteChannel();
30     proc->waitForFinished(-1);
31     proc->close();
32     delete proc;
33 }
```

The process is opened as usual, except that it is started several times to simulate a series of user interactions. First, the PassMan window is minimized on line 7 to return window focus to the window previously selected by the user (like a web browser). The username is sent, followed by a *Tab*, the password, and finally the *Return* key.

Chapter 6

Conclusion

PassMan is currently at version 1.3.4, a stable release that is viable for continual usage in a Linux environment. It modernizes password usage by offering all common password management features, including confidentiality and assured integrity of account information, auto-type functionality, password strength calculation, and customizable password generation. Its simplistic interface was designed to minimize the learning curve for first-time password manager users. In some regards, it exceeds the default security level of other password managers, because it requires a second authentication factor via the YubiKey hardware token.

However, PassMan's security and usability could be improved further to make it a full-fledged password manager like KeePassX. As it currently stands, the auto-type functionality is only compatible with simplistic login forms (where the username and password field are on the same page). Customizable auto-type, where the user can specify the order of field entry and emulation of keypresses, would allow usage with more complex login sequences such as Google's two-page system.

Password strength calculation could also use improvement. Currently, a naive calculation of entropy is used, meaning that it is assumed each possible symbol has equal chance of being chosen for a position in a password. While this is true when PassMan generates passwords, it is not for user-influenced passwords, where the frequency of alphabet letters is far from uniform in the English language. In practice, this means that a password with words in it may be misrepresented as holding more entropy than it actually does; consequentially, a dictionary attack on that password could be more viable than the user expects. Calculation

could instead be done using Claude Shannon's method, where the probability of each possible symbol is taken into account. In this case, the common letter "e" would be worth less entropy than the letter "z" in a password, and overall entropy is more accurately described.

Run-time security may be increased through the use of "secure memory" features available in most operating systems. `CryptProtectMemory()` in Windows, or `gcry_malloc_secure()` in Linux will encrypt allocated memory regions that hold passwords and other sensitive data. Doing so would shift responsibility for zeroing memory to the operating system, and further hinders other programs from reading PassMan's memory.

At-rest security could be expanded to a third authentication factor such as a "keyfile", a sequence of data that is used as part of the master key for encryption. In this case, the user's master password, YubiKey, *and* the keyfile would be necessary to recover the user's data.

Finally, PassMan could be refactored to be fully cross-platform compatible. Currently, code that communicates with the YubiKey and code for auto-typing is specific to the Linux environment. These sections could be generalized for Windows and Mac OS X, which would improve its appeal to users, and keep their accounts secure across every machine they encounter.

Bibliography

[1] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. http://cseweb.ucsd.edu/~mihir/papers/hmac-new.pdf, 2006.

[2] The Qt Company. About us. https://www.qt.io/about-us/, 2017.

[3] The Qt Company. Signals & slots. http://doc.qt.io/qt-4.8/signalsandslots.html, 2017.

[4] Crypto++. Benchmarks. https://www.cryptopp.com/benchmarks-p4.html, 2009.

[5] Crypto++. Random number generation. https://www.cryptopp.com/wiki/RandomNumberGenerator, 2016.

[6] Crypto++. Advanced encryption standard. https://www.cryptopp.com/wiki/Advanced_Encryption_Standard, 2017.

[7] Crypto++. Authenticated encryption. https://www.cryptopp.com/wiki/Authenticated_Encryption, 2017.

[8] Crypto++. Gcm mode. https://www.cryptopp.com/wiki/GCM_Mode, 2017.

[9] Crypto++. Key derivation function. https://www.cryptopp.com/wiki/Key_Derivation_Function, 2017.

[10] Joan Daemen and Vincent Rijmen. The design of rijndael. https://autonome-antifa.org/IMG/pdf/Rijndael.pdf, 2002.

[11] Inc. IBM. Hmac: Keyed-hashing for message authentication. https://tools.ietf.org/html/rfc2104, 1997.

[12] Texas Instruments Inc. C implementation of cryptographic algorithms. http://www.ti.com/lit/an/slaa547a/slaa547a.pdf, 2013.

[13] KeePass. Security. http://keepass.info/help/base/security.html, 2017.

[14] KeePass. Keepass & yubikey. http://keepass.info/help/kb/yubikey.html, 207.

[15] RSA Laboratories. Pkcs #5: Password-based cryptography specification version 2.0. https://tools.ietf.org/html/rfc2898, 2000.

[16] LastPass. How it works. https://www.lastpass.com/how-it-works, 2017.

[17] LastPass. Lastpass and yubikey. https://lastpass.com/yubico, 2017.

[18] Matt Mackall. random.c – a strong random number generator. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/drivers/char/random.c?id=refs/tags/v3.15.6#n52, 2005.

[19] PC Magazine. Keepass. http://www.pcmag.com/article2/0,2817,2408063,00.asp, 2012.

[20] NIST. Announcing the advanced encryption standard (aes). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, 2001.

[21] NIST. Estimating password strength. http://csrc.nist.gov/archive/pki-twg/y2003/presentations/twg-03-05.pdf, 2003.

[22] NIST. The galois/counter mode of operation (gcm). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf, 2007.

[23] NIST. Security policy - crypto++ library. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp819.pdf, 2007.

[24] NIST. Validated fips 140-1 and fips 140-2 cryptographic modules. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm#2267, 2017.

[25] NSA. National policy on the use of the advanced encryption standard (aes) to protect national security systems and national security information. http://csrc.nist.gov/groups/ST/toolkit/documents/aes/CNSS15FS.pdf, 2003.

[26] Cryptography Coding Standard. Coding rules. https://cryptocoding.net/index.php/Coding_rules, 2017.

[27] Wired. Facebook pushes passwords one step closer to death. https://www.wired.com/2013/10/facebook-yubikey/, 2013.

[28] Yubico. Improving yubikey physical security. https://www.yubico.com/2014/04/improvements-physical-yubikey-attacks/, 2014.

[29] Yubico. How it works. https://www.yubico.com/why-yubico/how-yubikey-works/, 2016.

[30] Yubico. About the yubikey 4 series. https://www.yubico.com/products/yubikey-hardware/yubikey4/, 2017.

[31] Yubico. Challenge response. https://www.yubico.com/products/services-software/personalization-tools/challenge-response/, 2017.