Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

NoKey - A Distributed Password Manager

Master Thesis

Florian Zinggeler
zifloria@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Simon Tanner, Gino Brunner
Prof. Dr. Roger Wattenhofer

June 7, 2018

Abstract

Choosing strong and unique passwords for every online account is difficult without using a password manager. However, most password managers require users to remember a strong master password to securely store their credentials.

In this project, we present NoKey, a password manager that does not make use of a master password. Instead, when users want to access their stored passwords, all they have to do is to allow the access on another of their paired devices. How many devices are needed to unlock a password can be configured. This is achieved by making use of Shamir's Secret Sharing. This algorithm allows storing a secret in a distributed way, such that the secret can only be recovered if enough parts of the secret are brought together.

NoKey is available for Android, as a browser extension for Firefox and Chrome and as a web application that runs on most modern browsers. The application is completely open source and freely available at `nokey.xyz`.

Keywords: password manager, CRDT, Shamir's Secret Sharing, elm

Contents

Chapter 1

Introduction

Millions of online accounts get breached every month. As of 2018, the website haveibeenpwned.com by security expert Troy Hunt lists over 5 billion stolen accounts [25]. As a user, not much can be done to protect oneself against such a data breach. However, by only using strong and unique passwords everywhere, the impact of such a breach can be greatly reduced. Unfortunately, efforts to educate users about the perils of weak passwords seem to be fruitless, as the majority of used passwords are terrible[1].

Strong passwords consist of a long sequence of random letters and symbols that have never been used before. However, since remembering these passwords would be practically impossible, many use a password manager to help with that task. Popular password managers include LastPass [33], Enpass [20], 1Password [1], Dashlane [14], KeePass [30].

Most password managers work by storing passwords in a database, encrypted with a user chosen master password. The security that a password manager offers is therefore proportional to the strength of the chosen master password. Users then have to enter that master password every time they want to retrieve their saved passwords or if they want to save a password.

In this work we present a distributed password manager called NoKey that does not use a master password. Instead, passwords are stored on multiple devices of the user in such a way that they can only be read if enough devices come together. With this application, using a saved password only requires that the user gives permission on another device. For instance, to access the stored passwords on the users laptop, the user has to allow that access on her phone.

1.1 Goals

The goal of this project is to create a password manager that can work without any master password, while still offering comparable or possibly even better

[1] 86% of Passwords are Terrible, Troy Hunt, https://www.troyhunt.com/86-of-passwords-are-terrible-and-other-statistics/, Accessed: June 7, 2018

security than traditional password managers. One of the design goals for this application is that it should be as convenient as possible to use. Ideally, we hope that people who are currently not using a password manager, possibly because they find it too cumbersome to remember and type in their strong master password every time they need their passwords, might find our application to their liking. From the start, we knew that we wanted to make use of a secret sharing scheme to offer security. This also makes the application more flexible, as it allows users to adjust the trade-off between convenience and security when storing passwords.

1.2 Related Work

A project that started out with a similar goal to this one has already been developed as an internal project at the same institute in "Convenient Password Manager" [16], but they took a very different path. Conceptually, there is quite a bit of overlap between the two projects, however, they differ in the targeted platforms and in the implementation details. For this work we started from scratch except for the experiences learned from the previous project.

Traditional password managers like LastPass [33] or Enpass [20] have laid the groundwork of how a user-friendly password manger should work. Many of them are able to detect sign up and login pages and thus are able to conveniently offer to fill password forms automatically. Many also allow users to synchronize all their saved passwords across all their devices.
These password managers all use a similar approach to security. They require users to choose a strong master password from which they derive a key using a key stretching functions such as PBKDF2 [29] or Argon2 [8]. Then, the derived key is used to encrypt passwords with a symmetric encryption scheme such as AES [13].

A different approach to password management is that of a stateless password manager. This approach also requires a master password, but no passwords are actually stored. Instead, the password is derived from the website URL, the user name and the master password. This is usually done by feeding the above information to a strong hash function from which the password is derived. This approach has the advantage that users cannot lose their passwords, as they can always be derived again. However, using this approach makes it difficult to change passwords or to deal with complex password policies. Some examples of applications that make use of this approach are LessPass [34] and novault [36].

Chapter 2

Concept

NoKey was designed to work in a peer to peer fashion. Clients communicate directly with each other with a message passing protocol. However, for practical reasons this direct communication is not actually a direct peer to peer communication, but happens over a simple relay server.
The server has another function, which is to facilitate the pairing process as described in Section 2.2. Apart from that, its only function is to forward messages.

There are different variations of the client application: A web application, a browser extension [9] and an Android app. All of them offer the full functionality of NoKey, except for the few extra functionalities offered by the corresponding platforms.

This chapter will provide an overview of the technical aspects of NoKey.

2.1 Communication

There are two ways for the clients to communicate with the server: one via HTTP requests, the other via WebSockets [48]. A WebSocket is a protocol implemented on top of TCP which provides an interface to a two way communication channel for web browsers. In contrast to an HTTP request which consist of a single query-response, a WebSocket is a bidirectional, stateful connection to the server. This way, the server can push messages to the client, which would not be possible using HTTP.

WebSockets are used to implement the direct communication between clients. When the application starts, every client opens a WebSocket connection to the server. This connection is kept open at all times. In case a client loses the connection, it will try to reconnect with an exponential backoff strategy.

To send a message from one client to another, a client sends an HTTP POST request to `/sendMsgTo/:otherId` with the message data in the request body. The server then pushes this message in the corresponding WebSocket of the receiving client. If the client is listening at that time, the message will be delivered

and the receiver can take appropriate actions.

2.2 Pairing

Since NoKey is a distributed password manager, we need some way to establish trust between different devices of a user. To do that, a user can pair her devices. After two devices have been paired, they will be able to communicate with each other and thus are able to synchronize their state. This section describes how this trust is established.

When starting NoKey for the first time, a random ID and two RSA [41] public-private key pairs are generated. After the pairing process, both devices have added the ID and the public keys of the other device to their list of trusted devices. One of the RSA keys is used to verify the authenticity and integrity of messages, the other to encrypt confidential data.

The pairing process is initiated by sending a `StartPairing DeviceId` request to the server. The server then stores the ID along with a randomly generated token, which is also sent back as a reply. The device that started the request then displays that token encoded as an easy to type list of words or alternatively, as a QR code.

The user can then either scan the QR code or enter the token on the other device. The other device will then send the token back to the server, which will reply with a `PairedWith DeviceId` message if the token is valid and has not expired yet. The device that got the `PairedWith` message will then directly contact the other device with a `FinishPairing Token SyncData` message. The `SyncData` contains all the replicated state and is further described in Section 2.3. At this point, both devices know that the pairing was successful, but they do not commit to it yet. The device that got the `FinishPairing` message will send an acknowledgment back and then wait on receiving an acknowledgment in return. After that, both devices will be paired and will have their state synchronized with each other.

If not done carefully, it is very easy for this process to go wrong. For example, without the last round of acknowledgments, if one device can communicate with the server via HTTP but not via the WebSocket, it is possible to get into an inconsistent state. One device will think that the pairing process was successful, while from the perspective of the other device, it will look as though it failed. By adding an additional round of acknowledgments, we can guarantee that this scenario would result in the desired outcome with both devices knowing that the pairing was not successful.

Note that we cannot completely eliminate the possibility of an inconsistent state. It is always possible that the last acknowledgment gets lost (for instance if the device gets turned off), and never reaches the other device. If that happens,

the two devices will disagree on whether the pairing succeeded or failed. This is because the problem of pairing is essentially equivalent to the Two Generals' Problem [4], which is unsolvable. However, the time window for something to go wrong is very small here, i.e., only about two round trips.

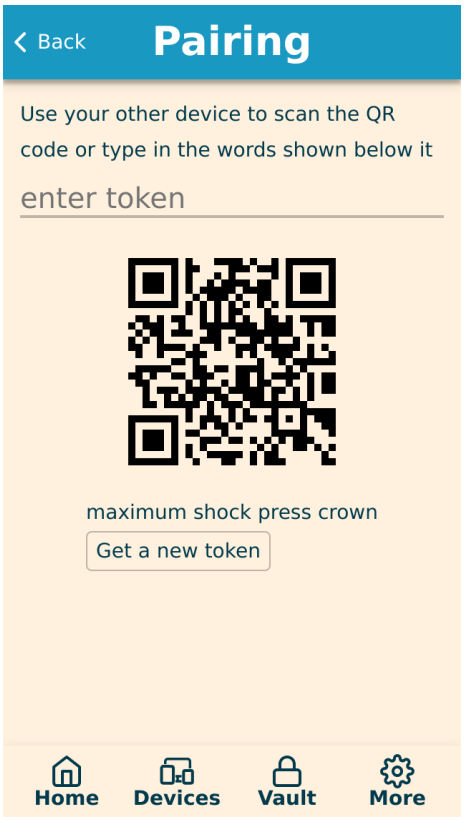


Figure 2.1: The pairing screen

2.2.1 Visual Hash Icon

After pairing a device, we want to make sure a user immediately recognizes if something went wrong.

To this end, every device has a unique icon that gets derived from its ID, essentially forming a visual hash of the ID. This icon is prominently displayed and should automatically be remembered by the user.

If a user now wants to pair a new device, but is observed by an attacker, the attacker could enter the token faster than the user, which would result in getting paired with the attacker's device. Thanks to the hash icon, such an attack could be quickly recognized, as the device ID of the attacker is unlikely to hash to the same icon as the one from the device the user wanted to pair. The user will

then recognize that something went wrong and should then remove the attackers device from the list of trusted devices.

Designing a Visual Hash Icon

An ideal hash icon for our application should have the following properties:

1. Large number of possible icons
2. Look pretty
3. No two icons should look alike
4. Easy to memorize

Most of these design goals are conflicting: If we want to increase the number of possible icons, e.g., by increasing the complexity, icons will become harder to remember and distinguish.

Given these goals, our hash icon is composed of the following elements: An icon taken from the large icon pack Font Awesome [22] and three colors taken from a small set of nice colors. Not all color combinations work well, especially when considering people with color-blindness, so combinations with a bad contrast ratio are discarded.
How these elements are combined is determined by hashing the device ID and by using the resulting bits as a starting seed for a random number generator. Using that random number generator, a valid combination of three colors and an icon is picked. These are then combined into the final hash icon. This combination of elements results in a good trade-off of the above listed goals. Most combinations look decent, no two icons can be confused and for an acceptable minimum contrast ratio there are approximately one million possible icons.

The idea for such a visual hash function was inspired by GitHub's Identicons [27] for users that did not yet set their own avatar. An overview of other visual hash functions can be found in [6].

The implementation of this visual hash function has been published on the official Elm package channel as a separate library [19]. Figure 2.2 shows a few visual hashes generated with the method described above.

Figure 2.2: A few examples of our visual hashes

2.3 Synchronizing State

An important part of NoKey is the synchronization of its state. If users add a password or otherwise perform any other modification to the state, they expect that this information will be synchronized to their paired devices.

Since we want to allow modifications even when users are offline, we can not always have a consistent state on all clients. Instead, the replicated state used by NoKey eventually becomes consistent, meaning that as soon as devices are able to exchange state information, their state will be consistent again. This is achieved by expressing the whole shared state as a conflict-free replicated data type (CRDT).

2.3.1 Shared State CRDT

A CRDT is a data structure where copies can be modified independently and concurrently. The newly modified structures can then be merged which will result in a new copy that contains all modifications. Merging a CRDT is guaranteed to not result in a merge conflict. CRDTs have been formalized in "Conflict-free replicated data types" [43], which also provides many building blocks for creating more complex structures.

The useful properties of CRDTs are achieved by making the data structure less powerful. An example of a very simple CRDT is the Grow Only Set: This is a set where elements can only be added, never removed. It is obvious that this simple data structure has the desired properties, as the order in which elements are added to a set does not matter, but its uses are very limited. However, by combining these simple structures and ideas, more useful CRDTs can be derived.

NoKey uses these CRDT building blocks to express all of its replicated state. One of the most commonly used data structure is a CRDT version of a dictionary (also called hash map, associative array or key value store) which is not mentioned in the original paper. This dictionary like data type (here called OR-Dict) is built with an OR-Set (Observed Remove Set [44]) to maintain the keys and a store for the corresponding values. Similar structures have been used in various libraries for distributed systems like Akka [3] or in Riak [40], a distributed database.

2.4 Password Generation

Since people are very bad at coming up with random passwords, NoKey includes a password generator. When signing up for a new account for a website, the browser extension of NoKey will automatically show its password generator to

encourage users to choose a strong password. By default, a password with 16 random characters will be offered.

If a website has restrictive password policies, a user can customize the generated passwords. A user can adjust the password length and which characters are allowed. Additionally, it is possible to indicate that one wishes to always include a character from a specific character set, such as a number or a special character.

This should make it possible to handle most password policies. However, some websites have very complex password policies that also forbid certain sequences of characters. For instance, some websites do not allow passwords that contain the sequence "123". The generator cannot deal with such complex requirements, but if such a sequence would show up, a user can just generate another password or manually edit the suggested password.

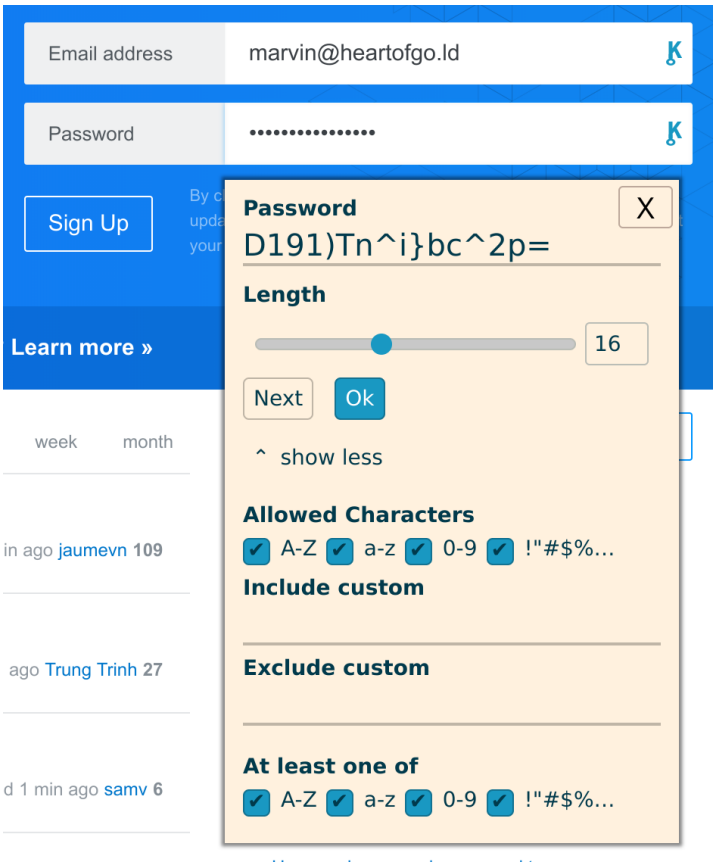


Figure 2.3: The password generator, as seen when signing up on a website

2.5 Storing Passwords

In a traditional password manager, passwords are usually securely stored by using a symmetric encryption scheme such as AES [13] to encrypt passwords with the user defined master password. In contrast, NoKey stores passwords by combining AES with Shamir's Secret Sharing [42]. Shamir's Secret Sharing is an algorithm for sharing a secret with a set of peers. Once the secret shares have been distributed, the secret can be recovered by bringing the shares of a subset of peers back together. How many peers are required to reconstruct the secret can be configured.

Shamir's Secret Sharing works by constructing a random polynomial $P(x)$ of order k where $P(0) = Secret$. A secret share is then an $(x, P(x))$ tuple. The secret can be reconstructed from $k + 1$ secret shares by solving a system of equations for $P(0)$. For this scheme to be information theoretically secure, all calculations need to be done using finite field arithmetic. Otherwise, an attacker is able to gain information about the secret even if not enough shares are known.

When adding the first password to NoKey, a new password group is created. A password group is a collection of passwords that are encrypted with the same symmetric key. The group key is a randomly generated, 256 bit long string. Passwords are then encrypted with the group key using AES. The complete group key is never stored anywhere. Instead, Shamir's Secret Sharing is used to generate secret shares of the group key. These shares are then distributed to all trusted devices, such that eventually every device receives a share for each password group.

Each group has a security level associated with it. The security level indicates how many clients we need to unlock a password group. It corresponds to the degree of the polynomial used in Shamir's Secret Sharing algorithm.
Then, when users want to access the passwords in a password group, their device will ask all trusted devices for their share for that group. The user then has to allow this on the other devices until enough shares have been collected. Now the device that requested the shares can reconstruct the group password using Shamir's Secret Sharing. Passwords can then be read out with the group password using AES to decrypt the saved passwords.

2.6 Key Boxes

It seems like with this approach it is not possible for users to use their saved passwords if they do not carry enough devices with them. With only one device, there simply would not be enough secret shares present to unlock any group. To still make NoKey useful in a scenario where a user does not carry enough devices with them, key boxes are introduced. A key box is a password protected vault

that contains an additional share per group. This way, a user can for instance open a group with security level two by opening a key box while only carrying a single device.

A key box keeps track of the following values: The AES encrypted secret shares, a salt used to derive the key and a password hash plus the salt used for hashing the password. These value are part of the synchronized state and are replicated to all devices.
The symmetric key used to encrypt the secret shares is derived from the user defined password. To derive this key, the password is first fed to the key derivation function PBKDF2 [29] along with a random salt that was generated when creating the box.
Since the encrypted secret share is just a number and there is no way to check if our decryption used the correct key, we need another way of telling the user whether the entered password was correct. For this, we store the PBKDF2 hash of the password with a different salt to the one used to derive the key. To check whether the entered password is correct, we just hash the entered password together with the stored salt and compare the hash with the saved hash. This is very similar to how the login process works on a UNIX system.

Key boxes are an optional feature and meant to be used less frequently than the more convenient way of just confirming on another device. But even if used infrequently, it is still a good idea to add a key box anyway, as this also makes losing a device less severe. Without a key box, if a user has only two devices and loses one device, she would be unable to unlock a group secured with security level two and consequently lose all passwords in it.

2.7 Security

In this section, we will have a look at the security offered by NoKey. To make this section easier to read, we assume that a user only made use of security level two. In a scenario where the user used a higher security level, the attacker would need to steal more devices to have the same possibilities as in the scenarios below.

2.7.1 Read Only Attack

In this scenario, we assume an attacker was able to steal one device and can now access all internal state of the application. The attacker will find all the encrypted passwords and a secret share for each password group. Since one secret share does not give away any information about the secret, the available information is useless and the attacker cannot read any passwords. The single secret share also does not give the attacker an easier way of brute forcing the secret. It would be easier to directly brute force the group key, however this

is computationally infeasible. At 256 bits, the used key size is well above the currently NIST recommended minimum length of 112 bits [7].

Sometimes the application state contains more secret shares, e.g., while shares are still in the process of being distributed. However, these shares are encrypted with the RSA public key of the receiving device, so these shares cannot be read either.

The bad news is that privacy wise, the attacker can learn a bit about the user. NoKey stores websites and corresponding user names in plain text. This is needed by the browser extension to provide a good user experience. If this information was stored in an encrypted form, the browser extension would not be able to provide form auto-completion without the user having to unlock a certain password group first. In the last chapter (4.1), we look at how this could be improved, at the cost of usability.

2.7.2 Man in The Middle

Since the communication between clients and server is secured with TLS, a man in the middle attack seems very unlikely. However, the server that forwards messages can basically be seen as a man in the middle. So if a user does not trust the official server, they are probably wondering what attacks a malicious server would be able to do.

A malicious forwarding server could basically perform the same attack as the read only attack described in the previous section. The only big difference to the previous section is that the server will not have access to any secret shares at all.
The server is also not able to alter messages, as all messages are authenticated and integrity protected using an HMAC [32].

The only active attack that a man in the middle could perform is during the pairing phase, where the initial trust between devices is established. However, this type of attack can be quickly detected by the user, as described in Section 2.2 on pairing.

2.7.3 Read-Write Attack

In this scenario, the attacker managed to steal a device and can now communicate with the other devices to achieve his goals. Similar to the previous scenario, passwords are still not accessible for the attacker. However, if the user is not careful, the attacker might be able to trick the user into pressing “allow” to gain more secret shares.

Since the attacker is not able to read any passwords, she might try to cause harm in other ways. By using the underlying communication protocol directly,

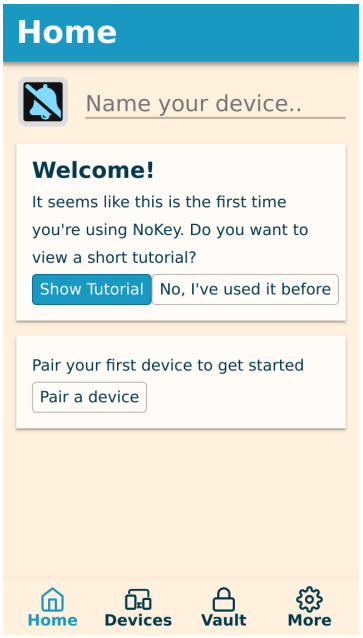
the attacker could alter the replicated state in all sorts of ways, as long as the structure follows the CRDT implementation. This way, it would be possible to delete all stored passwords. In Section 4.2, we look at how this could be limited.

The attack described above only works if the users devices are online at the same time as the stolen device and only if the stolen device was not removed from the list of trusted devices yet.
In practice, doing such an attack is made more difficult by the fact that an attacker has to write a custom client that allows such operations and has to be able to access the internal storage where the state is kept. Especially on Android, the latter point might be more difficult, as the device would have to be rooted first.

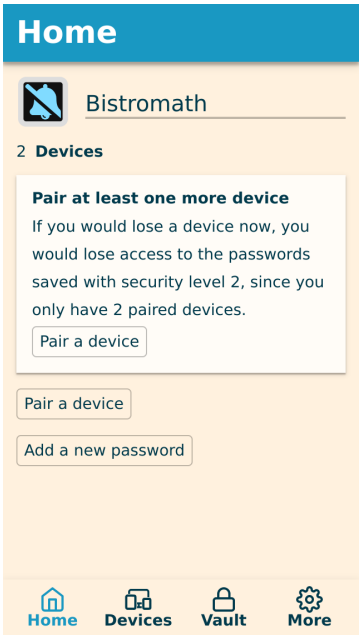
2.8 Design

In this section we look at what goals drove the design choices and how this manifests in the final implementation.
One of the design principles when creating NoKey was that a user should be able to discover all of its features gradually. That is, when starting NoKey for the first time, some UI elements are hidden.
Which UI elements to show is mostly influenced by the number of paired devices. For instance, if a user only paired two devices, the password UI will not show an option to move passwords into another group.

To further help users along, the home screen displays context dependent hints. For instance, if a user makes use of security level three, but only has three devices, a hint is displayed. The hint states that if the user would lose a device now, the passwords stored with level three would be lost. It further contains a shortcut to the pairing screen, to encourage users to pair yet another device.



(a) Home screen hint on first usage



(b) Hint if there are exactly as many devices as the highest security level

Chapter 3

Implementation

When developing an application like NoKey, there are many technologies and programming languages to choose from. Some of the choices taken here have been purely made on personal preferences, others on technical merits and some were automatically made by the corresponding target platform. This chapter will provide an architectural overview and will explain which choices were made and why.

3.1 Overview

Since NoKey was from the start envisioned to be used on multiple devices, it made sense to choose a technology that would allow us to reuse as much code as possible. We chose the web platform as our target, since almost all devices can display a website and since the browser extension had to be written in JavaScript anyway. This way, the majority of the application code, including the UI, could be shared on all platforms. This worked out well, as can be seen in Figure 3.1. All the Elm code and some of the JavaScript code gets used on all platforms, while the rest is platform specific. This also means that adding a new platform is not much work, as long as the platform can display a website and supports the used APIs.

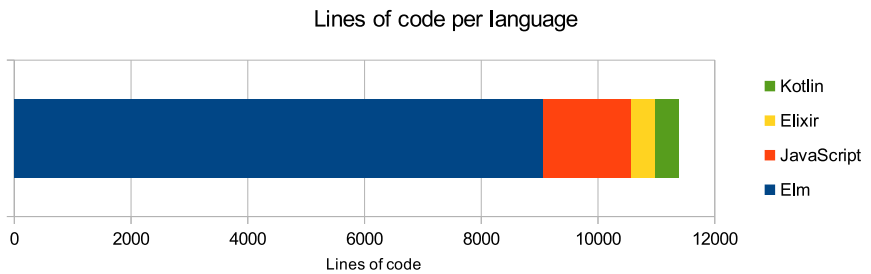


Figure 3.1: Lines of code broken down by language

3.2 The Server

The server is written in Elixir [17] and implemented using the Phoenix framework [37]. It implements a simple REST [46] API that provides functionality for pairing and client to client communication.

The server does not store any usage data, it does not even have a database. The only state that is maintained on the server is short lived and is only kept in RAM. This makes it very easy to deploy the server, as there is no need to set up a database.

Elixir is a dynamically typed functional programming language that runs on the Erlang VM [5]. It is very well suited for writing concurrent applications and lends itself well for server side code.
The Phoenix framework is a library for Elixir that makes it easy to create web applications. It provides a high level abstraction for WebSockets, called channels, which make it easy to create real time applications such as chat applications. In NoKey, channels are used to implement the forwarding of messages between clients.
The framework performs very well under high load: it has reportedly been able to sustain two million simultaneously connected clients via WebSockets on a single machine [45].

3.3 The Clients

There are currently three different versions of the client: a web application, a browser extension and an Android client.
Most of the client application is written in Elm [12, 18], a purely functional programming language that compiles to JavaScript. Parts that are specialized for a certain platform are written in the respective platform's language. E.g., the additional functionality needed by the web extension is written in JavaScript.

3.3.1 Shared Code

The part of the code that handles everything from state management to communication between devices to displaying the user interface is written in Elm and is used across all clients.
Since Elm and functional programming in general are not very mainstream, this section will provide a short overview of what this means in practice.

Functional Programming and Elm

Elm is part of the ML (Meta Language) family of languages and heavily inspired by Haskell [26], Standard ML [35], F# [21] or similar. However, compared to the above mentioned languages, it is carefully designed to be much simpler. It is a statically typed, purely functional programming language that targets the web platform.

For a language to be purely functional means that there are no mutable variables to hold state and that every function always returns the same value given the same input. Also, compared to an imperative programming language, a purely functional one does not have statements, only expressions. Meaning there are no constructs for "while" or "for" loops as found in most imperative programming languages. Additionally, functions cannot perform any side effects, such as performing an HTTP request or reading from a file. To still be useful in practice, a purely functional language therefore has to offer another way to deal with these cases.
In Elm, there exists a runtime that keeps track of the application state and performs effects when needed. This ensures that every application is written in a similar style and allows the language to remain purely functional.

At first sight, these restrictions might seem limiting, but in practice they are not and offer many benefits. One of the most compelling selling points of Elm is that programs written in Elm are almost guaranteed to not produce any runtime errors. This is achieved with its static type system that forces programmers to handle all possible cases that can happen in an application. The only way to really crash an Elm application is to write an infinitely recursive function, as the compiler obviously has not solved the halting problem [47]. In practice, this rarely happens.
Another benefit of writing code in a purely functional style is that it is very easy to reuse code. Since every function is pure, all functions are completely independent from each other as they can not depend on some internal state.

Having to follow this strict architecture made a few things that can be difficult in other languages extremely simple.
For instance, the browser extension has a few specialized user interfaces not used anywhere else. Since how the UI is displayed is just a pure function that takes in the current state and returns a description of the HTML to display, these views are just an alternative view of the same internal state. There are no possibilities for bugs that would cause this specialized views to become out of sync with the main view, as they both are derived from the same state.
Another thing that can be difficult in other languages is serializing the application state and restoring it later. This is almost trivial to do in Elm, since the whole application state can only live at a single place. Implementing persistent state in other programming languages has to be carefully considered whereas in Elm it can be an afterthought.

By not having access to mutable variables, a large class of programming errors can be completely eliminated. In practice, this theoretical argument seems to be supported by a large scale study on code quality conducted on GitHub [23]. The study concluded that "[t]he data indicates functional languages are better than procedural languages; it suggests that strong typing is better than weak typing; that static typing is better than dynamic; and that managed memory usage is better than unmanaged" [39].

3.3.2 Web Application

The web application is implemented as a Progressive Web App (PWA) [38]. It offers all the functionality of the other clients, and can be run in any modern web browser. The only modern browser where NoKey does not run is Safari, as it does not implement the full Web Crypto API as needed by NoKey.

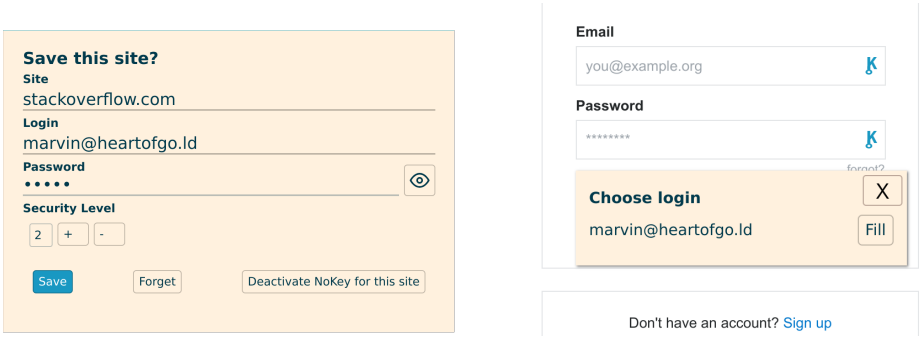
The term PWA is not very precise, but it generally means that a website makes use of some of the more modern web APIs to provide an app like experience. In the case of NoKey this means the ability to use the website when offline, as well as being able to add the website to the home screen on supported platforms, such as Android.

3.3.3 Web Extension

The web extension is basically just a small wrapper around the web application. It lets the web application run in the background and displays its UI in the extension popup. On top of that, it can detect and classify sign up and sign in pages and offer to remember and fill out passwords. Figure 3.2 illustrates how this looks like.
The extension is available on Chrome and Firefox. It should also run on Microsoft Edge and Opera, but this has not been tested. Safari seems to be the only modern browser that uses a different API for extensions, all other browsers have adopted the WebExtension API [11].

3.3.4 Android

Just like the browser extension, the android application is also just a simple wrapper around the web application. The application runs completely inside a WebView [10].
The Android application injects a few additional functionalities into the web application. For instance, on Android we can make use of the camera to scan QR codes for pairing. Android also deals differently with uploading and downloading files, which the wrapper also has to properly handle to provide the password import/export feature.



(a) Popup after signing up (b) Auto fill on login

Figure 3.2: Some exclusive features of the browser extension

This wrapper is written in Kotlin [31], a Java alternative that runs on the JVM. One reason for choosing Kotlin over Java is that it has null safety, meaning that most null pointer exceptions are caught at compile time. Another reason for choosing Kotlin over Java is that Google seems to be focusing their development efforts mostly on Kotlin, which can for instance be seen in their latest developer conference[1].

[1] Google I/O 2018: What's new in Android , https://android-developers.googleblog.com/2018/05/google-io-2018-whats-new-in-android.html, Accessed: June 7, 2018

Chapter 4

Outlook

There are many ways to expand NoKey to be more secure and feature rich. In this chapter we present some of the ideas we had that were out of scope for this project, but would make for a good addition.

4.1 Improved Privacy

As written in Section 2.7.1, NoKey stores saved user names and site URLs in plain text. It would be useful to have an option to also encrypt that information when saving a password, to better protect the privacy of users. However, this privacy feature would have a high usability trade-off: As long as the group stays locked, the browser extension would not be able to offer to fill out a password form, as it cannot know if the user has an account on the current site.

4.2 Keep Deleted Passwords

In Section 2.7.3, we wrote that it would be possible for an attacker who managed to steal and unlock a device to delete all the saved passwords of the user. An easy way to prevent such an attack would be to move a password to a bin before it gets deleted.

The bin would than be local to a single device. With such a bin in place, to fully delete a password, a user would have to remove it from the bin of each paired device after having pressed delete.

It is not clear whether the added safety is worth the usability trade-off. Having to go to each device separately to truly delete something is neither convenient nor intuitive. On top of that, a bin would add additional complexity to the UI, which makes it harder to keep a clean design.

4.3 Add Trusted Friends

With key boxes (Section 2.6), it is possible to unlock a password group even if a user does not have enough devices at hand. However, this comes at the cost of having to remember an additional password.
One idea to completely get rid of passwords would be to add trusted friends. In addition to creating a secret share for each device, NoKey could then also create a secret share for each trusted friend. Friends would not be able to access the users passwords, since they possess only one secret share and also do not have access to the encrypted passwords. To unlock a device, users could then ask their friends to send their secret shares.

To make sure this feature would not compromise the security of NoKey, users would have to agree upon some way to make sure that it is truly them that are asking for the key, and not a thief. As an example, they might send a selfie in an agreed upon silly pose, such that a thief would not be able to impersonate them.

4.4 QR Codes

Another way to get more secret shares without pairing another device would be to print them out as QR codes. A QR code would then contain one share for a password group. When unlocking a password group, a user could then scan the QR code to access an additional share.

However, a QR code is less safe than a device. With a device, the shares are usually also protected by another factor such as a fingerprint, as the device has to be unlocked first. A QR code has no such protection. Additionally, a QR code can easily be stolen undetected by simply taking a picture of it. This is a problem, since a user has no way of knowing if a QR code was stolen. A stolen device on the other hand is detected quickly and can be removed from the list of trusted devices as soon as possible.

4.5 Online Storage Providers

With key boxes (Section 2.6), it is possible to use NoKey with only one device. But would it be possible to also use it with no devices at all? By integrating an online storage service such as Google Drive [24] or Dropbox [15] this would become possible. When connecting with such a service, all the replicated data could be stored in the cloud.

Users making use of this feature would now be able to access their passwords without carrying any of their devices as follows: First, navigate to NoKey's

website and start the web application. Then, connect to an online service by entering their password for that service. After that, all their encrypted passwords and key boxes would be available. To then unlock their passwords, they would have to open as many key boxes as the security level they were stored with.

This process requires that a user can remember at least three passwords. One for the online service and two for the key boxes to unlock a password group with security level two. This seems very inconvenient, but since using NoKey without any devices should not be a very common situation, it seems acceptable.

4.6 More Communication Channels

Right now, the only way two clients can communicate with each other is via the message forwarding server. To make NoKey less dependent on any server, it would be useful to add other means of communication.
One idea would be to integrate communication over bluetooth. This way, devices could directly communicate with each other without any middle man. However, since not many PCs support bluetooth, this channel would not be useful for many users.

Another interesting communication channel would be using WebRTC [2] (Web Real-Time Communications). This is a set of protocols that allow direct peer to peer communication between browsers.
Note that a server would still be necessary, but its job would be simplified even further. The only function that would still need a server would be the pairing process (Section 2.2). In addition to that, it could also act as a STUN (Session Traversal Utilities for NAT) [49] server that helps to establish peer to peer connections.

Chapter 5

Conclusion

With NoKey, we explored how a distributed password manager without a master password may look like in practice. With the convenience of not having to remember any passwords, we believe that this approach offers a great alternative to traditional password managers.

With traditional password managers, the security largely depends on the chosen master password. However, many people find it difficult to come up with strong passwords that are memorable at the same time. Our presented approach offers good security to all users, even for those that are unable to come up with good passwords.

NoKey is free and fully open source under the MIT license [28]. The source code is available at: `https://github.com/Zinggi/NoKey`.

Whether the approach taken by NoKey will become a popular alternative to traditional password managers will have to be seen, but we hope that more password managers will follow the presented approach in the future.

References

[2] Bernard Aboba et al. *WebRTC 1.0: Real-time Communication Between Browsers*. Candidate Recommendation. https://www.w3.org/TR/2017/CR-webrtc-20171102/. W3C, Nov. 2017.

[4] Eralp A Akkoyunlu, Kattamuri Ekanadham, and Richard V Huber. "Some constraints and tradeoffs in the design of network communications". In: *ACM SIGOPS Operating Systems Review*. Vol. 9. 5. ACM. 1975, p. 73.

[5] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic bookshelf, 2013.

[7] Elaine Barker. "NIST Special Publication 800-57 Part 1 Revision 4, Recommendation for Key Management Part 1: General". In: *NIST* (2016).

[8] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE. 2016, pp. 292–302.

[11] Mike Pietraszak (Microsoft Corporation). *Browser Extensions*. Draft Community Group Report. https://browserext.github.io/browserext/. W3C, 2017.

[12] Evan Czaplicki. "Elm: Concurrent frp for functional guis". In: *Senior thesis, Harvard University* (2012).

[13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[16] Manuel Eggimann and Christelle Gloor. "Convenient Password Manager, Group Project". In: (2017).

[26] Paul Hudak et al. "Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2". In: *SIGPLAN Not.* 27.5 (May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699. URL: http://doi.acm.org/10.1145/130697.130699.

[29] Burt Kaliski. "PKCS# 5: Password-based cryptography specification version 2.0". In: (2000).

[32] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. "HMAC: Keyed-hashing for message authentication". In: (1997).

[35] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[39] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 155–165.

[41] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[42] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[43] Marc Shapiro et al. “Conflict-free replicated data types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[44] Marc Shapiro et al. “Convergent and commutative replicated data types”. In: *Bulletin-European Association for Theoretical Computer Science* 104 (2011), pp. 67–88.

[46] Roy Thomas. “Fielding. Chapter 5: Representational state transfer (rest)”. In: *Architectural Styles and the Design of Network-based Software Architectures (Ph. D.)* (2000).

[47] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[48] Web Hypertext Application Technology Working Group (WHATWG). *Web sockets HTML Living Standard*. Tech. rep. https://html.spec.whatwg.org/multipage/websockets.html. WHATWG, May 2018.

[49] Dan Wing et al. “Session traversal utilities for NAT (STUN)”. In: (2008).

Web Links

[1] *1Password, homepage*. https://1password.com/. Accessed: June 7, 2018.

[3] *Akka Documentation, Distributed Data, Lightbend, Inc.* https://doc.akka.io/docs/akka/snapshot/distributed-data.html. Accessed: June 7, 2018.

[6] *Avatars, identicons, and hash visualization, Jussi Judin.* https://barro.github.io/2018/02/avatars-identicons-and-hash-visualization/. Accessed: June 7, 2018.

[9] *Browser Extension, Mozilla Developer Network, Mozilla.* https://developer.mozilla.org/en-US/Add-ons/WebExtensions. Accessed: June 7, 2018.

[10] *Building Web Apps in WebView, Android Developer Guide, Google.* https://developer.android.com/guide/webapps/webview. Accessed: June 7, 2018.

[14] *Dashlane, homepage*. https://www.dashlane.com/. Accessed: June 7, 2018.

[15] *Dropbox, cloud storage provider, Dropbox, Inc.* https://www.dropbox.com/. Accessed: June 7, 2018.

[17] *Elixir, a dynamic, functional language*. https://elixir-lang.org/. Accessed: June 7, 2018.

[18] *Elm, a purely functional language for the web*. http://elm-lang.org/. Accessed: June 7, 2018.

[19] *Elm package for hash icons, Florian Zinggeler*. http://package.elm-lang.org/packages/Zinggi/elm-hash-icon/latest. Accessed: June 7, 2018.

[20] *Enpass, homepage*. https://www.enpass.io/. Accessed: June 7, 2018.

[21] *F#, a functional-first programming language*. https://fsharp.org/. Accessed: June 7, 2018.

[22] *Font Awesome - the iconic SVG, font, and CSS toolkit, Fonticons, Inc.* https://fontawesome.com/. Accessed: June 7, 2018.

[23] *GitHub, homepage*. https://github.com/. Accessed: June 7, 2018.

[24] *Google Drive, cloud storage provider, Google*. https://www.google.com/drive/. Accessed: June 7, 2018.

[25] *Have I Been Pwned, Troy Hunt*. https://haveibeenpwned.com/. Accessed: June 7, 2018.

[27] *Identicons, GitHub.* https://blog.github.com/2013-08-14-identicons/. Accessed: June 7, 2018.

[28] Open Source Initiative et al. *The MIT license.* https://opensource.org/licenses/MIT. 2006.

[30] *KeePass, homepage.* https://keepass.info/. Accessed: June 7, 2018.

[31] *Kotlin, a statically typed language for the JVM, JetBrains.* https://kotlinlang.org/. Accessed: June 7, 2018.

[33] *LastPass, homepage.* https://www.lastpass.com/. Accessed: June 7, 2018.

[34] *LessPass, Stateless Password Manager.* https://lesspass.com/. Accessed: June 7, 2018.

[36] *novault, Stateless Password Manager and Brain Wallet.* https://github.com/novault/novault. Accessed: June 7, 2018.

[37] *Phoenix Framework, a web framework for Elixir.* http://phoenixframework.org/. Accessed: June 7, 2018.

[38] *Progressive Web Apps, Google.* https://developers.google.com/web/progressive-web-apps/. Accessed: June 7, 2018.

[40] *Riak, a distributed NoSQL database, Basho Technologies, Inc.* https://docs.basho.com/riak/kv/2.2.3/learn/concepts/crdts/. Accessed: June 7, 2018.

[45] *The Road to 2 Million Websocket Connections in Phoenix.* http://phoenixframework.org/blog/the-road-to-2-million-websocket-connections. Accessed: June 7, 2018.