# Horcrux, a Password Manager for Paranoids

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment
of the requirements for the degree

Master of Science

by

Haina Li

August 2017

# APPROVAL SHEET

This Thesis
is submitted in partial fulfillment of the requirements
for the degree of
Master of Science

Author Signature: ____________________

This Thesis has been read and approved by the examining committee:

Advisor: David Evans

Committee Member: Baishakhi Ray

Committee Member: Vicente Ordonez

Committee Member: Yanjun Qi

Committee Member: ____________________

Committee Member: ____________________

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

August 2017

UNIVERSITY OF VIRGINIA

# Horcrux, a Password Manager for Paranoids

by

Hannah Li

Advisor:

David Evans

A thesis submitted in partial fulfillment for the degree of Master of Science

in the
Security and Privacy Lab
School of Engineering and Applied Science

August 2017

# Declaration of Authorship

I, Hannah Li, declare that this thesis titled, 'Horcrux, a Password Manager for Paranoids' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________________

Date: ________________________________________

UNIVERSITY OF VIRGINIA

# *Abstract*

Security and Privacy Lab
School of Engineering and Applied Science

Master of Science

by Hannah Li

Vulnerabilities in password managers are unremitting because current designs provide large attack surfaces, both at the client and server. We describe and evaluate Horcrux, a password manager that is designed holistically to minimize and decentralize trust, while retaining the usability of a traditional password manager. The prototype Horcrux client, implemented as a Firefox add-on, is split into two components, with code that has access to the user's master's password and any key material isolated into a small auditable component, separate from the complexity of managing the user interface. Instead of exposing actual credentials to the DOM, a dummy username and password are autofilled by the untrusted component. The trusted component intercepts and modifies POST requests before they are encrypted and sent over the network. To avoid trusting a centralized store, stored credentials are secret-shared over multiple servers. To provide domain and username privacy, while maintaining resilience to off-line attacks on a compromised password store, we incorporate cuckoo hashing in a way that ensures an attacker cannot determine if a guessed master password is correct. Our approach only works for websites that do not manipulate entered credentials in the browser client, so we conducted a large-scale experiment that found the technique appears to be compatible with over 98% of tested login forms.

# *Acknowledgements*

This work was partially funded by an award from the National Science Foundation (1422332) and gifts from Amazon, Google, and Microsoft.

I am deeply grateful to my advisor, Professor David Evans, for his support and guidance. His dedication for his work and students help us to grow into successful scholars.

I owe my gratitude to my husband, Wenjiang Fan, for his support when times get tough.

# Contents

*For my grandparents…*

# Chapter 1

# Introduction

Users are frequently beseeched to come up with unique, strong passwords for every online account, but remembering more than a few high-entropy passwords is well beyond the capabilities of normal humans. Das reported that 43–51% of Internet users reuse the same password across multiple sites [1]. According to a recent survey of security experts, using a password manager was among the most widely-accepted recommendations for improving security [2].

Current password managers, however, do not provide adequate protection for paranoid users. Vulnerabilities are frequently discovered in both the client and server components, as revealed in several recent reports of compromises to major commercial password managers [3–6]. Password managers which store user credentials in cloud databases have been susceptible to theft and successful dictionary attacks on stolen encrypted data, according to recent news reports on LastPass [7] and KeePass [4]. Password manager critics often cite database theft as a compelling reason not to use password managers: "when password managers fail, they offer a one-stop destination for hackers to obtain all of a target's passwords" [4].

## 1.1 Contributions

We present the comprehensive design and evaluation of a password manager, Horcrux,[1], that provides a level of security and privacy well beyond what is achieved by current systems. Horcrux is designed to minimize exposure of secrets to a small, auditable

[1] A "Horcrux" is a dark, magical object in the *Harry Potter* book series in which a witch or wizard may hide a fragment of their soul. The antagonist of the series, Voldemort, uses multiple horcruxes to distribute the trust of his soul into multiple objects. Voldemort cannot die as long as at least one of his horcruxes is alive.

component. Although our design is intended for direct integration into a browser (see Section 7), we have demonstrated and evaluated its effectiveness by implementing a prototype as an open source Firefox add-on (Section 5).

Our server side design (Section 3.2) distributes trust by using secret sharing to store passwords across multiple hosts, and makes novel use of cuckoo hashing to provide user and domain privacy without enabling off-line attacks on password stores. Our client design isolates the component that has access to passwords from the rest of the password manager. Horcrux never exposes the credentials to the DOM, minimizing exposure of user credentials by replacing autofilled dummy credentials with real credentials in outgoing network traffic (Section 3.3). Ideas similar to the password swapping in intercepted outgoing network traffic we use have been proposed before [8], but not adopted by password managers due to usability and compatibility concerns. To evaluate the deployability of our design, we conducted a large-scale study of Alexa's top million websites. As reported in Section 6, we find that Horcrux appears to be compatible with 98% of the websites where a login form was found.

Security researchers have long advocated for minimizing trusted computing bases [9–11] and using privilege separation [12, 13], and secret-sharing is a well established technique [14]. The main contribution of this work is combining established techniques and our privacy-preserving credential storing scheme in a holistic way to solve an important security problem, and performing a comprehensive evaluation of both the security and compatibility of that design.

# Chapter 2

# Context

This section provides background on vulnerabilities in current password manager designs and implementations, and presents the threat model that drives our design decisions.

## 2.1 Password Manager Vulnerabilities

We divide password manager vulnerabilities into client-side vulnerabilities, where the adversary is able to compromise the client or its network connection, and server-side vulnerabilities, where the adversary compromises the password store.

### 2.1.1 Client Vulnerabilities

Major password managers autofill user credentials into authentication forms found on visited webpages. This exposes those credentials to the DOM, where they are visible to injected malicious scripts [8, 15, 16] that can read the form using `Element.value`. This returns the string currently present in an input field, even if it is a password type field that displays as asterisks. Any value filled in by the password manager or typed in by the user can be stolen by the script. The dynamic nature of JavaScript means functions used in the password manager code may be replaced by adversaries. Li et al. found that all three of the tested password managers that supported bookmarklets were vulnerable to these kinds of vulnerabilities [16]. Silver et al. examined the autofill policies of 15 password managers and found that all had followed some unsafe practices with the security of their autofill policies [15]. None of the password managers comprehensively checked whether the protocol and action of the login form are secure before autofilling the credentials, making the credentials susceptible to being sent to a different destination or through an insecure protocol (HTTP). This risk may be exacerbated by autofill policies that autofill without any user interaction required, allowing attackers to obtain credentials from a large number of domains through an access point.

Password manager clients are also vulnerable because of implementation bugs, particularly in complex URL parsing code. In 2016, Karlsson [17] detected a bug in the Lastpass client script code that treated the wrong part of the URL as the domain, which allowed an attacker to fool the LastPass browser extension into providing a user's credentials for any stored domain. This motivates our design to separate the complex UI code from the small trusted core.

### 2.1.2 Server Vulnerabilities

Password managers can store credentials either in the cloud or locally on the user's computer. Using local storage eliminates the need to trust an external provider, but means that it is now up to the user (who lacks the physical and technical resources of a cloud provider with a data center) to protect the store, and that there is no way to share credentials across multiple devices. It also means that an attacker who can compromise the user's device would now have access to both the password client and store. Although our design allows for the storage devices to include a mix of user-hosted or local stores, for nearly all users, we expect it is a better option to outsource storage to cloud providers.

Users who store their credentials in the cloud or user-hosted platforms inherit the risk of server-side compromises. Unlike a locally stored password database, an adversary could mount an attack on such servers from anywhere and attempt to steal their data, as happened to LastPass in 2015 [18] and OneLogin in 2017 [19]. These thefts typically expose encrypted versions of users passwords, URLs, and emails for all of their online accounts, which are open to thieves to attempt to decrypt offline. Notably, the LastPass server compromise included leakage of a cryptographic hash of many users masters passwords, which was used to decrypt their sensitive online account login information. Modern password managers seek to mitigate offline attacks by storing encrypted passwords using intentionally slow key derivation functions that amplify the cost of a dictionary attack. Currently, in 2017, it is much more common for password managers to never store the user's master password on any device, and instead rely on the key derivations from the user's master password and other local secrets to decrypt passwords on the servers.

## 2.2 Threat Model

Our focus is on mitigating the risks posed by motivated and capable adversaries who can inject scripts into trusted web pages, compromise the client's network access, and may be able to compromise a server database to acquire a full copy of its contents.

### 2.2.1 Client Side

We assume the adversary can inject scripts into any visited webpage, including pages delivered using HTTPS. This covers the possibility of script injection vulnerabilities in the trusted website, as well as wireless access point attacks, both of which we consider realistic threats.

We consider stronger client-side attackers out of scope, since they have so many other ways to victimize the user. An adversary who can compromise the client's browser to access its internal state (for example, by exploiting a memory corruption vulnerability in the browser to executed arbitrary code with a ROP attack) can extract secrets from anywhere in the browser's memory, and can generate authentic-looking dialog boxes to request the master password from the user. Similarly, an attacker with a root-level compromise of the client's system, can install a keylogger or alter the clients certificate store to spoof HTTPS connections to targeted sites. No password manager design can provide strong defenses to compromises at those levels. Hence, we consider the web browser and host operating system to be a trusted computing base.

We also do not address client attacks that exploit vulnerable pages on the target server. For example, if the server domain hosts a login with an open redirect or a XSS vulnerability where the attacker is able to adjust the URL path to send the credentials to a compromised page on the server's site that will resend the submitted password to another site owned by the attacker. The only way to prevent this type of attack would be through requiring the entire URL path to match the enrolled path, not just the domain. Enforcing a specific path may break many websites that dynamically generate their target URLs. In this respect, Horcrux is not more vulnerable than traditional password managers or manual input of credentials by a user.

We do not consider social engineering or interface spoofing attacks where victims would be tricked into entering their master password into a rogue dialog box or directly providing their unencrypted credentials to an adversary's site. The lack of trusted input paths in commonly-used computing systems is an important problem and serious threat, but outside the scope of this work. As discussed further in Section 4.1, our prototype implementation also assumes the user will not be tricked into installing a malicious add-on that can observe network traffic after the password manager add-on has inserted the real password (this is necessary for our prototype because of limitations in Firefox's extension mechanism, but would not be an issue for a password manager built into a browser or mobile OS).

### 2.2.2 Server Side

For the server side, we include the threat of full release of all data stored by a cloud server. This could happen as the result of a server vulnerability [20], insider attack from a cloud service employee, or the cloud service complying with a subpoena or national security letter. Hence, it is important that the data stored by the cloud servers is not vulnerable to an offline guessing attack on the master password. This leads to a design objective that everything encrypted with keys derived from the master password must be indistinguishable to an attacker so there is no way to determine if a guess is correct.

Finally, we assume the user should not have any intrinsic trust in the provider of the password manager. This means we want a design where the code that has access to sensitive data is as small and simple as possible, to make individual or third-party auditing realistic.

# Chapter 3

# Design

Our prototype password manager, Horcrux, is implemented as an open-source Firefox add-on [21]. (As discussed in Section 4.1, this is a temporary approach to support our experimentation, and it is not possible to provide sufficient security with the current extension mechanisms.) The repository is available at `https://github.com/HainaLi/horcrux_password_manager`.

## 3.1 Requirements

The driving motivation for our design is to provide strong security against both client and server vulnerabilities, while providing usability that is similar to current password managers. In particular, our design aims to:

- Minimize exposure of user credentials in both time and the amount of code they are visible to.
- Limit the size of the trusted component on the server to a small, auditable core.
- Provide resistance against server compromises by ensuring that even a full compromise of a single password store does not enable at attacker to conduct offline attacks on the user's master password.
- Provide functionality and usability similar to current password managers, including supporting single click logins for sites with stored credentials.

Next, we describe how the password stores are implemented. Section 3.3 describes the client.

## 3.2 Password Stores

For server-side storage of domain credentials, our goal is to ensure that an attacker who obtains a full copy of a single server's store cannot execute a successful attack to learn the user's credentials, or even the domains where the user has stored passwords. Thus, we need to store credentials at the server in a way that an off-line guessing attack is not possible. There are a small set of domains that a user is likely to have credentials for, and an attacker can probably guess domains like facebook.com and gmail.com that will be used by most users. Hence, it is essential that the passwords are stored in a way that does not enable an attacker with access to the store to determine if a guessed master password is correct.

Designing loss-resistant password vaults with such goals in mind has been studied notably by Bojinov et al. [22], who fabricated decoy password sets and by Juels and Ristenpart [23], who introduced the concept of honey encryption to yield plausible-looking but bogus plaintexts for every guess of the encryption key. These designs, however, have been constructed with specific data patterns (i.e. passwords or RSA keys) in mind, and are susceptible to attacks exploiting the differences in the generated distribution of passwords, which is a problem with all static Natural Language Encoders (NLE) [24].

In our case, a better solution is to take advantage of secret sharing since this means all of the actual entries are indistinguishable from randomness. All we need to hide is the locations of actual entries. Our solution is to adopt *cuckoo hashing* [25]. Although cuckoo hashing was not originally designed to support privacy, it can be adapted to meet our goals and provides a solution that is cost-effective for both storage and bandwidth.

In the original cuckoo hashing, a given data point, $x_i$, gets two possible positions, $h_1(x_i)$ in $T_1$ and $h_2(x_i)$ in $T_2$. The hash functions, $h_1$ and $h_2$, are assumed to behave like independent and random oracles for the possible keys in the table. During an insert operation, one empty position is chosen at random to store $x_i$. If both positions are occupied, it follows the cuckoo bird's approach of replacing the current resident with the new resident and repeating and bouncing between the two tables until an empty position is found or when the maximum number of iterations is reached. Pagh and Rodler set this maximum number at $C \log t$, where $C$ is a constant and $t$ is the number of data points. At this point, the entire table is rehashed by choosing a different $h_1$ and $h_2$. Multiple rehashing attempts may be necessary before all the data points find a spot. Successful cuckoo hashing achieves worst case constant lookup and deletion time and amortized constant time for insertions.

In our server design, we use a variant of cuckoo hashing that puts all data in a single table of size $p$, instead of using multiple independent tables. We use $n$ independent hash functions capable of mapping $x_i$ uniformly and randomly into any of the $p$ locations in the table. (Discussion on how parameters are selected is deferred to the end of this subsection.) For each entry, with cuckoo hashing the table can store at any one of (up to) $n$ possible locations:

$$\begin{aligned} h_1(d, e) &= H(\text{str}(1) \mid\mid e \mid\mid H(d)) \mod p \\ &\vdots \\ h_n(d, e) &= H(\text{str}(n) \mid\mid e \mid\mid H(d)) \mod p \end{aligned} \tag{3.1}$$

where $d$ is the domain, $e = \texttt{PBKDF2}(\textit{master\_password})$ is the encryption key, $p$ is the table size, and $H$ is a cryptographic hash function (our implementation uses SHA-256).

### 3.2.1 Account Entries

For each account, we store a value in JSON format containing shares of the id tag, username, username length, username tag, password, password length, password tag, and IV. The tags and IV are for AES-GCM encryption and decryption of the username and password. (The IV is incremented by 10 when encrypting and decrypting the password.) The data stored in the keystores are not the encrypted plaintext credentials, but a secret share for each of the $s$ keystores, so no information is disclosed in these values. These are padded to a fixed length prior to encryption so no information is disclosed. The username and password fields are padded up to a maximum length (our prototype uses a default maximum of 64 characters), and the length parameters indicate the real length of the credentials. The id tag is just the hash of the domain, $\textit{id\_tag} = H(d)$, with SHA-256 this is a 256-bit output. An id tag indicating an empty table key location is 256 bits of zeros.

### 3.2.2 Initial Table Setup

The first time the keystores are used, we prepopulate the all the tables with $p$ table key-value pairs that are meant to mask the rows holding real accounts after Horcrux is used. The pre-secret shared values for all the table values are hexadecimal strings of zeros of the length prior to secret sharing. We rely on secret sharing to produce the different shares stored in $p$ rows of the keystores. Using Amazon AWS DynamoDB API's `batchWriteItem` method, we could store up to 25 key-value pairs at a time. All users will start with a table that appears to be full of account entries, and there is no way to distinguish real from empty entries from a single keystore.

### 3.2.3 Item Lookup

To lookup a domain's account information, we simply calculate the table keys associated with the domain using Equation 3.1 and obtain $n$ table keys. Next, we query each of the $s$ keystores and obtain $n$ values, $v_{s,n}$, from each keystore. After combining the shares, we acquire $n$ values for each of the entries in the table (Note that only the username and passwords were encrypted and the rest do not need to be decrypted):

$$\begin{aligned} v_1 &= dec(\text{combineShares}(v_{1,1}, ..., v_{s,1}), e, IV_1, tag_1) \\ &\vdots \\ v_n &= dec(\text{combineShares}(v_{1,n}, e, ..., v_{s,n}), e, IV_n, tag_n) \end{aligned}$$

We then compare each of the combined id tags with the hash of the domain, picking the value containing the correct id tag.

### 3.2.4 Item Insertion

In our variant of cuckoo hashing, we have $n$ possible places for the credentials to a domain, $d$. We use cuckoo hashing's collision resolution technique, as described in Section 3.2, to find a "nest" for every credential by calculating the locations for the booted values from its id tag. Inserting new item is the main concern for user-perceived latency, which is discussed in Section 5. Deleting or updating an account, is simply a matter or writing shares of the new value (all zeros for deletion) into the stored location.

### 3.2.5 Parameters

We select the default parameters for our cuckoo hash table for a reasonable balance of storage costs and minimal probability of collisions. Each location in the table can only hold one item. With $n = 2$, the effective load factor is less than 0.5. At higher load factors, cuckoo hashing performance drastically decreases and frequently needs rehashing. With $n \geq 3$, the load factor increases substantially. The value $c^*$ is used to represent the load factor for which the probability that all $p \cdot c^*$ items can be placed in the table. Fountoulakis and Panagiotou proved that for $n = 5$, $c_5^* = 0.992$ [26], meaning that nearly the entire table can be filled before rehashing is needed. For performance reasons, we are also concerned with the expected number of collision resolutions needed.

Fountoulakis et al. analyzed the number of insertions needed for a cuckoo hash table, and proved a polylogarithmic bound on the number needed holds for all but negligible probability [27]. For our implementation, the key stores support batch read and right requests (up to 25 elements at a time for AWS' DynamoDB API), so the network cost of increasing the number of hash functions is low, and the local computation cost is also fairly low. Hence, we select $n = 5$.

A 2007 study [28] on the number of accounts owned by the typical user found that the average user as 6.5 passwords, which are shared between 3.9 different sites, and that each user has about 25 accounts. We reason that in 2017, the typical user has more accounts. As a result, we design Horcrux to support a default maximum capacity of 10,000 accounts, and use $p = 10079$ (the first prime number above 10,000), and $n = 5$. This puts our expected maximum random-walk insertion time at around 8 reassignments for a fully-loaded table. Frieze et al. [29] discussed that while breadth-first search gives constant *expected* time, it cannot guarantee sub-polynomial runtime for the insertion of each element. The amortized time complexities are unsuitable for applications that rely on cuckoo hashing to guarantee fast individual insertions. Therefore, we stick to the original cuckoo hashing random-walk insertion algorithm.

## 3.3 Client

The overriding designs goal of the client is to minimize exposure of user credentials to a small, auditable component. Hence, the client is divided into two components: a trusted *core* that needs access to the master password and sensitive credentials, and an untrusted *UI component* that manages interactions with the webpage but cannot access any Firefox API. The core and UI components are completely isolated from each other and can only communicate through the message-passing. For all sensitive computation, The core component creates a NodeJS subprocess which uses its `crypto` library for all cryptographic algorithms[30], as well as AWS and Azure APIs for making requests to keystores.

Next, we describe how to setup the client, and what happens when it is used in the brower to process a login request. Figure A.1 (in the Appendix) illustrates the interactions between the user, browser add-on, and keystores.

### 3.3.1 Setup

To use Horcrux users need to set up servers that store shares of the users passwords. Our prototype add-on prompts users to create accounts on AWS or Azure and create access keys with permissions to use their noSQL keystores (AWS DynamoDB or Azure Table Storage) in a specific datacenter region. More paranoid users will configure keystores themselves using trusted services and spread across jurisdictions. A user may also use their own servers as a keystore, any server that supports the appropriate keystore APIs can be used. Each keystore in initialized as a cuckoo hash table full with shares of empty values as described in Section 3.2.2.

The user is prompted to provide credentials obtained from AWS or Azure (their *accessKeyID* and *secretAccessKey*) for each keystore, and to create a master password. The core component derives a keystore authentication key from the master password (*MP*) using a Password Based Key Derivation Function (`PBKDF2`) over 100,000 iterations, following similar practices used by 1Password [31]. The keystore server credentials provided are encrypted using *AuthKey* using AES-256 with GCM mode in `base64` encoding, and stored along with the IV, tags, and PBKDF2 salt in a `config.json` file on the client's device.[1] The IV is newly generated for each encryption, and incremented by 10 for each access key or secret. This file is not accessible to webpages in the browser due to Firefox's isolation policies [32]. Although our design attempts to limit the effectiveness of guessing attacks on the master password, it is still important that users select a strong enough master password to resist at least on-line guessing attacks. Access credentials are randomly generated by AWS or Azure and their `base64` encoding prevents them from being susceptible to such dictionary attacks. On-line attacks are limited by the cloud servers limits on incorrect login attempts.

### 3.3.2 Initialization

When a user opens Firefox and begins the browser session, the core component checks for `config.json`, a file in the add-on containing the user's encrypted keystore credentials. If the encrypted credentials are not present, the setup steps are performed. Otherwise, the core component prompts the user for their master password. The encrypted credentials are read from `config.json` and stored as strings in the core component's private memory. The master password is used to derive the 512-bit *AuthKey* using `PBKDF2` over 100,000 iterations. The *AuthKey* is then used as the decryption key to an `AES-256` cipher, and the core component deciphers each encrypted credential. The *AuthKey* is kept in the core component memory for the duration of the session (it is kept until browser is closed) to avoid needing to repeatedly request the master password.

[1] For AWS keystores, the region in which each set of credentials is configured to use is stored unencrypted. Storing region information is necessary in order to use the AWS API, and encrypting the region name would present an opportunity for an adversary to perform an offline dictionary attack on the master password. We view the risk of leaking the keystore region to an adversary who acquires the encrypted configuration file as much less serious than any risk of enabling an off-line attack on the master password.

Next, the core component verifies the user's keystore credentials by initiating a connection to each keystore. If the request fails, the user is presumed to have given an incorrect master password, and the core component returns to prompting for the master password. There is no enforced limit on password attempts at the client side, as all verification is done online and subject to AWS or Azure policies for making many requests with incorrect credentials.

### 3.3.3 Login Form and Account Retrieval

When the browser loads a webpage, the UI component finds login forms on the page by searching for forms with an input `type=password` and uses heuristics to exclude registration forms based on the form title and input labels. Once a login form is found, then the UI component notifies the core component with the URL of the containing page. The core component proceeds to query to keystores to check for the presence of an account associated with the domain, as previous described in Section 3.2. For the protocol, we assume that each domain only has one account associated with it. See the discussion on supporting multiple usernames per domain in Section 7.

Although it would be simplest and most secure to wait until the user submits the form to start the process of obtaining and reconstructing the account credentials, we considered this unacceptable for user experience. If we fetch the password shares after the user clicks submit, the user would experience the delay of fetching and reconstructing shares as well as the usual server response latency. Further, due to the indeterministic nature of JavaScript, we cannot guarantee that the core component would receive the ”user clicked” signal from the UI component before the login traffic leaves the browser.

### 3.3.4 Swapping Credentials

After obtaining credentials for a login form, the core component sends dummy credentials to the UI component to be autofilled in the form, and starts actively listening to network traffic. The UI component autofills the fields in with dummy user and password strings by setting the elements' `value`. These dummy values are different for each user but the same for every website that the user visits. Hence, the actual credentials are never exposed to the DOM, but a user will receive visual feedback that stored credentials are available for the form. The idea of avoiding exposure of passwords to injected scripts by using dummy credentials to autofill forms and replacing them in network traffic was previously suggested by Stock and Johns [8].

The core component waits for the user to click submit, at which point all instances of the dummy username and password in the request traffic are intercepted swapped for the real credentials before TLS encryption. Before inserting the real credentials, the core component checks that the target domain matches the credentials' domain and that the request is secure (the target URL is HTTPS and the dummy username and password are not used as URL parameters in a GET request).

### 3.3.5 Enrolling New Accounts

If the user has not already registered with the domain, Horcrux generates a strong password using random bytes from NodeJS' crypto library (users may override password generation to provide a user-generated password manually). The generated password is prefixed with a fixed set of characters (`AaBa12#$`) which satisfy the majority of web account password creation policies.[2] The credentials are then stored using cuckoo hashing's insert algorithm, as described in Section 3.2.

[2] We have not focused on password generation, and appreciate that generating passwords that pass some site's password rules is a challenging (and annoying) problem, and no single password will satisfy all rules. The problem of generating good and permissable passwords is orthogonal to the password management issues that are our primary research focus.

# Chapter 4

# Security Analysis

Horcrux is designed to provide strong protection of user credentials against the realistic threat model described in Section 2.2. Here, we analyze how well it resists both client-side (Section 4.1) and server-side (Section 4.2) adversaries.

## 4.1 Client Security

Horcrux's overriding design goals are to minimize attack surface and decentralize trust as much as possible. To achieve this goal, Horcrux is split into two components, a trusted component that has access to the user's entered master password and secrets derived from it and obtained from the keystores using credentials obtained from the master password, and an untrusted component that manages the user interface and everything else that does not involve any sensitive data. The main strategies to minimize the client-side attack surface are to limit the window of opportunity for an attacker to steal the login credentials and to make the client code that has access to the master password as small and simple as possible so that it may be realistically audited. The entire code for the trusted component is around 800 non-comment lines of JavaScript.

### 4.1.1 Password Exposure

Because the password is never inserted into the DOM, it is not vulnerable to injected scripts. Password managers that reveal the real credentials earlier in the process are limited to checking the form action at the time of the autofill, so could be vulnerable to scripts that dynamically modify the form action after the password has been provided. Horcrux ensures that the password is only ever sent to the correct domain by checking the URL in the POST request in the intercepted network traffic. The request must use HTTPS, so this ensures (assuming the browser verifies HTTPS certificates and implements TLS correctly) that the real credentials will only ever be inserted into an outgoing network request to the intended domain.

### 4.1.2 Resisting Guessing Attacks on Master Password

An adversary who compromises the client's host may be able to obtain the `config.json` file that contains keystore credentials encrypted with a key derived from the user's master password. This file may also be exposed when a user moves it to setup a new device. As discussed in Section 3.3.1, Horcrux derives an encryption key from the master password using a password-based key Derivation Function (`PBKDF2`) over 100,000 iterations. This limits the number of guessing attempts an adversary could make, but a sufficiently resourceful adversary could still be able to execute an effective dictionary attack against a weak master password. Since the user-selected master password may be weak, it is important that there is no way for an adversary to do an off-line dictionary attack against it. There should be no way for the adversary to check if a guessed master password is correct without sending a request to an external service such as an attempt to authenticate with a keystore server. We ensure this by being careful to never encrypt anything with keys derived from the master password that is distinguishable from randomness to an off-line adversary. An adversary can only verify their guess on the master password by deriving the key, using it to decrypt access credentials, and performing an online query with the credentials. The access credentials are random bit strings, so an adversary cannot determine if a guess on the master password is successful without submitting those credentials to a guessed keystore server. This is slow and expensive, and also subject to methods the keystore server should use to limit the number of inauthentic requests attempted (i.e., "throttling"). Neither Amazon nor Microsoft makes these mechanisms public, but at worst, they are no more than the limits for regular requests. AWS limits the steady-state request rates to 1000 requests per second using the token bucket algorithm [33]; Azure limits read requests to 15,000 per hour and write requests to 1,200 per hour [34].

### 4.1.3 Security Limitations to a Firefox Add-on

Browsers add-ons naturally limit the attack surface by isolating the higher privileged core component from the UI component [35]. Because the UI component runs in an insecure environment, the web pages, they are exposed to more threats despite the fact that each script running on the page can only access its own variables. Cross-extension attacks [36] exploit add-ons that have not properly defined its namespace. When a malicious add-ons is using the same JavaScript namespace of a benign add-ons, it can access and modify all of the benign add-on's global variables, functions, and objects. Recently, LastPass reported that it had updated its non-mobile browser extensions to fix this vulnerability, which allowed a clever attacker to force the LastPass extension to reveal stored user data [37]. We are aware of the threat that cross-extension poses to Horcrux and avoid using global variables, functions, and objects.

As an add-on subjected to the limitations of Firefox's API, our prototype Horcrux implementation intercepts and changes the network traffic using the same methods available to other add-ons. Since Firefox does not have a specific resolution for conflicts between different add-ons, it is not determined which add-on will have the last opportunity to view and edit the request. Another add-on with traffic interception capabilities that executes after Horcrux, would be able to read the final version of the POST request which includes the real credentials. This is serious risk, but mitigated by the user interactions needed to install an add-on. In February 2015, Mozilla made it more difficult for attackers to run malicious add-ons by only allowing reviewed and signed add-ons to run on browsers [38]. This security feature, however, could be turned off by the user, and an insufficiently paranoid user could be tricked into installing a malicious add-on. Section 7 discusses ways Horcrux could be deployed.

## 4.2 Server Security

Horcrux is designed to avoid trusting any single keystore. Here we consider the risks if a single keystore is compromised, and if all of the user's keystores are compromised.

### 4.2.1 Single Keystore Compromise

If an adversary obtains the complete keystore database from one of the keystores (or several of the keystores, up to the secret sharing threshold), they learn no semantic information other than its size if the user's keystore has expanded beyond the default size. The secret-sharing mechanism means that a single share provides no semantic information. Up to the threshold limit number of keystores may be fully compromised without any loss of credentials. Because of the way we use cuckoo hashing and fill all entries of the table with shares which will appear indistinguishable to an adversary, there is no way for an adversary to tell with entries in the table correspond to real accounts. This prevents a guessing attack on master password with popular domains, since all possible guesses lead to an indistinguishable set of possible locations.

Our design does not hide the access pattern, however. This means an adversary who can observe unencrypted requests to the keystore over time (e.g., the keystore operator itself, or an adversary that compromises a keystore server without detection and maintains a monitor there) would be able to learn common patterns of requests. Such an adversary could perform a guessing attack on the master password with common domains, to look for sets of locations that match the requests. Hiding this access pattern would either require giving up on domain privacy (so the domain is no longer encrypted with the master password, but revealed in cleartext in the keystore request), or using expensive methods such as Oblivious RAM [39] to hide access patterns.

The secret sharing schemes we use are malleable, so an adversary who has complete access to a user's keystore could modify their passwords and credentials (with XOR-secret-sharing this is simple bit flipping; with Shamir-sharing it is more difficult to do in a predictable way). This could prevent users from being able to obtain their passwords, but is not a threat to confidentiality.

### 4.2.2 Multiple Keystore Compromise

If all the user's keystores are fully compromised, the adversary can reconstruct the users' encrypted passwords. This seems unlikely to happen from vulnerabilities or insider threats if the user has keystores operated by different cloud services, but may be a risk under subpoena threats (users paranoid about NSLs or subpoenas will want to choose the jurisdictions of their keystores accordingly). Each password is encrypted with the master-password derived key before sharing it; the adversary would still need to do a dictionary attack on the master password to obtain the user's password.

# Chapter 5

# Performance

In this section, we evaluate the performance of Horcrux, focusing on the latency that a user may experience with Horcrux' multi-keystore and secret sharing design.

### 5.0.1 Experiment Setup

We conduct both our microbenchmarks (timings for DynamoDB requests) and actual client latency on an EC2 c4.xlarge node in the Northern Virginia region to simulate user experience. The c4.xlarge nodes are equipped with 4 vCPUs, 7.5 GB of memory [40, 41]. For the microbenchmark tests in Table 5.1, we send individual read and write requests to keystores located in Northern Virginia, Oregon, Ireland, and Singapore. This models a user paranoid enough to want four keystores spread across multiple jurisdictions to resist state-level attacks. In reality, a paranoid user would also want to use different service providers to host the keystores, but for simplicity of our experiments we use AWS for all of them. Since the main issue is latency to the keystores, it should not have a significant impact on the results if the keystores were hosted by different providers, so long as they provide a similar batch request API as the one we use. Both the store and write experiments are taken sending an individual item to the keystores. For the amount of data that we send, a batch request of up to 25 items (the maximum allowed by DynamoDB) does not take noticably longer than an individual item request.

For the user experience tests (Table 5.2), we report the latency a user would experience when trying to retrieve credentials and enroll a new account. For credentials retrieval, the timer starts when a form is detected and ends when the password is reconstructed and Horcrux is listening for the submit click. This doesn't include the time required to detect the form, but that time mostly depends on the page load time which depends on the individual host. If the credentials are ready by the time the user clicks submit, then Horcrux will have little impact on end users. If not, the submit request would be delayed until the credentials are ready, and the delay may be noticable and annoying to users.

For enrolling an account, the timer starts when the user indicates that she wants to store account information and ends when the server responds with "write successful". This corresponds to the point when the credentials would be submitted to the account's host server. Cuckoo hashing inserts can sometimes take multiple read and write round trips to the keystores to find a placement. Here, we assume that each store succeeds on the first try. As discussed in Section 3.2.5, the system parameters are set so it should be very rare for multiple attempts to be needed.

### 5.0.2 Results

Table 5.1 presents the results from our microbenchmark latency tests. The duration for each read and write requests depends largely on the distance they are from the client. Since the password cannot be reconstructed until all shares are received (using a paranoid configuration where the secret-sharing threshold is set to the number of shares), the latency experienced by the user will depend on the latency to the furthest keystore.

Table 5.2 shows the average time in seconds that it takes for Horcrux to retrieve credential for a user with keystores on US coasts and one with globally-distributed keystores in the locations in Table 5.1. In addition to the retrieval layency, there is significant overhead caused by the Firefox extension and cryptographic computations. Enrolling account information takes almost twice as long as fetching an account. This makes sense because in cuckoo hashing, storing into the table requires two roundtrips to the server. A paranoid user who stores shares of his credentials around the world would have to wait approximately 1.6 seconds after the page loads be able to login. To register a new account with Horcrux, the user needs to wait 2.5 or more seconds, depending on how many spots need to be evicted.

We have not done any user testing yet to know how acceptable these results are (that is, how often does a user click submit to login within the credentials reconstruction time, and how noticeable and annoying is the additional delay when the client has to wait for credentials to be ready). Although we suspect these times are long enough to be noticeable for most users, we note that with the right UI display paranoid users may be willing to wait a second or two to perform a login, and there are many opportunities to improve performance to approach the keystore latencies in Table 5.1, especially for a deployment that is integrated into a browser rather than running as an add-on as our prototype.

| | Virginia | Oregon | Ireland | Singapore |
|---|---|---|---|---|
| Write (ms) | 64.3 (5.03) | 409 (21.3) | 361 (25.1) | 973 (45.2) |
| Read (ms) | 65.0 (5.16) | 309 (23.7) | 356 (20.2) | 927 (37.0) |

Table 5.1: Time in milliseconds for a write or read request from a node in US East (Northern Virginia). Results are averages over 10 requests, standard deviations in parentheses.

| | US (3) | World (4) |
|---|---|---|
| Retrieve Credentials (s) | 0.80 (0.034) | 1.59 (0.49) |
| Enroll Account (s) | 1.16 (0.04) | 2.49 (0.17) |

Table 5.2: Time in seconds that an user would experience. The US column is for a user with three keystores, 2 in Northern Virginia and 1 in Oregon, The World column is for the 4 keystores listed in Table 5.1). Results are averages over 10 requests, standard deviations in parentheses.

# Chapter 6

# Compatibility Testing

For the credential swapping method to work correctly, the dummy credentials must be visible in the outgoing request traffic. Client scripts could alter the values entered for the username and password in ways that would disrupt recognition. In this section, we describe the SwapScan tool we built to perform compatibility testing (Section 6.1), reports on how we tested Alexa's top million websites (Section 6.1.3), and present our compatibility results (Section 6.1.7). SwapScan uses heuristics to find login forms on web sites, and then automates the login process using the dummy username and password. For sites where a test login can be performed, SwapScan checks the request traffic for the dummy username and password strings. We found this test to be successful on 98% of the tested logins, which gives us a high confidence that Horcrux would work on a majority of the web. As a by-product of our compatibility test, we also found many insecure login form development practices, which are not directly relevant to password managers; we report on these findings in Appendix A.2.

## 6.1 SwapScan

SwapScan, our login automation tool, is built on top of OpenWPM, a fully-automated, open-source framework for large-scale web scanning [42]. OpenWMP provides scalability by wrapping Selenium instances in a driver that monitors their activity, ensuring that if the web page stalls or if one Selenium crashes, the test would continue. OpenWPM, which was originally intended to measure web privacy from third-party scripts, also includes a proxy and a series of hooks for data collection — two features that were essential for our experiments.

SwapScan starts by using OpenWPM to visit a URL from a list of URLs of the front page of websites, and then uses heuristics to scan the site to attempt to find, fill in, and submit a login form. Our approach is adapted from SSOScan [43], but is more complicated since we are looking for login forms instead of SSO buttons.

### 6.1.1 Finding Login Forms

After visiting the URL, SwapScan attempts to find a login form on the website. The strongest indicator of a potential authentication form is the presence of a HTML element with `type=password` in the form children. Once an authentication form is found, SwapScan determines whether the current form is for login or registration. If there are two password children in the form, then the form is likely a registration form. Other indicators used to separate registration forms from logins include keywords in the title of the page, keywords in the form attributes, and whether the form contains an input with registration topics (e.g., birthdate, security question). SwapScan determines the topic of an input box by matching regexes with element attribute values. If the candidate form is not a login, SwapScan will try other forms on the page.

If no suitable form is found on the current page, SwapScan uses heuristics to click on buttons likely to reveal forms until it either finds one or reaches the maximum number of attempts allowed. Candidate buttons are identified on the page and ranked by matching regular expressions such as `[Ll][Oo][Gg][liOo][Nn]` with each of the attribute values or the innerHTML of a *visible* node [43]. The likely candidates are ranked by the frequency of the matched regular expressions and the attribute the keywords are found in (e.g., the `innerText` of an element would be a better indicator of the purpose of the login). Invisible elements are not considered because a human user would not interact with them, and single-sign-on buttons are ruled out completely. If none of the attempts lead to a login form, SwapScan records that no login was found for this site and concludes this test.

### 6.1.2 Testing Submission

If a login form is found, it is autofilled with the dummy username and password and submitted. Outgoing traffic is observer to look for the dummy username and password. SwapScan also saves all outgoing traffic from the browser for analysis explained in Section 6.1.3.

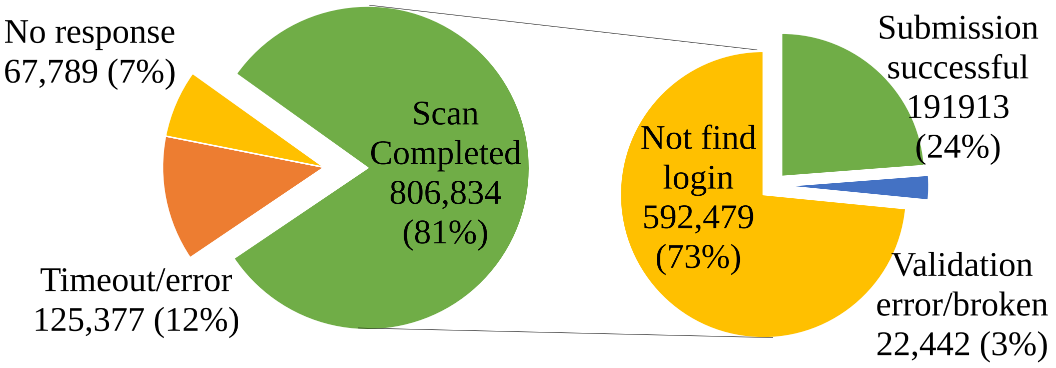

Figure 6.1: Overview of scan results

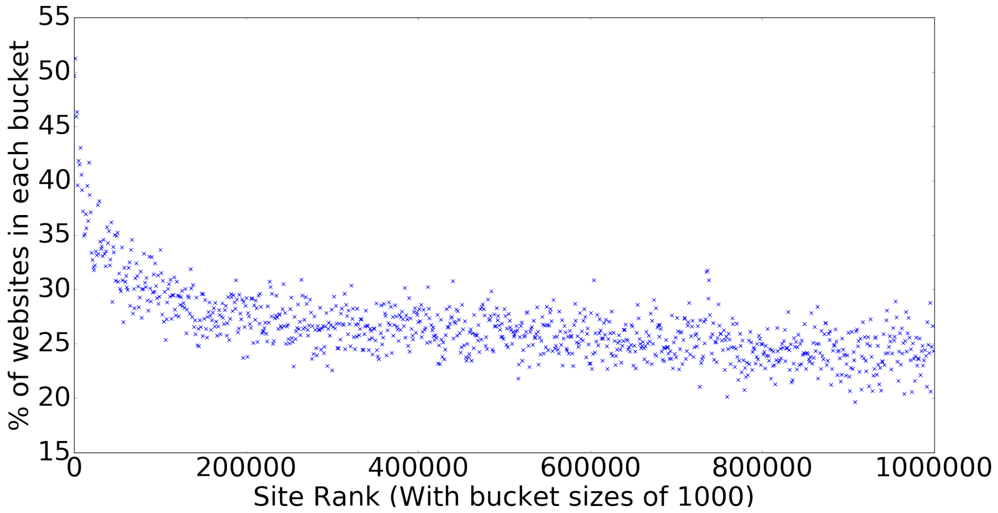

Figure 6.2: Percent of sites where login is found by popularity rank. Each bucket is for 1000 sites for which the scan completed.

### 6.1.3 Scanning

In February 2017, we used SwapScan to scan Alexa's top million websites. This took approximately 16 hours using 200 Amazon AWS c4.2large instances, each running 10 parallel scans, averaging around 2 minutes per website.

Figures 6.1 and 6.2 summarize the results of the scan. The key finding, discussed in Section 6.1.7, is that the password swapping approach works on the vast majority of websites (98% in our study). Here, we explain why some websites were not tested and the success rate for finding login forms among those that were tested.

### 6.1.4 Nonresponsive Websites

Out of the 1 million sites attempted, 932,211 (93%) responded with 2xx or 3xx response codes. This result is consistent with OpenWPM's scan result in January 2016, which found that 917,261 (92%) of the websites successfully loaded [42]

The scanner imposes a four-minute timeout for the total time to scan each website. Time outs and errors are affected by network speed and slow server response times (this primarily impacts websites hosted in other countries). SwapScan considers the scan incomplete if it has not completed the full scan itinerary at a maximum depth before the allotted four minutes expire. Errors and timeouts excluded 123,481 (12%) of the websites out from the scan.

### 6.1.5 No Discovered Login Forms

Among the remaining 806,834 sites, SwapScan found 214,355 (27%) websites with native login forms. Some sites do not provide any native login form, either because they do not have user accounts or only support single-sign-on authentication, which we do not include since there is no password to manage. There are several possible reasons, however, why SwapScan could fail to find a login form on a site that has one. The scanning heuristics assume login forms always include an element with `type=password`, but this does not hold for all login forms. For example, some sites use a two-step login process where the username is collected by the first form, and the password input is only revealed after the visitor submits a username. Our heuristics only consider English-language keywords, so may miss login forms labeled with other languages. Finally, we assume that an important login form would be accessible within a few clicks from the front page of a website.

Figure 6.2 shows the fraction of websites for which a login was found as the popularity of the site decreases. The success rate reflects the likelihood that more popular sites are more likely to provide user accounts, less likely to rely fully on single-sign-on authentication, and perhaps more likely to be designed in a way that makes the login form easier for an automated scanner to find. We are not able to distinguish among these (and other) possible reasons in our study, however.

Out of the 214,355 sites found with native logins, 22,442 (10%) sites' login did not respond with an outbound request upon submission. This may be due to broken logins or client-side validation errors. The remaining 191,913 sites (90%) were tested for compatibility with Horcrux, and these are the sites analyzed in Section 6.1.7.

### 6.1.6 Comparison with Previous Results

The largest previous scan of website login forms was Acker et al.'s 2017 study of the top 100,000 websites [44]. They reported finding 48,547 logins on those sites, where our scan only found 25,449. There are several reasons for the difference. We used heuristics to eliminate registration forms, ruling out forms with any indications of a registration-related input. Their scan counted any form with a `type=password` field as a login form, so counted some forms that were not considered login forms by our scan. Acker et al. counted 2,760 as not reachable within the 51% failed websites, while we considered 16,584 websites as not having completed the scan because they were either broken or did not respond fast enough to make our 4 minute threshold. Our scan required more interactions with the websites because we needed to observe the actual submission traffic, not just find the form, so more websites timed out. Further, we relied on the fact that the website was functioning quickly enough and that the login was reachable from the front page, whereas Acker et al.'s use of crawlers and search engines may have revealed login forms that were not directly accessible from the front page, which boosted their rate of finding logins.[1]

### 6.1.7 Compatibility Results

A site is counted as compatible if both the username and password dummy credentials are recognized in any request. SwapScan records all network traffic during a 3-second duration following a form submission. Of the websites where forms that were successfully found and submitted, 187,736 (98%) appeared to be compatible. On these sites, both the dummy username and password were observed in the intercepted traffic so the credential substitution approach should work. Note that while we define compatibility broadly, for security reasons our Horcrux password manager only reveals real credentials in HTTPS POST requests to the correct target domain (Section 4.1).

In cases where multiple requests are sent following a login form submission, we consider the request that contained the most information (e.g., we would favor a request containing both username and password over a request only containing the username). When we only see one of the username or the password in a request, we would check other requests from that website to see if the missing credential is in a different request.

We found some websites that were doing client-side encryption on credentials before sending them out to the server. We found 817 (<1%) of websites sending their requests out with a plaintext username and MD5-hashed password. These websites are counted as compatibale because Horcrux can look for the MD5-hashed dummy password in outgoing traffic and replace it with the MD5-hash of the real password.

[1] We considered using search queries for our scan also, but decided against this because of the unavailability of search engine APIs that allow enough queries for the larger scale of our scan.

### 6.1.8 Reasons for Incompatibility

Of the tested sites, 4177 (2%) appear to be incompatible with credential substitution. For 3266 (78%) of these, the dummy username was observed in intercepted traffic but not the dummy password. We manually examined a sample of ten of these sites, and found that nine of them were sending the dummy password transformed by some function other than MD5 hash. The remaining sampled site had an empty password field, which means that the site was either broken or did not send the password because client-side validation failed. The (probably misguided) inclination for sites to do client-side hashing of passwords explains why not observing the dummy password is the most common reason for incompatibility.

We also manually examined a sample of ten sites selected from the 881 incompatible sites where the password was observed but not the username. Three of these sent requests with empty username fields, one with a transformed username (but not a cryptographic hash). For the six remaining sites, we could not determine the reason why the dummy username had not appeared in the traffic.

One threat to validity is searching for dummy credentials that are not unique and long. We choose credentials with a goal of passing any client-side validation checks, so they could not be long random strings. Still, there is little risk of encountering the same string meant for another context, as our checking scope is limited to requests made in response to our form submissions.

SwapScan found that the password swapping would work on the preponderance (98%) of websites in the top million, and could be successfully deployed in a modern browser to work on today's web. If any popular sites are incompatible (i.e. www.sohu.com) , it may be necessary to include special rules for matching custom client-side password transformations. At worst, users can fall back to manual logins for incompatible sites.

# Chapter 7

# Discussion

Our goal in building the Horcrux prototype and conducting our experiments was to demonstrate the feasibility and practicality of a more secure password manager. Here, we discuss a few issues that we did not address in our prototype implementation but that would be important to handle before deployment.

## 7.1 Browser Integration

Our prototype was implemented as a Firefox add-on, but Mozilla is planning on deprecate the Firefox add-on SDK and migrate to WebExtension [45] with the goal of supporting cross-browser compatibility. We built our add-on using the soon-to-be legacy SDK because WebExtension does not currently have support for the majority of low-level APIs [46], some of which we depend on to manipulate HTTPS traffic. WebExtension's `webRequest` API only allows less powerful capabilities such as canceling and redirecting a request, modifying request and response headers, and supplying authentication credentials in-flight [47]. Our add-on needed the capacity to change `POST` request content. Google Chrome's `webRequest` API has similar limitations [48]. We recognize that this design choice is motivated by security—the browser vendors do not want to give third-party, untrusted browser add-ons too much power in case permissions are not reviewed carefully by the user or the developer [35]. In this regard, our design is best-suited for incorporating directly into the browser by changing how the default password manager autofills and stores passwords. Integrating Horcrux into a browser would also eliminate possible traffic visibility among multiple add-ons, which was discussed in Section 4.1. Providing support for mobile devices would also be important, but we have not yet considered that.

## 7.2 Supporting Multiple Accounts per Domain

The current Horcrux design does not support multiple usernames for one domain. The keystore scheme can be easily modified to support multiple usernames by including the username when calculating the table keys and picking a larger hash table size. In this design, however, Horcrux needs to know the full username prior to making the request to the keystores. The user would either need to enter the username manually, or Horcrux would need to maintain a local copy of domain and usernames, which defeats the goal of domain and username privacy. An alternative, would be to assume for most domains the user only has one username, but to store something special in the value entry for multi-user domains so the first request would return a list of the usernames instead of the password. Then, those names could be presented to the user to select from in the manager's interface. We believe multiple usernames could be handled without any security compromises.

## 7.3 Setting up New Devices

As discussed in Section 3.3.1, there are two ways to set up new devices to work with Horcrux securely transfer `config.json`, the file containing encrypted keystore credentials, to the new device or re-enter the database access stores into Horcrux. Neither options provide the convenience supported by commercial password managers because the user is not trusting a password manager to store his access keys. Upon entering their master password on the new device, the core component will use `PBKDF2` to derive the key needed to successfully decrypt the keystore credentials.

# Chapter 8

# Related Work

In this section, we discuss related work on secret sharing and data protection and authentication.

## 8.1 Secret Sharing and Data Protection

Although splitting secrets using XOR was known to ancient cryptographers, the first efficient threshold secret-sharing schemes were discovered separately by Shamir [14] and Blakeley [49] in the late 1970s. Our implementation uses Shamir's scheme.

Several prior works have used secret-sharing to protect data. Password-protected secret sharing (PPSS) [50] presents a Public Key Infrastructure (PKI) model for distributing the trust of sensitive data to $t+1$ hosts, with the ability to protect user data from being reconstructed by at most $t$ compromised hosts. PPSS mentions the utility of distributing user credentials in a password manager setting, though it does not investigate the application of its protocol in relation to common password manager database schemes, such as those found in Gasti et al. [20].

## 8.2 Password Manager Storage

Gasti notes the storage weaknesses of numerous password managers such as Chrome's unencrypted local SQLite database, 1Password's credential storage with AES-128 and CBC mode encryption, and KeePass' header-hashing data integrity checks [20]. The paper finds that many popular password managers are poorly suited against IND-CDBA (eavesdropping) or MAL- CDBA (data manipulation) schemes. Gasti categorizes the managers as (1) "those that can be assumed safe on an insecure storage medium" (2) “those that can be used if the underlying storage mechanism provides integrity and data authenticity” and (3) “those that can be used securely only if the underlying storage provides integrity, authenticity and secrecy” . The paper concludes that many password managers used database storage schemes that were vulnerable to read-only and read-write attacks.

## 8.3 Protecting Passwords from Scripts

Stock and Johns considered the problem of injected scripts stealing autofilled credentials from password forms, and proposed a client-side defense similar to the password swapping method used by Horcrux [8]. They implemented a prototype Firefox extension to perform password swapping, similar to what is done by our implementation, and conducted an evaluation of the top 4000 websites to determine how many included scripts that accessed password data, and manually inspected some of those scripts to understand whether password swapping was likely to work on those sites. Since their study required manual analysis, it could not scale to a large number of websites. Their implementation used local storage of passwords, and did not consider ways to reduce the size of the trusted client-side component.

# Chapter 9

# Conclusion

Passwords remain essential for web security, and improving the security of password management is an important goal for our community. We have demonstrated that it is possible for a password manager to minimize exposure of passwords to a few hundred lines of simple code, while using secret-sharing to mitigate the threat of server compromises.

## Appendix A

# Appendix

## A.1 Interactions between Components

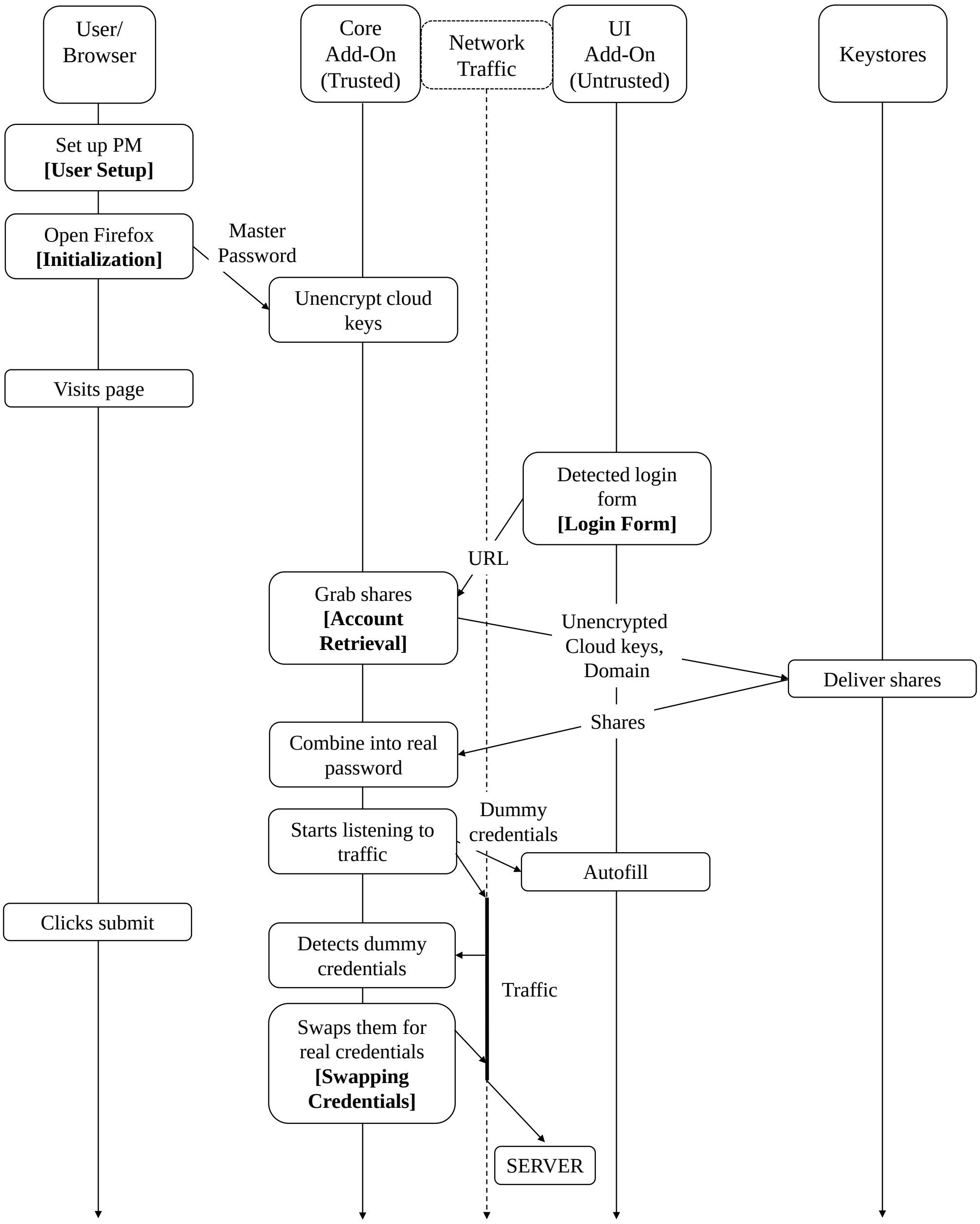
User/ Browser
Core Add-On (Trusted)
Network Traffic
UI Add-On (Untrusted)
Keystores
Set up PM **[User Setup]**
Open Firefox **[Initialization]**
Master Password
Unencrypt cloud keys
Visits page
Detected login form **[Login Form]**
URL
Grab shares **[Account Retrieval]**
Unencrypted Cloud keys, Domain
Deliver shares
Shares
Combine into real password
Dummy credentials
Starts listening to traffic
Autofill
Clicks submit
Detects dummy credentials
Traffic
Swaps them for real credentials **[Swapping Credentials]**
SERVER

FIGURE A.1: Horcrux protocol

## A.2 Other Results from Scanning Experiment

Although the goal of our study was to learn about the compatibility of password swapping across the web, as a byproduct of our scan we learned some other interesting things about web logins. Figure A.2 summarizes these results.

### A.2.1 Protocol

From an analysis of the URL destination of these login requests, we found that 102,948 (51%) of logins do not submit a request through HTTPS protocol, which exposes the username and password to man-in-the-middle attacks.

### A.2.2 AJAX

When a login form uses AJAX to grab the username and password from the form, a header named ”X-Requested-With” with a value ”XMLHTTPRequest” could be seen in the request. SwapScan used this property to identify 15,707 (8%) websites using AJAX requests, which indicates that JavaScript is used to submit the form data. This is a significant portion of websites that are compatible with Horcrux, but potentially incompatible with other secure login mechanisms, such as Secure Filling [15]. Submitting a form through JavaScript also means that the JavaScript may or may not consider the form action; even if a password manager ensure that the action is secure, it may not be actually used.

### A.2.3 Form action

Traditional password managers check the form action to ensure that the credentials are being submitted to the correct protocol and domain. Out of the login forms that were scanned, 18,628 (10%) percent of the forms did not have an action listed. For these sites, a non-Horcrux password manager would not be able to check the action to ensure that the credentials are headed to the correct domain. Since static analysis of JavaScript is difficult, the only way to ensure the correct destination for credentials is to dynamically check them, which Horcrux does naturally.

| Category | Percentage |
|---|---|
| HTTPS Protocol | 49% |
| HTTP Protocol | 51% |
| GET Request | 2% |
| POST Request | 98% |
| Other Methods | <1% |
| AJAX Login | 9% |
| Non-AJAX Login | 91% |
| Action Present | 90% |
| Action Not Present | 10% |

FIGURE A.2: Types of logins found by the scan (percentages are out of the number of completed form submissions).

# Bibliography

[1] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. In *22$^{th}$ Network and Distributed System Security Symposium*, 2014.

[2] Iulia Ion, Rob Reeder, and Sunny Consolvo. "... no one can hack my mind": Comparing Expert and Non-Expert Security Practices. In *11$^{th}$ Symposium on Usable Privacy and Security*, 2015.

[3] Swati Khandelwal. LastPass bug lets hackers steal all your passwords. http://thehackernews.com/2016/07/lastpass-password-manager.html, 2016. 2016-7-27.

[4] Dan Goodin. Hacking tool swipes encrypted credentials from password manager. http://arstechnica.com/security/2015/11/hacking-tool-swipes-encrypted-credentials-from-password-manager/, 2015. 2015-11-2.

[5] James Titcomb. Password manager 1Password criticised for leaking users bookmarks. http://www.telegraph.co.uk/technology/internet-security/11939920/Password-manager-1Password-criticised-for-leaking-users-bookmarks.html, 2015. 2015-10-19.

[6] Concerned LastPass User. PSA: LastPass Does Not Encrypt Everything In Your Vault. https://hackernoon.com/psa-lastpass-does-not-encrypt-everything-in-your-vault-8722d69b2032, 2017. 2017-1-18.

[7] Kate Vinton. Password Manager LastPass Hacked, Exposing Encrypted Master Passwords. http://www.forbes.com/sites/katevinton/2015/06/15/password-manager-lastpass-hacked-exposing-encrypted-master-passwords, 2015. 2015-6-15.

[8] Ben Stock and Martin Johns. Protecting Users Against XSS-based Password Manager Abuse. In *9$^{th}$ ACM Symposium on Information, Computer and Communications Security*, 2014.

[9] John Rushby. A Trusted Computing Base for Embedded Systems. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, 1984.

[10] Department of Defense. Trusted Computer System Evaluation Criteria, 1986.

[11] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *3$^{rd}$ ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[12] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *12$^{th}$ USENIX Security Symposium*, 2003.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *17$^{rd}$ USENIX Security Symposium*, 2008.

[14] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11), 1979.

[15] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password Managers: Attacks and Defenses. In *23$^{rd}$ USENIX Security Symposium*, 2014.

[16] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The Emperor's New Password Manager: Security Analysis of Web-based Password Managers. In *23$^{rd}$ USENIX Security Symposium*, 2014.

[17] Mathias Karlsson. How i made lastpass give me all your passwords. https://labs.detectify.com/2016/07/27/how-i-made-lastpass-give-me-all-your-passwords/, 2017. 2017-5-15.

[18] Dan Goodin. Hack of cloud-based LastPass exposes hashed master passwords. https://arstechnica.com/security/2015/06/hack-of-cloud-based-lastpass-exposes-encrypted-master-passwords/, 2015. 2015-6-15.

[19] Simon Sharwood. Identity Management Outfit OneLogin Sugar Coats Impact of Attack. *The Register*, 1 June 2017.

[20] Paolo Gasti and Kasper B Rasmussen. On the Security of Password Manager Database Formats. In *17$^{th}$ European Symposium on Research in Computer Security*, 2012.

[21] Mozilla Developer Network. Add-on SDK - Mozilla — MDN. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Content_scripts, 2016. 2016-2-10.

[22] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant Password Management. In *European Symposium on Research in Computer Security*, 2010.

[23] Ari Juels and Thomas Ristenpart. Honey Encryption: Security Beyond the Brute-Force Bound. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2014.

[24] Maximilian Golla, Benedict Beuscher, and Markus Dürmuth. On the Security of Cracking-Resistant Password Vaults. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[25] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[26] Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp Load Thresholds for Cuckoo Hashing. *Random Structures & Algorithms*, 41(3):306–333, 2012.

[27] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the Insertion Time of Cuckoo Hashing. *SIAM Journal on Computing*, 42(6):2156–2181, 2013.

[28] Dinei Florencio and Cormac Herley. A Large-scale Study of Web Password Habits. In *16$^{th}$ International Conference on World Wide Web*, 2007.

[29] Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An Analysis of Random-Walk Cuckoo Hashing. *SIAM Journal on Computing*, 40(2):291–308, 2011.

[30] NodeJS. Node.js v7.5.0 documentation. https://nodejs.org/api/crypto.html, 2017. 2017-2-10.

[31] 1Password. 1Password Whitepaper for Teams Security Design, 2015.

[32] Mozilla Developer Network. Introduction to the File and Directory Entries API. https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Introduction, 2016. 2016-9-27.

[33] AmazonAWS. Manage API Request Throttling. http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html, 2017. 2017-2-16.

[34] MSAzure. Throttling Resource Manager Requests. https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-manager-request-limits, 2017. 2017-1-11.

[35] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *18th Network and Distributed System Security Symposium*, 2010.

[36] Ahmet Salih Buyukkayhan, Kaan Onarlioglu, William Robertson, and Engin Kirda. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *23rd Network and Distributed System Security Symposium*, 2016.

[37] LastPass. Security Update for the LastPass Extension. https://blog.lastpass.com/2017/03/security-update-for-the-lastpass-extension.html/, 2017. 2017-5-20.

[38] Mozilla Developer Network. Review Policies - Mozilla — MDN. https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews, 2016. 2016-2-10.

[39] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3), 1996.

[40] Amazon. Amazon EC2 Instance Configuration. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html, 2017. 2017-5-15.

[41] Amazon. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/, 2017. 2017-5-15.

[42] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *23th ACM SIGSAC Conference on Computer Communications Security*, 2016.

[43] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *23rd USENIX Security Symposium*, 2014.

[44] Steven Acker, Daniel Hausknecht, and Andrei Sabelfeld. Measuring Login Webpage Security. In *32nd ACM Symposium on Applied Computing*, 2017.

[45] Mozilla Developer Network. WebExtensions - Mozilla — MDN. https://developer.mozilla.org/en-US/Add-ons/WebExtensions, 2016. 2016-2-10.

[46] Mozilla Developer Network. Comparison with the Add-on SDK - Mozilla - MozillaWiki. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Comparison_with_the_Add-on_SDK/, 2017. 2017-02-10.

[47] Mozilla Developer Network. webRequest - Mozilla — MDN. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest, 2016. 2016-2-10.

[48] Mozilla Developer Network. chrome.webRequest - GoogleChrome. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest, 2016. 2016-2-10.

[49] George Robert Blakley. Safeguarding Cryptographic Keys. In *AFIPS National Computer Conference*, 1979.

[50] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-Protected Secret Sharing. In *18th ACM conference on Computer and Communications Security*, 2011.