On The Security of Password Manager Database Formats

Paolo Gasti and Kasper B. Rasmussen

Computer Science Department
University of California, Irvine
{pgasti,kbrasmus}@ics.uci.edu

Abstract. Password managers are critical pieces of software relied upon by users to securely store valuable and sensitive information, from online banking passwords and login credentials to passport- and social security numbers. Surprisingly, there has been very little academic research on the security these applications provide.
This paper presents the first rigorous analysis of storage formats used by popular password managers. We define two realistic security models, designed to represent the capabilities of real-world adversaries. We then show how specific vulnerabilities in our models allow an adversary to implement practical attacks. Our analysis shows that most password manager database formats are broken even against weak adversaries.

1 Introduction

As the number of services offered on the Internet continues to increase, the number of passwords an average user is required to remember increases correspondingly, to the point where it is no longer feasible for most people to remember a new, strong password, for every account.

Users typically solve this problem in one of two ways. A common solution is to reuse the same password on many different websites [41]. This approach increases the potential damage if a password is stolen, cracked, or if a service that has access to it is compromised, since the attacker will be able to reuse it on all online services that share the password. Another approach is to use a "password manager" to store strong (random) passwords for each site. A password manager is a piece of software that requires a user to remember a single strong master password, used to decrypt the password manager's database. Remembering a single master password is much more feasible for users, who still get the security benefits of using a different password for each online service.

Using a password manager has other potential benefits. Full URLs (or at least domain names) of are typically stored alongside the corresponding passwords, and used to fill login form automatically. As such, users who rely on password managers are less susceptible to typo-squatting and phishing attacks [11, 20]: even if a user is directed to a malicious website that is designed to look identical to the website the user expects, the password manager will not log in automatically, providing an extra layer of protection.

Due to the sensitivity of the information typically stored in password databases, most password managers protect their content from unauthorized access. Database formats typically rely on encryption for data protection, where the encryption/decryption key is generated from a master password entered by the user.

This protection is also often designed to allow users to store the password manager database on untrusted storage. Several producers of password managers suggest storing password databases on USB sticks [35, 37, 40], in the cloud [1, 24] or on mobile devices [2, 4, 27], to allow convenient access to stored passwords. These storage options however, can also enable potential attackers to get hold of the database. Even when a password database is stored on a local hard drive, it may be possible for an attacker to obtain a copy through other means.

If the password manager database format is insecure, then all the advantages of a good password manager are wasted and the user may actually be less secure and more susceptible to, e.g., leakage of private information: privacy-conscious users may want to keep their browsing habits private and therefore delete cookies, history and cache often. On the contrary, password managers represent long-term storage facilities, storing (ideally) the only copy of passwords, and therefore their content is typically never deleted. If a password manager database leaks information about browsing habits, e.g., by storing URL's unencrypted, then clearing the cache and browsing history does not prevent an attacker from learning sensitive information.

In this paper we analyze the security provided by the database formats of some of the most poplar password managers in use at the moment. We define two different adversaries: a passive attacker that only tries to infer information from a password database, and an active attacker that modifies the content or meta-data. We highlight that using "industry standard practices", such as AES-CBC, is not enough to obtain a secure database format, even assuming the implementation of AES-CBC is correct. Note that we do not attempt to provide an exhaustive list of all possible attacks on all password managers. Rather, we model the security provided by common password manager database formats and provide examples of practical attacks.

The rest of this paper is organized as follows: Section 2 provides a brief overview of password managers used in our study; Section 3 introduces our system- and attacker models, while Section 4 analyzes the various database formats in such models. In Section 5 we discuss various general issues regarding database formats, and Section 6 covers related work. We conclude in Section 7.

2 Overview of Password Managers

Password managers differ in many aspects, including database format, functionality, availability of source code, supported platforms and access to cloud storage. Table 1 summarizes the main features of the password managers we considered. Some popular password managers invent their own database format, used exclusively by them. This is especially true for the password managers embedded in major browsers. We include these in our analysis because these

Password Manager	Database Format	Storage	Open Source	Platform	Browser Integration
Google Chrome [18]	Chrome	local/cloud	✓	Win/Mac/Linux	✓
Mozilla Firefox [30]	Firefox	local/cloud	✓	Win/Mac/Linux	✓
Internet Explorer [28]	MSIE	local	×	Win	✓
1Password [2]	1Password	local/cloud	×	Win/Mac	✓
KeePass 1.x [23]	KDB	local	✓	Win	×
KeePass 2.x [23]	KDB/KDBX4	local	✓	Win/Mono	×
KeePassDroid [4]	KDB/KDBX4	local	✓	Android	×
KyPass [27]	KDB/KDBX4	local	✓	iOS	×
PassDrop [36]	KDB/KDBX4	local	×	iOS	×
PINs [34]	PINs	local	×	Win	×
Password Safe [32]	PasswordSafe	local	✓	Win	×
Password Gorilla [13]	PasswordSafe	local	✓	Win/Mac/Linux	×
Roboform [39]	Roboform	local/cloud	×	Win/Mac/Linux	✓

Table 1. This table shows the password managers that where analysed in detail, along with the database format used by the software, the storage options available and the platforms supported. In addition we indicate whether the source code is available and whether the password manager is integrated with a browser.

password managers are widely used [16]. Several stand-alone password managers share the same database format, so even though each password manager provide a different experience to the user, the underlying storage format is the same.

In the rest of this paper focus solely on database formats and the security they provide, rather than on each password manager implementation. We assume that the password managers themselves correctly implement what the format specifies. As such, we do not consider, e.g., side channel attacks on the cryptographic primitives, or other attacks against the implementation. Rather we investigate the best possible security achievable given a specific storage format. For this reason our analysis focuses primarily on password managers that provide local storage, at least as an option. We leave the analysis of "cloud-only" password managers to future work.

We investigate nine popular password database formats. Three database formats used by in-browser password managers: Google Chrome, Mozilla Firefox and Microsoft Internet Explorer; and six formats used by a large number of stand-alone password managers: 1Password, KDB, KDBX4, PasswordSafe v3, PINs and RoboForm (refer to Table 1.)

3 Adversary and System Model

We consider two efficient adversaries: $\mathsf{Adv}_{\mathsf{r}}$ who has read access to the password database, and $\mathsf{Adv}_{\mathsf{rw}}$ who has read-write access. The goal of both adversaries is to extract as much information as possible and, for $\mathsf{Adv}_{\mathsf{rw}}$, to produce a database

that (1) was not created by the user and (2) once opened, will not trigger any warning or error message from the password manager. Clearly, $\mathsf{Adv}_{\mathsf{rw}}$ is strictly stronger than $\mathsf{Adv}_{\mathsf{r}}$: any attack that can be performed by $\mathsf{Adv}_{\mathsf{r}}$ is also available to $\mathsf{Adv}_{\mathsf{rw}}$. Both adversaries are allowed to gather multiple snapshots of the database at different points in time, in order to detect modifications in the database content.

We emphasize that our analysis does not rely on any modification of the user environment, e.g., tampering with the password manager code or installing a key logger. We focus solely on the security provided by the password manager databases, when the password manager software is operated in the "most secure" setting provided. We assume that users choose a strong, high-entropy, master password and that all underlying cryptographic algorithms (e.g., encryption, MAC, etc.) are properly implemented. Additionally, we assume that no additional mechanisms are in place to prevent file tampering. This allows us to compare the security offered by the database formats themselves.

3.1 Untrusted Storage

Consider an adversary who has full access to an encrypted password database, and is able to record different versions of it. Such an adversary can clearly use any of the recorded versions to replace the current database, as long as the master password did not change. This is essentially a replay attack that applies to both cloud-based- and local database formats.

The security notions we define below do not capture this attack, nor do we attempt to address it in any other way. In order to protect against it, a password manager must maintain some local state (e.g., a hash of the latest version of the encrypted database) on a trusted medium. As such, while this attack is clearly relevant when a password database is stored on the cloud or on an unattended USB drive, it cannot be mitigated by the database format alone. Therefore we exclude it from our analysis.

3.2 Security Definitions.

We model password managers by defining four algorithms that represent various functionalities: Setup, Create, Open and Valid. These algorithms are defined as follows:

Definition 1. *A password manager $\mathcal{PM}$ consists of the following efficient algorithms:* $\mathsf{Setup}(\cdot)$ *a probabilistic algorithm that, given a security parameter 1^κ, outputs a master password mp;* $\mathsf{Create}(\cdot,\cdot)$ *a probabilistic algorithm that, on input mp and a set of triples $RS = \{(r_1, n_1, v_1), \ldots, (r_\ell, n_\ell, v_\ell)\}$ (which represents a record-set), outputs a database DB;* $\mathsf{Open}(\cdot,\cdot)$ *a deterministic algorithm that, given mp and a database DB, outputs the record-set RS encoded in DB if RS is a valid record-set, i.e, there exist DB' such that $DB' \leftarrow \mathsf{Create}(mp, RS)$, and $\perp$ otherwise; and* $\mathsf{Valid}(\cdot,\cdot)$ *a deterministic algorithm that takes as input a master password mp and a database DB and returns 1 if $\mathsf{Open}(mp, DB) \neq \perp$.*

In practice, Valid is implemented by password managers within the Open functionality: if validation fails, the password manager returns an error rather than the database content.

We also define two new games, which we call *indistinguishability of databases* game (IND-CDBA) and *malleability of chosen database* game (MAL-CDBA). The former captures the capabilities of a realistic passive adversary, i.e., an adversary that has read-only access to a password database. The latter models and active adversary, which is allowed both read and write access to a password database.

Game 1 (Indistinguishability of databases game $\mathsf{IND\text{-}CDBA}_{\mathsf{Adv_r},\mathcal{PM}}(\kappa)$**)** *A challenger* Ch *running* $\mathcal{PM}$ *interacts with* $\mathsf{Adv_r}$ *in as follows:*

- Ch *runs* $mp \leftarrow \mathsf{Setup}(1^\kappa)$.
- $\mathsf{Adv_r}$ *outputs two record-sets* RS_0, RS_1
- Ch *selects a bit* b *uniformly at random and the database* $DB_b \leftarrow \mathsf{Create}(mp, RS_b)$ *is returned to* $\mathsf{Adv_r}$.
- $\mathsf{Adv_r}$ *eventually outputs bit* b'*; the game outputs 1 iff* $b = b'$.

We say that $\mathsf{Adv_r}$ *wins* the IND-CDBA game if it can cause it to output 1.

Definition 2 (IND-CDBA security). *A password manager* $\mathcal{PM} = (\mathsf{Setup}, \mathsf{Create}, \mathsf{Valid}, \mathsf{Open})$ *is* IND-CDBA*-secure if there exists a negligible function* negl *such that, for any probabilistic polynomial time adversary* $\mathsf{Adv_r}$*, we have that* $\Pr[\mathsf{IND\text{-}CDBA}_{\mathsf{Adv_r},\mathcal{PM}}(\kappa) = 1] \leq 1/2 + \mathsf{negl}(\kappa)$.

For most database formats an attacker can trivially win the IND-CDBA game by submitting two record-sets of different sizes. In practice, this corresponds to the fact that the size of the database file is often roughly proportional to the number of records in the database and therefore an adversary may be able to infer information by simply observing the size of an encrypted database. While we do consider this a valid attack, we ignore it in the vulnerability analysis. Database formats that are *only* vulnerable to this attack will be considered secure.

Appendix A shows the relationship between IND-CPA and IND-CDBA. In particular, it shows that IND-CPA-security implies IND-CDBA-security.

Game 2 (Malleability of chosen database game $\mathsf{MAL\text{-}CDBA}_{\mathsf{Adv_{rw}},\mathcal{PM}}(\kappa)$**)** *A challenger* Ch *running* $\mathcal{PM}$ *interacts with* $\mathsf{Adv_{rw}}$ *in the following way:*

- Ch *runs* $mp \leftarrow \mathsf{Setup}(1^\kappa)$.
- $\mathsf{Adv_{rw}}$ *adaptively outputs* n *record-sets* RS_i *and receives, from* Ch*, the corresponding databases* $DB_i \leftarrow \mathsf{Create}(mp, RS_i)$.
- $\mathsf{Adv_{rw}}$ *eventually outputs* DB'*; the game outputs 1 iff* $\mathsf{Valid}(DB') = 1$ *and* $DB' \neq DB_i$ *for* $i \leq n$.

We say that $\mathsf{Adv_{rw}}$ *wins* the MAL-CDBA game if it can cause it to output 1.

Definition 3 (MAL-CDBA security). *A password manager* $\mathcal{PM} = (\mathsf{Setup}, \mathsf{Create}, \mathsf{Valid}, \mathsf{Open})$ *is* MAL-CDBA*-secure if there exists a negligible function* negl *such that, for any probabilistic polynomial time adversary* $\mathsf{Adv_{rw}}$*, we have that* $\Pr[\mathsf{MAL\text{-}CDBA}_{\mathsf{Adv_{rw}},\mathcal{PM}}(\kappa) = 1] \leq \mathsf{negl}(\kappa)$.

Our definition of MAL-CDBA security is equivalent to the notion of "existential unforgeability" of ciphertexts, introduced in [22]. As shown in the same paper, this security notion along with IND-CPA security implies IND-CCA security.

"Integrity of ciphertexts" [6] (also known as INT-CTXT security) is a related security notion. In particular, the main difference between MAL-CDBA and INT-CTXT is that an adversary for INT-CTXT is also given access to the $\mathsf{Verify}(mp, \cdot)$ oracle.

We argue that MAL-CDBA security (together with IND-CDBA security) is an appropriate security notion for a password manager database format in practice. Consider a database format that is not MAL-CDBA-secure, i.e., where $\mathsf{Adv}_{\mathsf{rw}}$ can compute the encryption of a record-set of its choice, and produce the corresponding valid output DB'. This format would be vulnerable to the following four-step attack:

> (1) $\mathsf{Adv}_{\mathsf{rw}}$ replaces Alice's password database DB with a new database DB' containing the login credentials for an `amazon.com` account created by $\mathsf{Adv}_{\mathsf{rw}}$. (2) $\mathsf{Adv}_{\mathsf{rw}}$ now induces Alice to go to `amazon.com`, at which point the password manager automatically logs into the account created by $\mathsf{Adv}_{\mathsf{rw}}$. (3) Alice buys something; during checkout, Alice is requested to add her credit card to the account; since she trusts `amazon.com`, she complies. (4) $\mathsf{Adv}_{\mathsf{rw}}$ now replaces $DB'$with Alice's original password database.

$\mathsf{Adv}_{\mathsf{rw}}$ is now in possession of an account which can be used to purchase goods on Alice's behalf. It is very hard for Alice to detect this attack; she does not receive any warning from her password manager or from `amazon.com`, since the database is well formed and the login information corresponds to an existing account. Additionally, SSL/TLS does not help since Alice is communicating with `amazon.com`. Furthermore, Alice may not even be able to find out which username was used in the maliciously crafted account after the adversary restores her original password database.

4 Database Format Vulnerabilities

We now present our analysis, which includes several database formats currently in use by stand-alone and browser-based password managers. For each format, we provide a short description of the relevant features and analyze its security with respect to the security model defined in Section 3. If the format allows for different levels of security, we analyze the most secure configuration.

4.1 Google Chrome

Format Description. Google Chrome stores usernames and passwords in an SQLite database file in the user profile directory. This database provides neither secrecy nor integrity.

Google Chrome can optionally store all browser preferences (including passwords) on Google's servers to allow synchronization between different devices. Chrome's support pages claim that passwords are stored in encrypted form on Google's servers [19].

Security Analysis. Any user with access to the database file can recover all its content and make arbitrary modifications. As such, users cannot rely on Chrome's password manager for integrity or secrecy of their data, and should implement additional security layers around it.

4.2 Mozilla Firefox

Format Description. Mozilla Firefox stores login data in an SQLite database. Users can specify an (optional) master password that is used to encrypt the database content. URLs are always stored unencrypted regardless of whether a master password is used or not.

Since the database is part of Firefox' user profile, it can be automatically synchronized across multiple devices, either through Firefox Sync [31], manually (e.g., using rsync [26]), or stored on a USB stick and used on different computers.

Security Analysis. Firefox does not provide an effective protection against $\mathsf{Adv_r}$. In order to win in the IND-CDBA game, $\mathsf{Adv_r}$ creates two same-size record-sets RS_0, RS_1 which differ in at least one URL field. The encrypted database DB_b can be immediately identified since URLs are not concealed. In practice that means that an attacker can learn a considerable amount of information, such as the websites in which the user has password-protected access, and it can mount effecting phishing attacks based on user information. Moreover, given two different versions of the same database, the attacker can identify which entries have been modified and their corresponding domain name.

Similarly, given any non-empty database DB an active adversary $\mathsf{Adv_{rw}}$ can trivially win the MAL-CDBA game by building DB' from DB replacing one or more URLs with a different valid URL. Since the entries are not integrity protected, Firefox cannot detect such an attack. This can be used to mount a very effective man-in-the-middle attack by replacing legitimate domain names with fraudulent ones controlled by $\mathsf{Adv_{rw}}$. In this way, the password manager will automatically submit sensitive information to an adversary-controlled website. The attack is even more effective if $\mathsf{Adv_{rw}}$ can also modify Firefox' bookmark database, which is stored in the profile alongside the password database.

4.3 Microsoft Internet Explorer

Format Description. Internet Explorer stores usernames and passwords in the registry. Each record is stored as a separate registry entry and encrypted using the system login credentials. When a user fills-in a password form at address url, Internet Explorer computes $h = \text{SHA-1}(url)$ (where and url uses the unicode character set) and encrypts username and password as

$c = E_k(\texttt{metadata} \parallel \texttt{username} \parallel \texttt{0x00} \parallel \texttt{password} \parallel \texttt{0x00})$, where `metadata` contains additional information such as the size of encrypted elements.

The encryption is performed using the `CryptProtectData` [29] system call, which uses Triple-DES in CBC mode and a hash-based MAC. k is derived from (1) a random salt (also stored in the ciphertext), (2) url and (3) the Windows login credential for the current user. Finally, Internet Explorer creates a new registry entry with key h and value c.

The security of Internet Explorer's password manager depends on the strength of the user account password. As such, accounts with no password provide no protection of the password database.

Security Analysis. Internet explorer is not secure against $\mathsf{Adv_r}$. Similarly to Firefox, $\mathsf{Adv_r}$ wins the IND-CDBA game by building two same-size record-sets RS_0, RS_1 which differ in at least one URL.

Say record rec with URL url is in RS_0 but not in RS_1. $\mathsf{Adv_r}$ can immediately recognize which record-set corresponds to the challenge DB_b by computing $h =$ SHA-1(url) and verifying whether h is in DB_b.

In practice, a passive adversary can use Internet Explorer's password database to determine whether a user has visited a particular web page and entered his username/password, even if the user deletes his browsing history and cache.

Assuming that `CryptProtectData` uses a secure MAC, an active adversary cannot alter password entries. However, $\mathsf{Adv_{rw}}$ can delete password entries by removing the corresponding registry entry, and as such $\mathsf{Adv_{rw}}$ can easily win the MAL-CDBA game.

4.4 1Password

Format Description. 1Password stores its database in multiple files. Each file contains a database entry, stored in JavaScript Object Notation (JSON). Entries are listed in an index file called "content.js".

1Password allows users to select a different "security level" for each record [3]. The lowest security level corresponds to unencrypted entries, while the highest level means that sensitive fields, such as username and password, are encrypted with a key derived from the user's master password. Regardless of the security level, some fields, e.g., the title of an entry, are never encrypted. We analyze the protection offered by the highest security level.

The encryption scheme used is AES-128 in CBC mode. Neither the records nor the index file are integrity protected. As a result, database corruption is only detected when the JSON parser fails to process the database.

Security Analysis. 1Password's database format is affected by vulnerabilities that give adversaries a non-negligible advantage in both the IND-CDBA and MAL-CDBA games.

$\mathsf{Adv_r}$ can win IND-CDBA with probability 1 as follows: $\mathsf{Adv_r}$ builds two same-size record-sets RS_0, RS_1 such that there exist two records r_0, r_1 from RS_0 and RS_1 respectively, which differ in at least one of the following fields: `title`,

`location`, `locationKey`, `createdAt`, `updatedAt` or `typeName`. These fields correspond to: the title of the record, the record URL, the URL used by the browser plugin to perform auto-complete, the time of creation and last update and the type of record (e.g., web form, protected note, credit card information). Since these fields are never encrypted, $\mathsf{Adv_r}$ can trivially determine bit b by testing which record belongs to DB_b. In practice this means that an adversary with access to a 1Password database can read these fields and thus gather sensitive information about the user's browsing habits.

$\mathsf{Adv_{rw}}$ can win the MAL-CDBA game with probability 1 as follows. $\mathsf{Adv_{rw}}$ selects an arbitrary record-set RS and receives the corresponding database DB. Then, $\mathsf{Adv_{rw}}$ can (1) alter any of the fields listed above, and/or (2) remove any entry by deleting the corresponding database file and altering the "content.js" index file correspondingly. In general, as long as the database is still composed of a set of correct JSON strings, 1Password will not show any warning. In practice, this means that an adversary can mount phishing attacks by replacing a legitimate URL with one pointing to an adversary-controlled website.

Additionally, if $\mathsf{Adv_{rw}}$ outputs at least two record-sets, say $RS \neq RS'$ and receives the corresponding databases DB, DB' in the MAL-CDBA game, it can construct DB'' selecting records from both DB and DB'. This allows an adversary, among other things, to replace individual records in a database with older versions.

4.5 KDB (aka KeePass 1.x)

Format Description. The KDB database is composed of a single file, divided in two sections: an unencrypted header (hdr) and an encrypted body (bdy). bdy stores the encryption of the various database entries. hdr contains, among other things, the number of groups and entries in the database and the hash of bdy before encryption [23]. This hash is computed every time the database is modified, and is used to check integrity. After decryption, the password manager verifies that the computed plaintext hashes to the same value stored in hdr. If this check fails, the application reports that either the database is corrupted or the master password entered by the user is incorrect.

Security analysis. Given a database DB, the hash stored (unencrypted) in hdr is computed deterministically from the record-set RS encoded in DB. This allows an adversary $\mathsf{Adv_r}$ to win the IND-CDBA game with probability 1 as follows. $\mathsf{Adv_r}$ selects two same-size record-sets $RS_0 \neq RS_1$ and computes their hash $h_i = H(RS_i)$. Once it receives a challenge database DB_b, $\mathsf{Adv_r}$ checks whether the header of DB_b contains h_0 or h_1 and outputs its choice for b' accordingly.

In practice, given two databases, this allows $\mathsf{Adv_r}$ to determine whether their content is identical even if their corresponding ciphertexts are different. Also, assuming that the record-set encrypted in a database has lower entropy than the database master password, $\mathsf{Adv_r}$ can recover the content of the record-set by simply making a guess and comparing it against the hash value in hdr. In other words, the complexity of breaking the database is a function of $\min(\eta_{mp}, \eta_{RS})$ –

where η_{mp} is the entropy of the master password and η_{RS} is the entropy of the record-set – rather than just a function of the master password.

hdr is not authenticated and, as such, is susceptible to malicious modifications. This can be used by $\mathsf{Adv}_{\mathsf{rw}}$ to win the MAL-CDBA game with probability 1 by selecting a challenge record-set RS which contains one or more entries. When $\mathsf{Adv}_{\mathsf{rw}}$ receives DB he changes the value corresponding to the number of entries (stored in *hdr*) to a smaller number. Since *bdy* is not altered, the hash verification does not fail. However, the record-set has been altered since the number of entries shown in the password manager is now the one chosen by $\mathsf{Adv}_{\mathsf{rw}}$.

We verified that the latest version of KeePassX (0.4.3) is susceptible to this attack. Moreover, if the victim makes any change in the modified database, KeePassX stores only the entries displayed. This can lead to silent (undetected) corruption of the database.

4.6 KDBX4 (aka KeePass 2.x)

Format Description. The KDBX4 database format is composed of a single password-protected file, divided in two sections: an unencrypted header (*hdr*) and a main encrypted body (*bdy*). *hdr* contains several fields, including `mseed` and `tseed` (used to compute the encryption/decryption key from the user-provided password), `IV`, `pskey` and `ssbytes`, used for secrecy and integrity protection as detailed below.

bdy contains the database records encoded as a single XML string, optionally compressed using the gzip algorithm [17] before encryption. *bdy* is encrypted using AES-256 in CBC mode, although Twofish is also available. The first 32 bytes of *bdy* contains the encryption of the `ssbytes` field in order to efficiently verify whether the provided master password is correct. The next 32 bytes of the body contain the hash of the (possibly gzip-compressed) XML string representing the various entries. This hash is used to detect modifications in the database.

In addition, all passwords in the XML string are XOR-ed with a pseudo-random string, computed using Salsa20 [7]. Every time the database is saved, a random 256-bit key k is generated and stored unencrypted in the `pskey` field; each password pwd_i is then encoded as $s_i = pwd_i \oplus \text{Salsa20}(k, IV)$ using a fixed value IV. Each pwd_i uses a different portion of the keystream generated by Salsa20. Passwords are recovered as $pwd_i = s_i \oplus \text{Salsa20}(k, IV)$.

Security Analysis. KDBX4 fixes some of the weaknesses of KDB. *hdr* does not store the (unauthenticated) number of entries, therefore an adversary cannot alter this value to remove content from the password database. Also, the hash of the unencrypted record-set is now stored in encrypted form. This prevents an adversary from verifying its guesses on the database content and from determining whether two encrypted databases carry the same content. More generally, $\mathsf{Adv}_{\mathsf{r}}$ cannot mount any successful attack on a KDBX4 database except with negligible probability. Due to lack of space, we omit the proof of IND-CDBA security for KDBX4, which is available in the extended version of this paper [15].

Unfortunately, this format introduces new vulnerabilities. Similarly to KDB, the main problem of this format is the lack of authentication of hdr. As such, is it susceptible to modifications. In particular, $\mathsf{Adv_{rw}}$ can win the MAL-CDBA game with probability 1 as follows. $\mathsf{Adv_{rw}}$ outputs a challenge record-set RS. Then, after receiving the corresponding database DB, it replaces the value stored in the `pskey` field of hdr with an arbitrary 256-bit string and outputs that as DB'. This modification is not detectable by the password manager, i.e., $\mathsf{Valid}(DB') = 1$, since the integrity check on the records is performed *before* the XOR with the output of Salsa20. However, a different `pskey` value will cause all passwords to appear as pseudo-random data after the decoding process.

It is impossible to recover from this attack unless it is detected immediately, i.e., before the user applies any modification to the record-set. The only way to recover the database content is to restore the original `pskey` value. However, this value is replaced with a fresh one, and all passwords are "re-scrambled" accordingly, each time the database is modified and saved. For this reason if a user alters, and then saves, a corrupted database, all passwords previously affected by the attack are lost forever.

This attack highlights a remarkable design flaw. Even an accidental bit-flip in the `pskey` field, e.g., due to a transmission error, cannot be detected, and leads to complete corruption of the database. Such corruption is unlikely to be immediately detected by users, who may subsequently add new entries. Over time, the database will be composed of both correct and corrupted entries, making it difficult to reconstruct the damaged records from a backup.

As an extension to the previous attack, $\mathsf{Adv_{rw}}$ can alter `pskey` in such a way that an arbitrary (small) number of bits of the first password(s) in the database are not altered. To do that $\mathsf{Adv_{rw}}$ computes a value k' such that the first n bits of Salsa20(k, IV) are equal to the first n bits of Salsa20(k', IV). Then $\mathsf{Adv_{rw}}$ stores k' in `pskey`. k' can be computed in exponential time in n, and therefore is practical only when n is small. As a proof of concept, we developed an application that implements such attack. The application is available upon request.

Finally, $\mathsf{Adv_{rw}}$ can also win the MAL-CDBA game as follows. Given an arbitrary database DB, $\mathsf{Adv_{rw}}$ flips a bit in the first 16 bytes of `ssbytes`, and then flips the corresponding bit in the `IV` field of hdr to create DB'. The password manager cannot detect the change, i.e., $\mathsf{Valid}(DB') = 1$, since flipping a bit in `IV` causes the corresponding bit in the first block of plaintext to be flipped as well (using CBC-mode), and no additional side effect. Since the first block of plaintext corresponds to the first 16 bytes of `ssbytes`, the modification produces a new correct database. This allows $\mathsf{Adv_{rw}}$, given a database DB, to produce up to $2^{128} - 1$ different databases $DB'_1, \ldots, DB'_{2^{128}-1}$ containing the same record-set as DB.

4.7 PINs

Format Description. The PINs database is stored in a single file, and encrypted using AES in CBC mode. Records are encrypted separately and stored one record per line, using hexadecimal representation written as ASCII text.

The first line of each database defines the version of the software used to create the database, while the second line contains the encryption of the string "`#TEST VERIFY`" followed by a variable number of up to fifty random bytes. This is used to verify that the user-provided master password is correct. After deriving the database encryption/decryption key from the user's input, PINs decrypts the second line and determines whether the result corresponds to the expected string.

Security Analysis. Each line containing user data is encrypted with AES in CBC mode, which is known to be IND-CPA-secure [10]. As shown in Appendix A, IND-CPA security implies IND-CDBA security. Therefore $\mathsf{Adv}_{\mathsf{r}}$ cannot extract any information from an encrypted database, besides the number of records and their approximate length.

However, PINs' database file does not provide any kind of data integrity. As such, an adversary can exploit the malleability of the CBC mode of operation to modify the content of the database. Since each line is encrypted separately, changes in one record do not affect other records. $\mathsf{Adv}_{\mathsf{rw}}$ can exploit this property to win the MAL-CDBA game with probability 1. After receiving a challenge database DB corresponding to an arbitrary record-set RS, $\mathsf{Adv}_{\mathsf{rw}}$ flips one bit in any of the records to obtain a new database DB' which is considered correct by PINs, i.e., $\mathsf{Valid}(DB') = 1$. $\mathsf{Adv}_{\mathsf{rw}}$ can also remove arbitrary entries, or replace them with versions collected from different challenge databases.

4.8 PasswordSafe v3

Description. The PasswordSafe v3 database is composed of a single file containing all entries [33]. The file can be logically divided into two parts: a header (hdr), an encrypted body (bdy). hdr includes (among other fields) an `IV` and a pair of 256-bit keys, `K` and `L`, which are used to encrypt bdy and to provide authentication using HMAC respectively. `K` and `L` are encrypted using Twofish [38] in ECB mode, under a key derived from a user-provided master password. bdy contains the various database entries, and terminates with an HMAC computed over all fields (before encryption) from hdr to the last entry of bdy, only excluding the database version number.

Security Analysis. PasswordSafe v3 is IND-CDBA-secure *and* MAL-CDBA-secure. (Due to lack of space we omit a formal proof of this statement. The proof is available in the extended version of the paper [15].) As such, neither $\mathsf{Adv}_{\mathsf{r}}$ nor $\mathsf{Adv}_{\mathsf{rw}}$ can win their respective games with non-negligible probability over 1/2.

However, we identified a design flaw that, although irrelevant in our security model, should be considered when adopting this format. The PasswordSafe v3 database format stores both the encryption key and the MAC key used to secure the database content in the file header. In this way, if the master password is changed, the database does not need to be re-encrypted. This technique is usually adopted by encrypted file systems (e.g., [14]) to avoid having to re-encrypt all the data if the master password is changed. However, we believe that this choice may not be appropriate for a password database file. In particular, every time

the database is modified, `IV` is changed and therefore the whole database is re-encrypted. For this reason, the reuse of the same values for `K` and `L` does not imply any savings.

Additionally, this specification detail opens the door to an attack. Assume that an adversary is able to obtain the master password for an encrypted database. Using the master password, the adversary would also be able to retrieve (and store) `K` and `L`. Subsequently, even if the user changes her master password, the adversary can still decrypt and/or modify any new version of the database. The only way to recover from a compromise of the master password is to completely discard the database and create a new one, i.e., changing the master password serves no purpose. It should be noted that some implementations that use the PasswordSafe v3 format are not vulnerable to this attack (e.g., Password Safe [32]), since they choose a new random `K` and `L` every time the database is saved. This makes such implementations less efficient than they could be, but secure.

4.9 Roboform

Format Description. Roboform stores its password database in several files. Each file contains a header, which encodes two URLs: `goto`, which is used as a bookmark by Roboform's browser plugin, and `match`, which is used by Roboform's plugin to determine which username/password record should be used on each web form.

The rest of the record is composed of a short header and an encrypted payload. Roboform allows users to choose between AES, Blowfish, DES, Triple-DES and RC6 for payload encryption.

Security Analysis. Roboform's password format is vulnerable to attacks from both $\mathsf{Adv}_{\mathsf{r}}$ and $\mathsf{Adv}_{\mathsf{rw}}$ in our security model.

Adversary $\mathsf{Adv}_{\mathsf{r}}$ can win the IND-CDBA game with probability 1 by constructing two same-size record-sets RS_0 and RS_1 which differ in at least one of the URLs in their records. Since neither the `goto` nor the `match` fields are encrypted, $\mathsf{Adv}_{\mathsf{r}}$ can always identify which record-set corresponds to challenge DB_b. As a proof of concept, we wrote a small script that decodes the `goto` and `match` URLs. The script is available upon request.

In practice, this allows $\mathsf{Adv}_{\mathsf{r}}$ to gather recover a list of web site visited by the user even if web cache and history have been deleted.

Similarly, $\mathsf{Adv}_{\mathsf{rw}}$ can win the MAL-CDBA game with probability 1, since neither of the URLs stored in Roboform's database are integrity protected. $\mathsf{Adv}_{\mathsf{rw}}$ requests a database corresponding to an arbitrary recordset RS, and after receiving the corresponding database DB, creates DB' by altering one or both URLs. The lack of integrity protection means that $\mathsf{Valid}(DB') = 1$.

In practice, an adversary can use this vulnerability to mount a phishing attack by altering URLs and redirecting users to a malicious website designed to capture login credentials.

	Read-Only Attacker (IND-CDBA)	Read-Write Attacker (MAL-CDBA)
Google Chrome	×	×
Mozilla Firefox	×	×
Microsoft Internet Explorer	×	×
1Password	×	×
KDB (aka KeePass 1.x)	×	×
KDBX4 (aka KeePass 2.x)	✓	×
PINs	✓	×
PasswordSafe v3	✓[1]	✓[1]
Roboform	×	×

Table 2. Vulnerabilities overview. This table shows, for each format, whether it is secure (✓) or broken (×) in the two security games IND-CDBA and MAL-CDBA, defined in Section 3. [1]PasswordSafe v3 is secure in our model but with an interesting design flaw (see Section 4.8).

5 Discussion

Table 2 summarizes the result of our security analysis. Almost all the formats are vulnerable to attack either in the IND-CDBA or MAL-CDBA security model, or both. What does that mean for the use of these formats in practice? The answer depends on the security provided by the storage mechanism that hosts the password database. We divide the database formats into three classes: *Class I*: those that can be used on an insecure storage medium. According to our analysis, the only format in this class is PasswordSafe v3; *Class II*: those that can be used if the underlying storage mechanism provides integrity and data authenticity. This class contains KDBX4 and PINs; and *Class III*: those that can be used securely only if the underlying storage provides integrity, authenticity and secrecy. This class contains the remaining formats.

Class I password managers can be used safely without any special considerations, except for one caveat with PasswordSafe v3 described in Section 4.8. To safely use Class II password managers in practice, users should make sure never to rely on any information in the database that could have been changed by a malicious adversary. For example, if privacy is not a concern and the password database is kept on, e.g., a read-only smart card, KDBX4 and PINs *can* be used to securely store passwords.

There is nothing inherently wrong with storing passwords in a Class III password manager, e.g., an unencrypted text file, as long as the user is made aware that the format provides no secrecy, integrity or authenticity. In fact, if the user is taking additional steps to store an unencrypted password database on a secure medium (e.g., an encrypted file system) this may be a perfectly safe approach. As an example, Google Chrome stores passwords in a database format that is not designed to provide security. To use the Google Chrome password

manager in practice, users should completely prevent access to the database from any unauthorized party (e.g., other users of the same machine).

It seems fair to require that a password manager that asks users to authenticate themselves with a password, at least provides secrecy and data authenticity. This is currently only achieved by a single password database format, namely PasswordSafe v3. As a general rule, a password manager should be explicit about the security offered by the underlying database format.

6 Related Work

Although the concept of a password manager is well known and used by people all over the world, there is very little scientific literature on the subject.

In 2003 Luo and Henry proposed a method for protecting multiple accounts [25]. Their solution requires a user to remember only one password, called a common password, to access any of a number of accounts. The authors propose a Web based implementation with a password calculator written in JavaScript.

In an attempt to solve the same problem, Blasko published an IBM Research Report in 2005 [8] proposing a Wristwatch-Computer Based Password-Vault. Blasko describes the design and implementation of a wearable computer with wireless connectivity, processing, input, and display capabilities, that is meant to store a users passwords for different services.

A year later, Gaw and Felten published a study of Password Management Strategies for Online Accounts [16]. The authors studied how many passwords 49 undergraduates had, and how often they reused these passwords. At that time about 38% of the people participating in the study used password managers. More than two thirds of those used online, web based password managers. With the inclusion of password managers in popular browsers, that number is presumably significantly higher today.

In 2009 Englert and Shah published a paper on the Design and Implementation of a secure Online Password Vault [12]. This works describes an architecture where encryption and decryption is done locally on the user's machine but storage done online.

Bonneau and Preibusch reported results of, what they claim is, the first large-scale empirical analysis of password implementations deployed on the Internet [9]. This study included 150 websites which offer free user accounts for a variety of purposes, including the most popular destinations on the web and a random sample of e-commerce, news, and communication websites. This work does not deal directly with password managers but the findings support the claim that many online services use poor practices when dealing with user credentials. This serves to highlight the need for password managers and consequently, the need for secure password manager database formats.

In [5], Belenko and Sklyarov analyze the security of several password manager applications running on iOS and BlackBerry smartphones. Their analysis focuses on a passive adversary, who is able to access a password database *at*

rest. The goal of the adversary is to determine the database master password, and therefore access the protected data. The authors show that most password managers either force the user to protect the database using a short (four digit) PIN, or do not use expensive key derivation functions to compute the database encryption/decryption key from the master password. This allows an adversary to perform password recovery attacks relatively short time for low-entropy passwords.

7 Conclusion

Password managers are critical pieces of software used to securely store sensitive information. This paper presents the first rigorous analysis of the storage formats used by popular password managers.

We defined two realistic security models, designed to represent the capabilities of real-world attacks. One for passive and one for active attackers. We analyzed popular password manager database formats in our security models; for each vulnerable format, we provided a formal argument for why it is broken. We also showed what the theoretical vulnerability means in terms of practical attacks. Additionally, when a database format was found to be secure, we provided a formal proof.

Unfortunately, most formats turned out to be broken even against very weak adversaries. For this reason, users should carefully consider whether a particular database format is acceptable for storing data in the cloud, on a USB drive or on a machine shared with other users.

Finally, our works shows that it is indeed possible to construct a format that provides security, usability and low computation and storage overhead, using standard cryptographic tools.

References

1. 1Password. Automatic Syncing Using Dropbox. `http://help.agilebits.com/1Password3/cloud_syncing_with_dropbox.html`.
2. AgileBits, Inc. 1password. `https://agilebits.com/onepassword`.
3. AgileBits, Inc. 1password agile keychain design. `http://help.agilebits.com/1Password3/agile_keychain_design.html`.
4. B. Pellin. Keepassdroid. `http://www.keepassdroid.com`.
5. A. Belenko and D. Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Co. Ltd., 2012. `http://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf`.
6. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *J. Cryptology*, 21(4), 2008.
7. D. Bernstein. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*. Springer-Verlag, 2008.
8. G. Blasko, C. Narayanaswami, and M. Raghunath. A Wristwatch-Computer Based Password-Vault. Technical report, IBM Research Division, 2005.

9. J. Bonneau and S. Preibusch. The Password Thicket: Technical and Market Failures in Human Authentication on the Web. *Information Security*, 8(1), 2010.
10. I. Damgaard and J. Nielsen. Expanding Pseudorandom Functions; or: From Known-Plaintext Security to Chosen-Plaintext Security. In *CRYPTO 2002*, 2002.
11. R. Dhamija, J Tygar, and M. Hearst. Why Phishing Works. In *SIGCHI conference on Human Factors in computing systems*, New York, NY, USA, 2006. ACM.
12. B. Englert and P. Shah. On the Design and Implementation of a secure Online Password Vault. In *ICHIT 09*. ACM Press, 2009.
13. F. Pilhofer. Password Gorilla. `http://www.fpx.de/fp/Software/Gorilla/`.
14. N. Ferguson. AES-CBC + Elephant diffuser A Disk Encryption Algorithm for Windows Vista. Technical report, Microsoft Research, 2006.
15. P. Gasti and K. Rasmussen. On The Security of Password Manager Database Formats. Technical report, UCI, 2012. Available from Cryptology ePrint Archive `http://eprint.iacr.org`.
16. S. Gaw and E. Felten. Password Management Strategies for Online Accounts. In *SOUPS '06*, Pittsburgh, PA, USA., 2006. ACM Press.
17. GNU zip. The GZIP homepage. `http://www.gzip.org/`.
18. Google. Get a fast, free web browser. `https://www.google.com/chrome/`.
19. Google. Protect your synced data. `http://support.google.com/chrome/bin/answer.py?hl=en&answer=1181035`.
20. A. Herzberg. Why Johnny can't Surf (Safely)? Attacks and Defenses for Web Users. *Computers & Security*, 28(1-2), 2009.
21. J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.
22. J. Katz and M. Yung. Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation. In *In FSE 00*. Springer-Verlag, 2000.
23. KeePass – A Free and Open-source Password Manager. `http://keepass.info/`.
24. KeePassDroid. Dropbox and KeePassDroid. `http://blog.keepassdroid.com/2010/06/dropbox-and-keepassdroid.html`.
25. H. Luo and P. Henry. A Common Password Method for Protection of Multiple Accounts. In *International Symposium on Personal, Indoor and Mobile Radio Communication*, 2003.
26. M. Frazier. Sync Firefox from the Command Line. `http://www.linuxjournal.com/content/sync-firefox-command-line`.
27. M. Vanhove. Kypass. `http://itunes.apple.com/us/app/kypass/id425680960?mt=8`.
28. Microsoft. Internet Explorer 9. `http://windows.microsoft.com/en-us/internet-explorer/products/ie/home`.
29. Microsoft Dev Center. CryptProtectData function. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa380261(v=vs.85).aspx`.
30. Mozilla. Firefox. `http://www.mozilla.org/`.
31. Mozilla. Firefox Sync for Mobile. `http://www.mozilla.org/en-US/mobile/sync/`.
32. Password Safe – Simple & Secure Password Management. `http://passwordsafe.sourceforge.net/`.
33. Password Safe V3 Database Format. `http://passwordsafe.svn.sourceforge.net/viewvc/passwordsafe/trunk/pwsafe/pwsafe/docs/`.
34. PINs, Secure Passwords Manager. `http://www.mirekw.com/winfreeware/pins.html`.
35. Portable Apps. Keepass password safe portable. `http://portableapps.com/apps/utilities/keepass_portable`.

36. R. Muiznieks. passdrop. http://itunes.apple.com/us/app/passdrop/id431185109?mt=8.
37. RomanLab Co. Ltd. USB password manager: When your password database is right where you need it. http://www.anypassword.com/password-database-in-usb-password-manager.html.
38. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. *Current*, 21(1), 1998.
39. Siber Systems, Inc. RoboForm. http://www.roboform.com/.
40. Siber Systems, Inc. Roboform2go for USB drives. http://www.roboform.com/platforms/usb.
41. Trusteer. Reused Login Credentials. http://www.trusteer.com/sites/default/files/cross-logins-advisory.pdf.

A Relationship Between IND-CPA and IND-CDBA

In this section we shed light on the relationship between our notion of IND-CDBA-security and the standard IND-CPA-security. Let $\Pi = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec})$ be a $\mathsf{IND\text{-}CPA}_{\mathsf{Adv_r},\Pi}(\kappa)$-secure encryption scheme. We recall the standard definition of IND-CPA security [21]:

Game 3 ($\mathsf{IND\text{-}CPA}_{\mathcal{A},\Pi}(\kappa)$) *Indistinguishability of chosen plaintext attack. A challenger* Ch *running* Π *interacts with* $\mathcal{A}$ *as follows:*

- Ch *runs* $mp \leftarrow \mathsf{Setup}(1^\kappa)$.
- $\mathcal{A}$ *is given oracle access to* $\mathsf{Enc}_{mp}(\cdot)$
- *Eventually* $\mathcal{A}$ *outputs two same size messages* RS_0, RS_1
- Ch *selects a bit* b *uniformly at random and the ciphertext* $DB_b \leftarrow \mathsf{Enc}(mp, RS_b)$ *is returned to* $\mathcal{A}$.
- $\mathcal{A}$ *eventually outputs bit* b'*; the game outputs 1 iff* $b = b'$.

Definition 4 (IND-CPA **security).** *An encryption scheme* $\Pi = (\mathsf{Setup}_\Pi, \mathsf{Enc}_\Pi, \mathsf{Dec}_\Pi)$ *has indistinguishable encryptions under chosen plaintext attack if there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any efficient adversary* $\mathcal{A}$, $\Pr[\mathsf{IND\text{-}CPA}_{\mathcal{A},\Pi}(\kappa) = 1] \leq 1/2 + \mathsf{negl}(\kappa)$.

It is easy to see that $\mathsf{IND\text{-}CPA}_{\mathcal{A},\Pi}(\kappa)$ security implies $\mathsf{IND\text{-}CDBA}_{\mathsf{Adv_r},\mathcal{PM}}(\kappa)$ security. Let $\mathcal{PM} = (\mathsf{Setup}, \mathsf{Create}, \mathsf{Open}, \mathsf{Valid})$ where $\mathsf{Setup} = \mathsf{Setup}_\Pi$, $\mathsf{Create} = \mathsf{Enc}_\Pi$, $\mathsf{Open} = \mathsf{Dec}_\Pi$ and Valid is defined as in Section 3.

Assume $\mathsf{Adv_r}$ is an adversary that has a non-negligible advantage in the IND-CDBA game. We show how to build a simulator SIM that uses $\mathsf{Adv_r}$ to win the IND-CPA game. SIM lets $\mathsf{Adv_r}$ choose RS_0 and RS_1, and forwards these to Ch. Ch returns DB_b which is forwarded to $\mathsf{Adv_r}$. Eventually $\mathsf{Adv_r}$ outputs its choice for b', and SIM uses it to answer the challenger. Since $(\mathsf{Setup}, \mathsf{Create}, \mathsf{Open})$ is defined as $(\mathsf{Setup}_\Pi, \mathsf{Enc}_\Pi, \mathsf{Dec}_\Pi)$, Sim's advantage is identical to $\mathsf{Adv_r}$'s.