Password Managers: Under the Hood of Secrets Management

February 19, 2019

Also see associated blog

FAQ

Link to Washington Post Exclusive

Abstract:

Password managers allow the storage and retrieval of sensitive information from an encrypted database. Users rely on them to provide better security guarantees against trivial exfiltration than alternative ways of storing passwords, such as an unsecured flat text file. In this paper we propose security guarantees password managers should offer and examine the underlying workings of five popular password managers targeting the Windows 10 platform: 1Password 7 [1], 1Password 4 [1], Dashlane [2], KeePass [3], and LastPass [4]. We anticipated that password managers would employ basic security best practices, such as scrubbing secrets from memory when they are not in use and sanitization of memory once a password manager was logged out and placed into a locked state. However, we found that in all password managers we examined, trivial secrets extraction was possible from a locked password manager, including the master password in some cases, exposing up to 60 million users that use the password managers in this study to secrets retrieval from an assumed secure locked state.

Introduction:

First and foremost, password managers are a good thing. All password managers we have examined add value to the security posture of secrets management, and as Troy Hunt, an active security researcher once wrote, “Password managers don’t have to be perfect, they just have to be better than not having one” [5]. Aside from being an administrative tool to allow users to categorize and better manage their credentials, password managers guide users to avoid bad password practices such as using weak passwords, common passwords, generic passwords, and password reuse.

The tradeoff is that users’ credentials are then centrally stored and managed, typically protected by a single master password to unlock a password manager data store. With the rising popularity of password manager use it is safe to assume that adversarial activity will target the growing user base of these password managers. Table 1, below, outlines the number of individual users and business entities for each of the password managers we examine in this paper.

Password Manager	Users	Business Entities
1Password	15,000,000 [6]	30,000 [6]
Dashlane	10,000,000 [7]	10,000 [7]
KeePass	20,000,000 [8]	Unknown
LastPass	16,500,000 [9]	43,000 [9]

Table 1. Number of private users and business entities of 1Password (all versions), Dashlane, KeePass and LastPass.

Motivation:

With the proliferation of online services, password use has gone from about 25 passwords per user in 2007 [10] to 130 in 2015 and is projected to grow to 207 in 2020 [11]. This, combined with a userbase of 60 million across password managers we examine in this paper, creates a target rich environment in which adversaries can carefully craft methods to extract an increasingly growing and valuable trove of secrets and credentials.

An example in which a password manager appears to have been specifically targeted is an attack that led to the loss of 2578 units of Ethereum (ETH), a cryptocurrency valued at the time of 1.5 million USD. The attack was carried out against a cryptocurrency trading assistant platform, Taylor [12]. Taylor issued a statement that indicated a device which was using 1Password for secrets management was compromised [13]. It remains unclear, whether the attacker found a security issue in 1Password itself or simply discovered the master password in some other way, or whether the compromise had nothing to do with password managers.

Given the combination of an increasing number of credentials held in password managers, the value of those secrets and the emerging threats specifically targeting password managers it is important for us to examine the increased risk a user or organization faces in terms of secrets exposure when using a password manager. Our approach for this was to survey popular password managers to determine common defenses they employ against secrets exfiltration. We incorporate the best security features of each into a hypothetical, best possible password manager, that provides a minimum set of guarantees outlined in the next section. Then we compare the password managers studied against those security guarantees.

Password Manager Security Guarantees:

All password managers studied work in the same basic way. Users enter or generate passwords in the software and add any pertinent metadata (e.g., answers to security questions, and the site the password goes to). This information is encrypted and then decrypted only when it is needed for display, for passing to a browser add-on that fills the password into a website, or for copying to the clipboard for use.

Throughout this paper we will refer to password managers in three states of existence: not running, unlocked (and running), and locked (and running; this state assumes the password manager was previously unlocked). We assume that the user does not have additional layers of encryption such as full disk encryption or per process virtualization. We define the three states below:

Not Running

We define “not running” as a state where the password manager has previously been installed, configured, and interacted with by the user to store secrets, but has not been launched since the last reboot or has been terminated by the user since it was last used.

In this “not running” state the password manager should guarantee:

> There should be no data stored on disk that would offer an attacker leverage toward compromising the database stored on disk (e.g. the master password or encryption key stored in a configuration file).
> Even if an attacker retrieves the password database from disk, it should be encrypted in such a

> way that an attacker cannot decrypt it without knowing the master password.
> The encryption should be designed in such a way that, so long as the user did not use a trivial password, the attacker cannot brute force guess the master password in a reasonable amount of time using commonly available computing resources.

Running: Unlocked State

We define running in an "unlocked state" as cases where the password manager is running, and where the user has typed in the master password in order to decrypt and access the stored passwords inside the manager. The user may have displayed, copied to clipboard, or otherwise accessed some of the passwords in the password manager.

In this "running, unlocked state" the password manager should guarantee:

> It should not be possible to extract the master password from memory, either directly or in any form that allows the original master password to be recovered.
> For those stored passwords that have *not* been displayed/copied/accessed by the user since the password manager was unlocked, it should not be possible to extract those unencrypted passwords from memory.

Knowing usability constraints that affect password managers, we concede that:

> It may be possible to extract those passwords from memory that were displayed/copied/accessed in the current unlocked session.
> It may be possible to extract cryptographic information derived from the master password sufficient to decrypt other stored passwords, but not the master password itself.

Running: Locked State

We define "in locked state" as cases where (1) the password manager was just launched but the user has not entered the master password yet, or (2) the user previously entered the master password and used the password manager, but subsequently clicked the 'Lock' or 'Log Out' button.

In this "running, locked state" the password manager should guarantee:

> All the security guarantees of a not-running password manager should apply to a password manager that is in the locked state.

Since a locked password manager still exists as a process in virtual memory, this requires additional guarantees:

> It should not be possible to extract the master password from memory, either directly or in any form that allows the original master password to be recovered.
> It should not be possible to extract from memory any cryptographic information derived from the master password that might allow passwords to be decrypted without knowing the master password.
> It should not be possible to extract any unencrypted passwords from memory that are stored in the password manager.

In addition to these explicit security guarantees, we expect password managers to incorporate additional hardening measures where possible, and to have these hardening measures enabled by default. For example, password managers should attempt to block software keystroke loggers from accessing the master password as it is typed, attempt to limit the exposure of unencrypted passwords left on the clipboard, and take reasonable steps to detect and block modification or patching of the password manager and its supporting libraries that might expose passwords.

Scope:

In this paper we will examine the inner workings as they relate to secrets retrieval and storage of 1Password, Dashlane, KeePass and LastPass on the Windows 10 platform (Version 1803 Build 17134.345) using an Intel i7-7700HQ processor. We examine susceptibility of a password manager to secrets exfiltration via examination of the password database on disk; memory forensics; and finally, keylogging, clipboard monitoring, and binary modification. Each password manager is examined in its default configuration after install with no advanced configuration steps performed.

The focus on our evaluation of password managers is limited to the Windows platform. Our findings can be extrapolated to password manager implementations in other operating systems to guide research to areas of interest that are discussed in this paper.

Target Password Managers:

The following password managers with their corresponding versions were evaluated:

Product	Version
1Password4 for Windows	4.6.2.626
1Password7 for Windows	7.2.576
Dashlane for Windows	6.1843.0
KeePass Password Safe	2.40
LastPass for Applications	4.1.59

Security of Password Managers in the Non-Running State

We first consider the security of password managers when they are not running. We focus on the attack vector of compromising passwords from disk. Unless password managers have severe vulnerabilities such as logging passwords to unencrypted log files or other egregious issues, the password managers' defenses against the disk attack surface rest on the cryptography used to protect the password database. Here, we examine which algorithm each password manager uses to transform the master password into an encryption key, and whether the algorithm and number of iterations is severely lacking in its ability to resist contemporary cracking attacks.

Table 2, below, outlines the key expansion algorithm type used and number of iterations in each password manager’s default configuration. With regard to key expansion recommendations set by NIST [14]we found that each key expansion algorithm used in the password managers was acceptable and that the number of iterations adequate. We concluded that the password managers were secure against compromising passwords from disk as the software is not running, and that brute forcing the encrypted password entries on disk would be computationally prohibitive, although not impossible if given enough computing resources. Given this, we moved on to the attack surface of passwords stored in memory while the password managers are running.

Password Manager	Key Expansion Algorithm	Iterations
1Password4	PBKDF2-SHA256	40,000 [15]
1Password7	PBKDF2-SHA256	100,000 [16]
Dashlane	Argon2	3 [17]
KeePass	AES-KDF	60,000 [18]
LastPass	PBKDF2-SHA256	100,100 [19]

Table 2. Each password managers default key expansion algorithm and number of iterations.

Security of Password Managers in Running States

We expected and found that all password managers reviewed sufficiently protect the master password and individual passwords while they are *not*running. The remaining bulk of our assessment of password managers in the running state was focused on the effectiveness of the locked state and whether the unlocked state left the minimum possible amount of sensitive information in memory. The following sections outline violations of our proposed security guarantees of password managers in a running locked and unlocked state.

1Password4 (Version: 4.6.2.626)

We assessed the security of 1Password4 while running and found reasonable protections against exposure of individual passwords in the unlocked state; unfortunately, this was overshadowed by its handling of the master password and several broken implementation details when transitioning from the unlocked to the locked state. On the positive side, we found that as a user accesses different entries in 1Password4, the software is careful to clear the previous unencrypted password from memory before loading another. This means that only one unencrypted password can be in memory at once. On the negative side, the master password remains in memory when unlocked (albeit in obfuscated form) and the software fails to scrub the obfuscated password memory region sufficiently when transitioning from the unlocked to the locked state. We also found a bug where, under certain user actions, the master password can be left in memory in cleartext even while locked.

Failure to Scrub Obfuscated Master Password from Memory

It is possible to recover and deobfuscate the master password from 1Password4 since it is not scrubbed from memory after placing the password manager in a locked state. Given a scenario where a user has unlocked 1Password4 and then placed it back into a locked state, 1Password4 will prompt for the master password again as shown in Figure 1below. However, 1Password4 retains the master password in memory, although in an encoded/obfuscated format as shown in Figure 2.

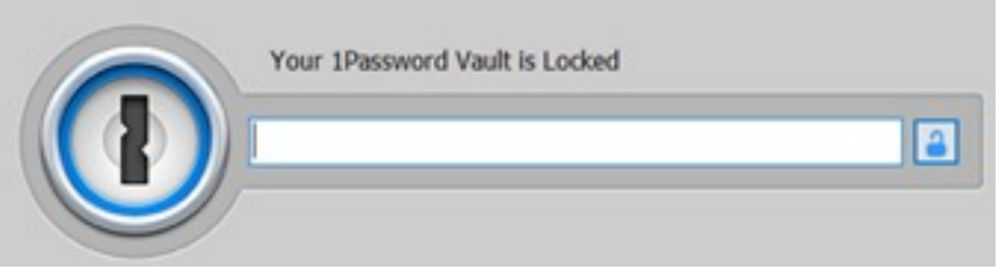
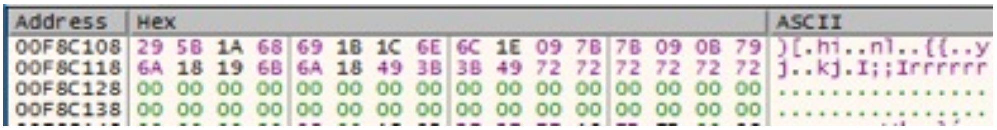


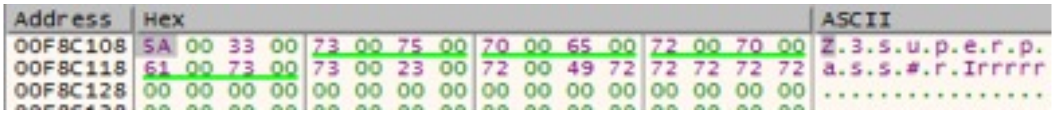
Figure 1. 1Password4 in a locked state awaiting master password input.



```
Address  |Hex                                              |ASCII
00F8C108 |29 5B 1A 68|69 1B 1C 6E|6C 1E 09 7B|7B 09 0B 79|)[.hi..nl..{{..y
00F8C118 |6A 18 19 6B|6A 18 49 3B|3B 49 72 72|72 72 72 72|j..kj.I;;Irrrrrr
00F8C128 |00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00|................
00F8C138 |00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00|................
```

Figure 2. Encoded master password present in memory while 1Password4 is in a locked state.

We can use this information to intercept normal workflows in which 1Password4 calls RtlRunEncodeUnicodeString and RtlRunDecodeUnicodeString to obfuscate the master password to instead reveal the already present, but encoded master password into cleartext (Figure 3).

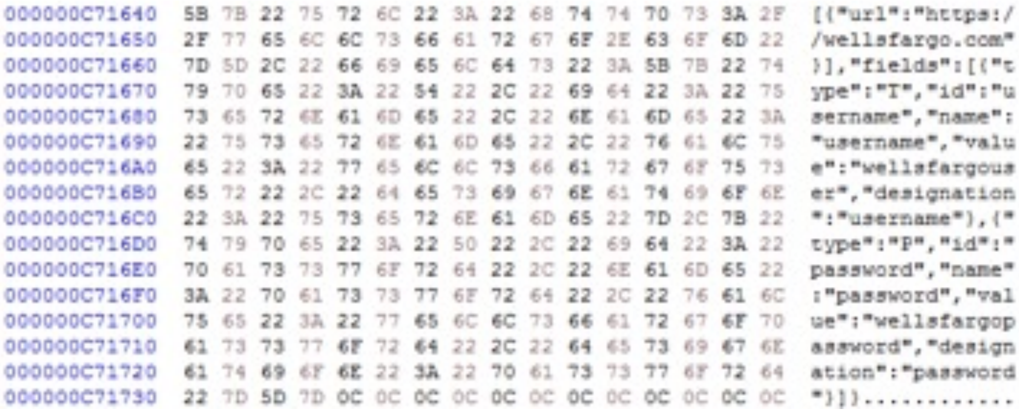


```
Address  |Hex                                              |ASCII
00F8C108 |5A 00 33 00|73 00 75 00|70 00 65 00|72 00 70 00|Z.3.s.u.p.e.r.p.
00F8C118 |61 00 73 00|73 00 23 00|72 00 49 72|72 72 72 72|a.s.s.#.r.Irrrrr
00F8C128 |00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00|................
```

Figure 3. Master password revealed after the expected RtlRunEncodeUnicodeString and RtlRunDecodeUnicodeString was reversed, thereby forcing 1Password4 to decode the encoded master password that was not scrubbed from memory.

Copying the Current Password Entry from Memory

Only entries that are actively being interacted with exist in memory as plaintext. Figure 4is an example of an entry in memory as its being interacted with. Once 1Password4 is locked, the memory region is deallocated . Note that the deallocated region is not first scrubbed, however the Windows memory manager will zero out any freed pages of memory before making them available for re-allocation by the Windows memory manager.



```
000000C71640  5B 7B 22 75 72 6C 22 3A 22 68 74 74 70 73 3A 2F  [{"url":"https:/
000000C71650  2F 77 65 6C 6C 73 66 61 72 67 6F 2E 63 6F 6D 22  /wellsfargo.com"
000000C71660  7D 5D 2C 22 66 69 65 6C 64 73 22 3A 5B 7B 22 74  }],"fields":[{"t
000000C71670  79 70 65 22 3A 22 54 22 2C 22 69 64 22 3A 22 75  ype":"T","id":"u
000000C71680  73 65 72 6E 61 6D 65 22 2C 22 6E 61 6D 65 22 3A  sername","name":
000000C71690  22 75 73 65 72 6E 61 6D 65 22 2C 22 76 61 6C 75  "username","valu
000000C716A0  65 22 3A 22 77 65 6C 6C 73 66 61 72 67 6F 75 73  e":"wellsfargous
000000C716B0  65 72 22 2C 22 64 65 73 69 67 6E 61 74 69 6F 6E  er","designation
000000C716C0  22 3A 22 75 73 65 72 6E 61 6D 65 22 7D 2C 7B 22  ":"username"},{"
000000C716D0  74 79 70 65 22 3A 22 50 22 2C 22 69 64 22 3A 22  type":"P","id":"
000000C716E0  70 61 73 73 77 6F 72 64 22 2C 22 6E 61 6D 65 22  password","name"
000000C716F0  3A 22 70 61 73 73 77 6F 72 64 22 2C 22 76 61 6C  :"password","val
000000C71700  75 65 22 3A 22 77 65 6C 6C 73 66 61 72 67 6F 70  ue":"wellsfargop
000000C71710  61 73 73 77 6F 72 64 22 2C 22 64 65 73 69 67 6E  assword","design
000000C71720  61 74 69 6F 6E 22 3A 22 70 61 73 73 77 6F 72 64  ation":"password
000000C71730  22 7D 5D 7D 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C  "}]}............
```

Figure 4. Password entry in memory during active interaction.

1Password7 (Version: 7.2.576)

After assessing the legacy 1Password4, we moved on to 1Password7, the current release. Surprisingly, we found that it is less secure in the running state compared to 1Password4. 1Password7 decrypted all individual passwords in our test database as soon as it is unlocked and caches them in memory, unlike 1Password4 which kept only one entry at a time in memory. Compounding this, we found that 1Password7 scrubs neither the individual passwords, the master password, nor the secret key (an extra field introduced in 1Password6 that combines with the master password to derive the encryption key) from memory when transitioning from unlocked to locked. This renders the “lock” button ineffective; from the security standpoint, after unlocking and using 1Password7, the user must exit the software entirely in order to clear sensitive information from memory as locking should.

It appears 1Password may have rewritten their software to produce 1Password7 without implementing secure memory management and secrets scrubbing workflows present in 1Password4 and abandoning the distinction between a ‘running unlocked’ and ‘running locked’ state in terms of secrets exposure. Interestingly, this is not the case. Prior marketing material for 1Password claimed [20]to feature Intel SGX technology. This technology protects secrets inside secure memory enclaves so that other processes and even higher privileged components (such as the kernel) cannot access them. Were SGX to be implemented correctly, 1Password7 would have been the most secure password manager in our research by far. Unfortunately, SGX was only supported as a beta feature in 1Password6 and early versions of 1Password7, and was dropped for later versions. This was only evident from gathering the details about it on a 1Password support forum [21].

Exposure of Cleartext Master Password, Secret Key and Entries in Memory

As stated before, all secrets are exposed by 1Password7 when in an unlocked and locked state. To demonstrate the severity of this issue we created proof of concept code to read 1Password7's memory address space to extract these items. The proof of concept applications ran in the existing user context (which was an ordinary non-administrative user).

Show below is 1Password7 in a locked state, Figure 5(having previously been unlocked but then again locked) awaiting password entry to unlock it.

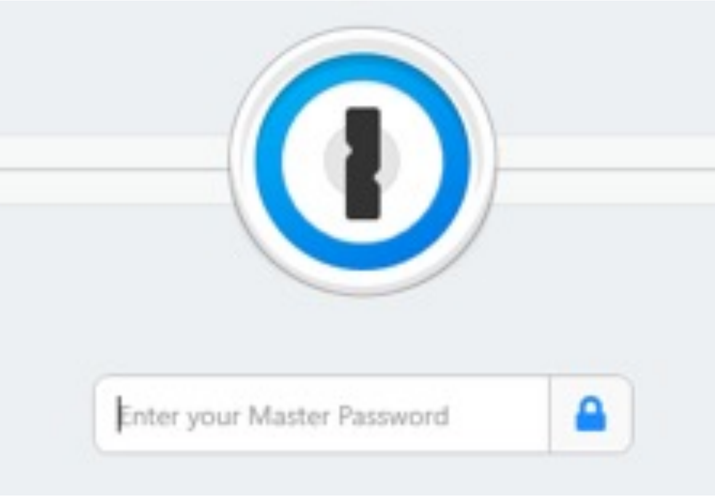
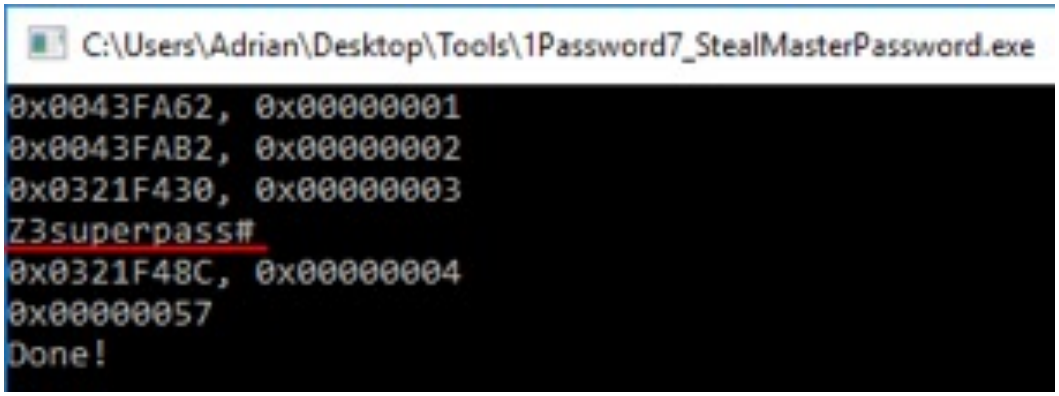


Figure 5. 1Password7 in a locked state, having previously been open and then locked.

Figure 6 illustrates the automated retrieval of the master password.

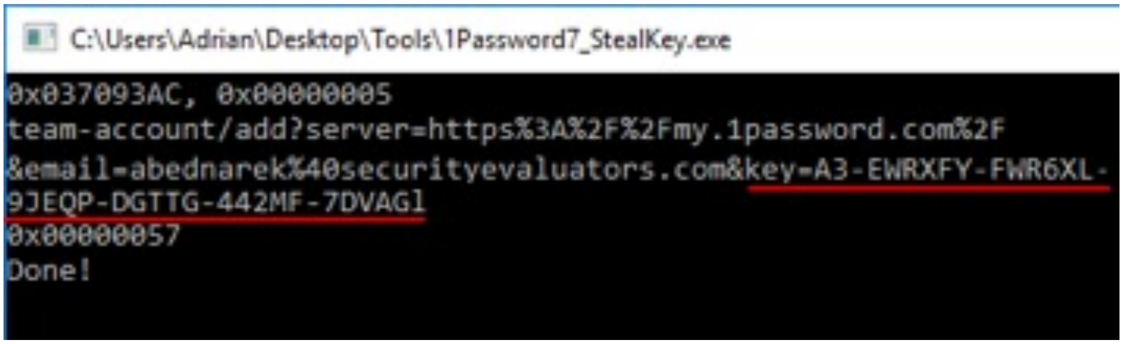


C:\Users\Adrian\Desktop\Tools\1Password7_StealMasterPassword.exe

```
0x0043FA62, 0x00000001
0x0043FAB2, 0x00000002
0x0321F430, 0x00000003
Z3superpass#
0x0321F48C, 0x00000004
0x00000057
Done!
```

Figure 6. Extracting the master password from a locked 1Password7 instance

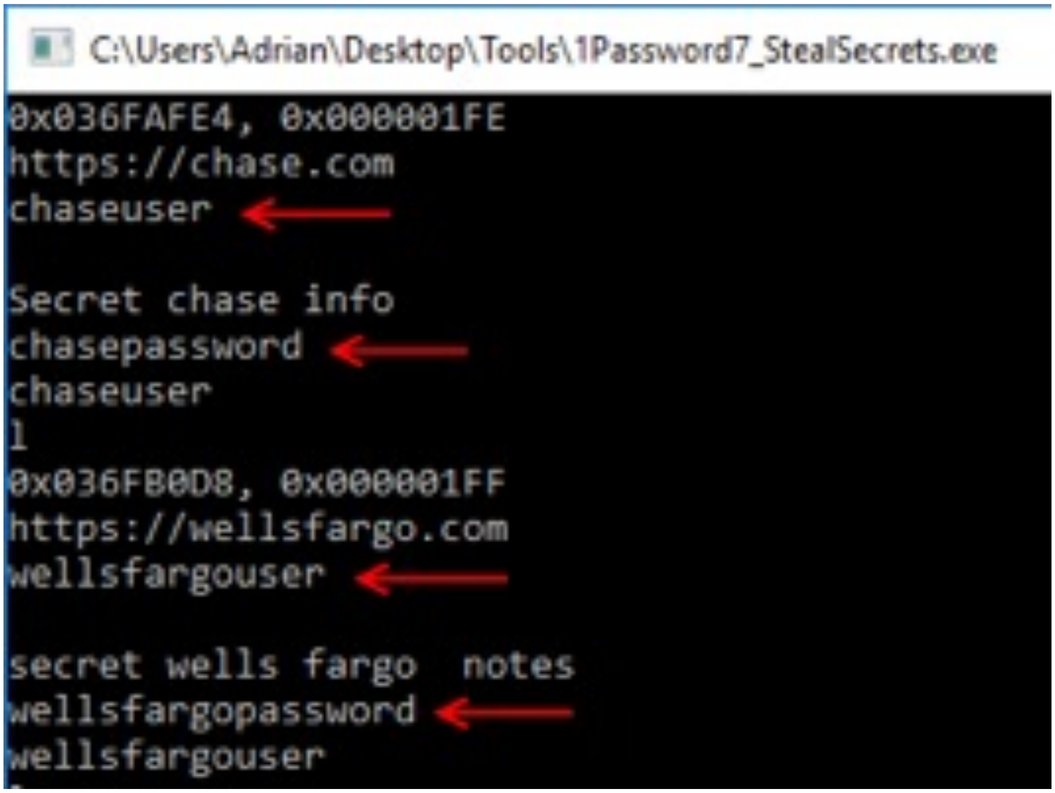
Figure 7 shows the extraction of the secret key that is needed along with the master password to unlock an encrypted database, and Figure 8shows the automated extraction of secret entries.



C:\Users\Adrian\Desktop\Tools\1Password7_StealKey.exe

```
0x037093AC, 0x00000005
team-account/add?server=https%3A%2F%2Fmy.1password.com%2F
&email=abednarek%40securityevaluators.com&key=A3-EWRXFY-FWR6XL-
9JEQP-DGTTG-442MF-7DVAGl
0x00000057
Done!
```

Figure 7. Extracting the secret key from 1Password7 in a locked state.



C:\Users\Adrian\Desktop\Tools\1Password7_StealSecrets.exe

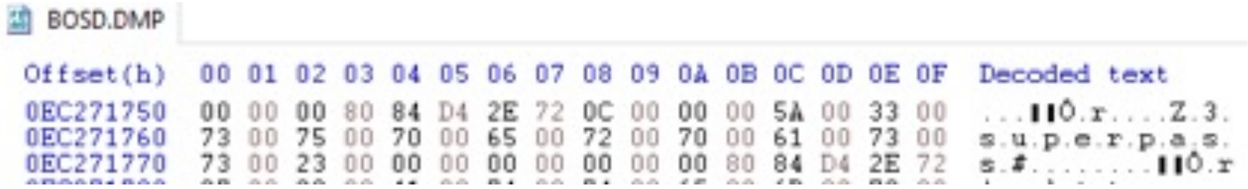
```
0x036FAFE4, 0x000001FE
https://chase.com
chaseuser

Secret chase info
chasepassword
chaseuser
1
0x036FB0D8, 0x000001FF
https://wellsfargo.com
wellsfargouser

secret wells fargo  notes
wellsfargopassword
wellsfargouser
```

Figure 8. Extracting password entries from a locked instance of 1Password7.

The memory “hygiene” of 1Password7 is so lacking, that it is possible for it to leak passwords from memory without an intentional attack at all. During our evaluation of 1Password7, we encountered a system stop error (kernel mode exception) on our Windows 10 workstation, from an unrelated hardware issue, that created a full memory debug dump to disk. While examining this memory dump file, we came across our secrets that 1Password7 held cleartext, in memory, in a locked state when the stop error occurred (Figure 9).



```
BOSD.DMP

Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
0EC271750  00 00 00 80 84 D4 2E 72 0C 00 00 00 5A 00 33 00  ...▮▮Ô.r....Z.3.
0EC271760  73 00 75 00 70 00 65 00 72 00 70 00 61 00 73 00  s.u.p.e.r.p.a.s.
0EC271770  73 00 23 00 00 00 00 00 00 00 00 80 84 D4 2E 72  s.#........▮▮Ô.r
```

Figure 9. Windows 10 crash dump file contained secrets 1Password7 held in memory in a locked state.

For all password managers that leave secrets in memory, this creates a threat model where secrets may be extracted in a non-running state as a by-product of system activity and/or crash/debug log files. Moreover, some companies have a policy to image workstations that have had malware encounters as part of the incident response procedure. A user that happened to be running 1Password7 while this procedure was initiated should assume that all secrets have been compromis

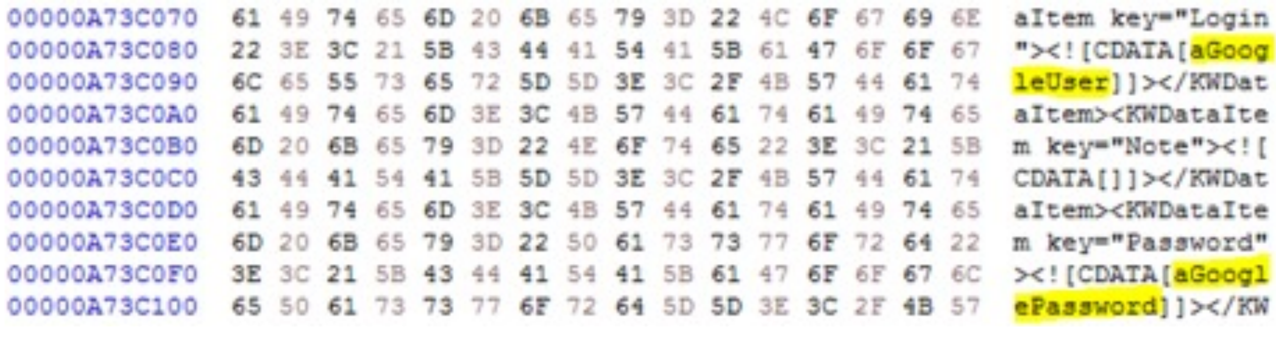
Dashlane (Version: 6.1843.0)

In our Dashlane evaluation, we noted workflows that indicate focus was placed on concealing secrets in memory to reduce their likelihood of extraction. Also, unique to Dashlane, was the usage of memory/string and GUI management frameworks that prevented secrets from being passed around to various OS API's that could expose them to eavesdropping by trivial malware.

Similar to 1Password4, Dashlane exposes only the active entry a user is interacting with. So, at most, the last active entry is exposed in memory while Dashlane is in an unlocked and locked state. However, once a user updates any information in an entry, Dashlane exposes the entire database plaintext in memory and it remains there even after Dashlane is logged out of or 'locked'.

Exposure of Cleartext Entries in Memory

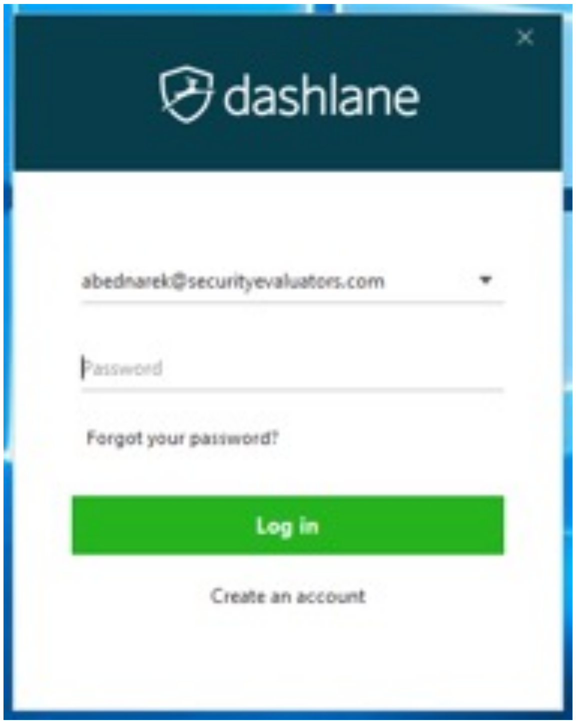
Password entries in Dashlane are stored in an XML object. Upon interacting with any entry this XML object becomes exposed in cleartext and can be easily extracted in both locked and unlocked states. Figure 10, below, is an example of a portion of this XML data structure.



```
00000A73C070  61 49 74 65 6D 20 6B 65 79 3D 22 4C 6F 67 69 6E  aItem key="Login
00000A73C080  22 3E 3C 21 5B 43 44 41 54 41 5B 61 47 6F 6F 67  "><![CDATA[aGoog
00000A73C090  6C 65 55 73 65 72 5D 5D 3E 3C 2F 4B 57 44 61 74  leUser]]></KWDat
00000A73C0A0  61 49 74 65 6D 3E 3C 4B 57 44 61 74 61 49 74 65  aItem><KWDataIte
00000A73C0B0  6D 20 6B 65 79 3D 22 4E 6F 74 65 22 3E 3C 21 5B  m key="Note"><![
00000A73C0C0  43 44 41 54 41 5B 5D 5D 3E 3C 2F 4B 57 44 61 74  CDATA[]]></KWDat
00000A73C0D0  61 49 74 65 6D 3E 3C 4B 57 44 61 74 61 49 74 65  aItem><KWDataIte
00000A73C0E0  6D 20 6B 65 79 3D 22 50 61 73 73 77 6F 72 64 22  m key="Password"
00000A73C0F0  3E 3C 21 5B 43 44 41 54 41 5B 61 47 6F 6F 67 6C  ><![CDATA[aGoogl
00000A73C100  65 50 61 73 73 77 6F 72 64 5D 5D 3E 3C 2F 4B 57  ePassword]]></KW
```

Figure 10. Excerpt of a fully decrypted Dashlane XML password database in an unlocked and locked state.

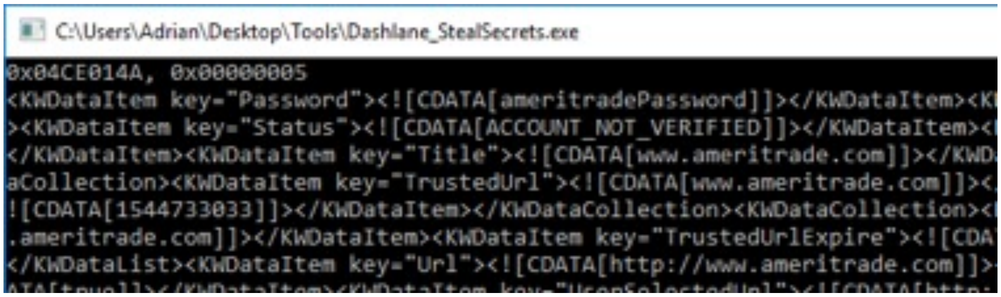
Knowing that this data structure exists in a locked state, we then created a proof of concept application to extract it from a locked instance of Dashlane. Figure 11, below, is a locked instance of Dashlane prompting for the master password to unlock it.



abednarek@securityevaluators.com

Figure 11. Locked instance of Dashlane.

In this locked state, we then run our proof of concept to extract all stored secrets (Figure 12).



```
C:\Users\Adrian\Desktop\Tools\Dashlane_StealSecrets.exe
0x04CE014A, 0x00000005
<KWDataItem key="Password"><![CDATA[ameritradePassword]]></KWDataItem><K
><KWDataItem key="Status"><![CDATA[ACCOUNT_NOT_VERIFIED]]></KWDataItem><
</KWDataItem><KWDataItem key="Title"><![CDATA[www.ameritrade.com]]></KWD
aCollection><KWDataItem key="TrustedUrl"><![CDATA[www.ameritrade.com]]><
![CDATA[1544733033]]></KWDataItem></KWDataCollection><KWDataCollection><
.ameritrade.com]]></KWDataItem><KWDataItem key="TrustedUrlExpire"><![CDA
</KWDataList><KWDataItem key="Url"><![CDATA[http://www.ameritrade.com]]>
```

Figure 12. Extracting secrets from a locked instance of Dashlane.

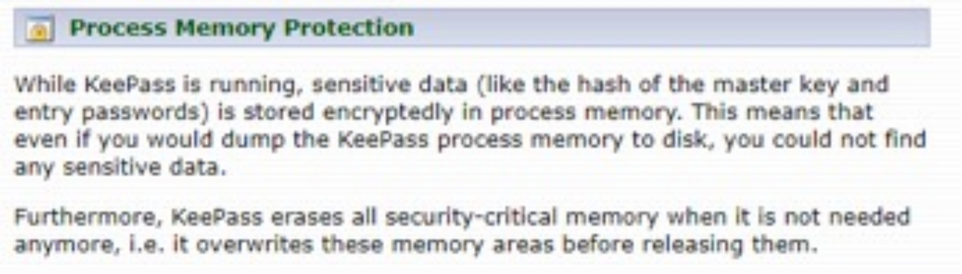
However, even though we are able to extract secrets from a locked state of Dashlane, the memory region they reside in has been dereferenced and freed. So, over time portions of the XML data structure may be overwritten. Throughout our examination, we noticed that secrets may reside for a few minutes. In some instances, we have observed them still resident in memory more than 24 hours.

Dashlane is also unique compared to the other password managers in our examination in that it does not allow you to exit the process via GUI components, such as clicking the close program [x] in the upper right or pressing the ALT-F4 key combination. Doing so causes Dashlane to minimize into the task tray, leaving it susceptible to secrets extraction for extended periods of time.

KeePass (Version: 2.40)

Unlike the other password managers, KeePass is an open source project. Similar to 1Password4, KeePass decrypts entries as they are interacted with, however, they all remain in memory since they are not individually scrubbed after each interaction. The master password is scrubbed from memory and not recoverable. However, while KeePass attempts to keep secrets secure by scrubbing them from memory, there are obviously errors in these workflows as we have discovered that while even in a locked state, we were able to extract entries that had been interacted with.

KeePass claims to use several defenses in depth memory protection mechanisms as stated in an excerpt from their site below (Figure 13). However, they acknowledge that these workflows may involve Windows OS API's that may make copies of various memory buffers which may not be exposed to KeePass for scrubbing.



Process Memory Protection

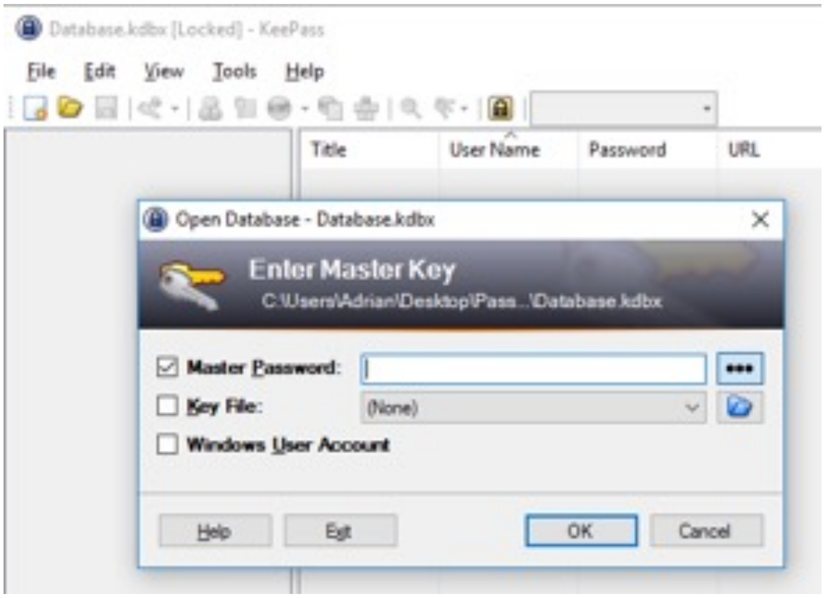
While KeePass is running, sensitive data (like the hash of the master key and entry passwords) is stored encryptedly in process memory. This means that even if you would dump the KeePass process memory to disk, you could not find any sensitive data.

Furthermore, KeePass erases all security-critical memory when it is not needed anymore, i.e. it overwrites these memory areas before releasing them.

Figure 13. KeePass statement on memory protection.

Exposure of Cleartext Entries in Memory

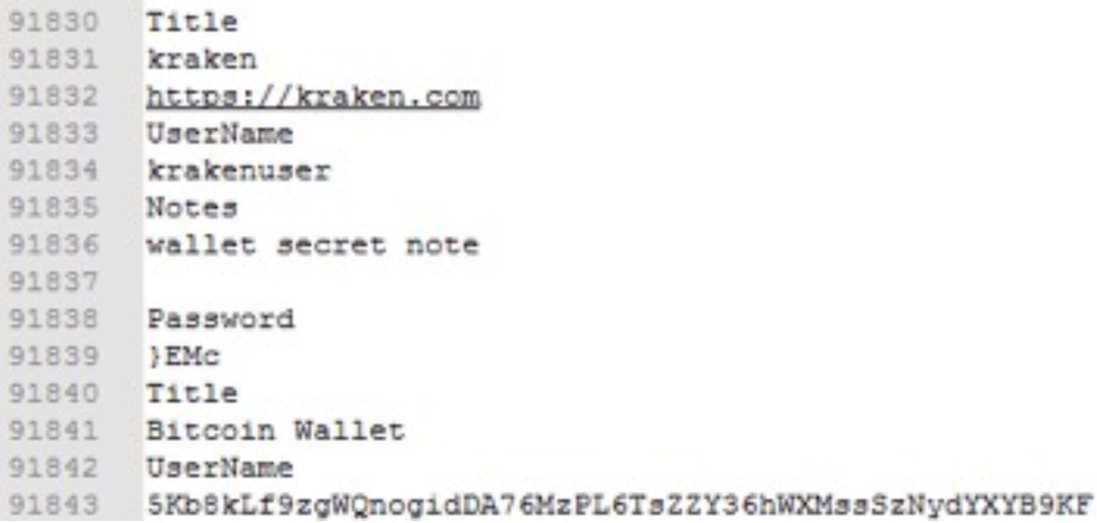
Entries that have been interacted with remain exposed in memory even after KeePass has been placed into a locked state. Figure 14, below, is an example of a locked instance of KeePass prompting for the master password before it can be unlocked.


Database.kdbx [Locked] - KeePass
File Edit View Tools Help
Title User Name Password URL
Open Database - Database.kdbx
Enter Master Key
C:\Users\Adrian\Desktop\Pass...\Database.kdbx
Help Exit OK Cancel

☑ Master Password:
□ Key File: (None)
□ Windows User Account

Figure 14. Locked instance of KeePass.

Secrets are scattered in memory with no references. However, performing a simple strings dump from the process memory of KeePass reveals a list of entries that have been interacted with (Figure 15).

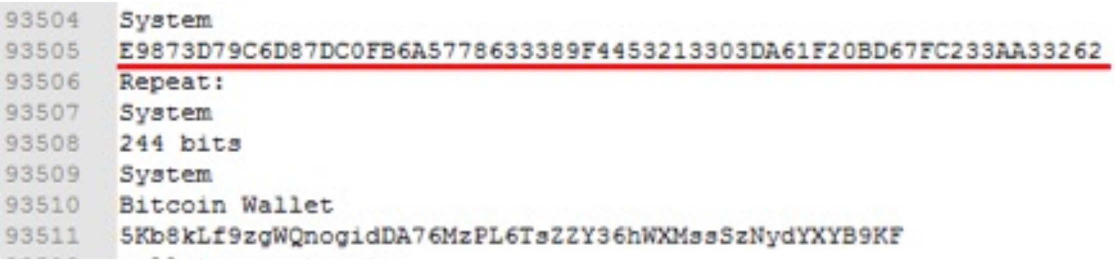


```
Title
kraken
https://kraken.com
UserName
krakenuser
Notes
wallet secret note

Password
}EMc
Title
Bitcoin Wallet
UserName
5Kb8kLf9zgWQnogidDA76MzPL6TsZZY36hWXMssSzNydYXYB9KF
```

Figure 15. List of entries from a locked instance of KeePass.

Using the above information, we can then search for a username to an entry and locate its corresponding password field entry, in the below image (Figure16) we locate the bitcoin private key which was stored in the password field.

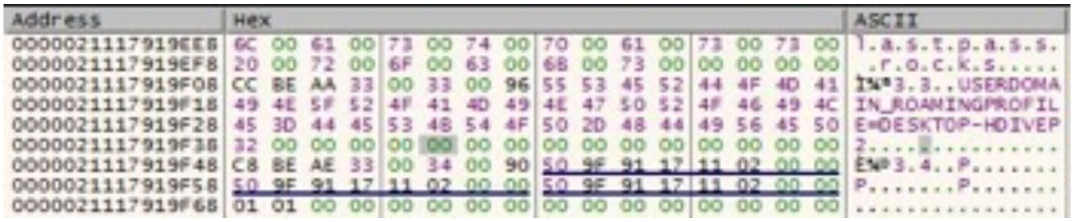


```
System
E9873D79C6D87DC0FB6A5778633389F4453213303DA61F20BD67FC233AA33262
Repeat:
System
244 bits
System
Bitcoin Wallet
5Kb8kLf9zgWQnogidDA76MzPL6TsZZY36hWXMssSzNydYXYB9KF
```

The above methodology can be used to extract any entries that have been interacted with before placing KeePass into a locked state.

LastPass (Version: 4.1.59)

Similar to 1Password4, LastPass obfuscates the master password as its being typed into the unlock field. Once the decryption key has been derived from the master password, the master password is overwritten with the phrase “lastpass rocks” (Figure17).



```
Address          Hex                                                ASCII
000002111791 9EE8 6C 00 61 00 73 00 74 00 70 00 61 00 73 00 73 00  l.a.s.t.p.a.s.s.
```

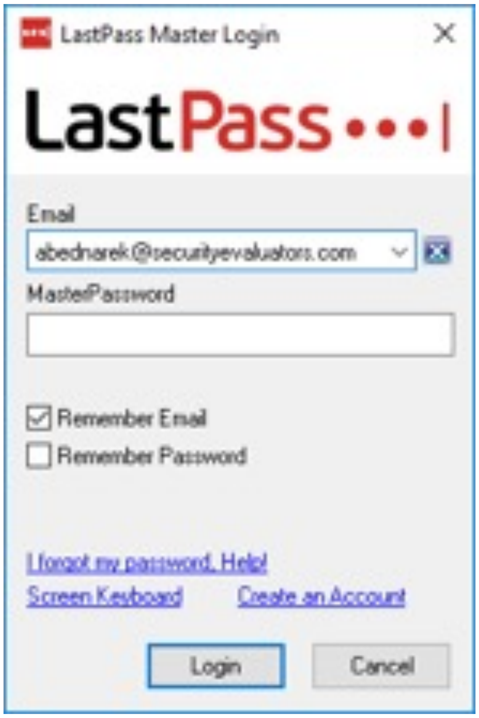
Figure 17. Master password overwritten once the master password has been used in a PBKDF2 key expansion routine.

Once LastPass enters an unlocked state, database entries are decrypted into memory only upon user interaction. However, these entries persist in memory even after LastPass has been placed back into a locked state.

Exposure of Cleartext Master Password and Entries in Memory

During a workflow to derive the decryption key, the master password is leaked into a string buffer in memory and never scrubbed, even when LastPass is placed into a locked state.

The below image, Figure 18, is an instance of LastPass in a locked state awaiting user entry of the master password.



Email: abednarek@securityevaluators.com

MasterPassword:

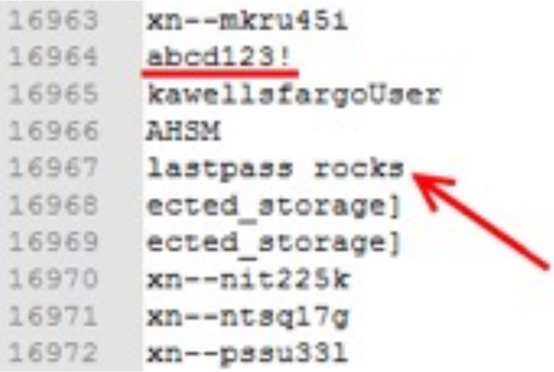
☑ Remember Email

□ Remember Password

Figure 18. Locked instance of LastPass.

In this locked state, we can recover the master password and any interacted with password entries with the same methodology used in KeePass, in which a simple strings dump was performed on the active process.

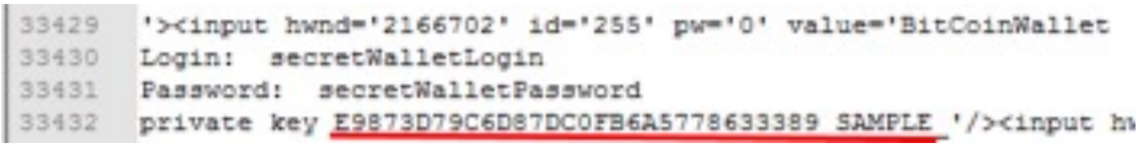
The image below, Figure19, is an example of recovering the master password, in a locked state, which ironically is always found within a few lines of ‘lastpass rocks’, the phrase used to conceal the master password in another buffer.



```
xn--mkru45i
abcd123!
kawellsfargoUser
AHSM
lastpass rocks
ected_storage]
ected_storage]
xn--nit225k
xn--ntsq17g
xn--pssu33l
```

Figure 19. Master password in cleartext (underlined red) typically within a few lines of ‘lastpass rocks’.

Strings encapsulated by a ‘<input hwnd=’ tag will allow us to enumerate all secret entries that have been interacted with. Below, Figure 20, is an example of extracting a private key to a bitcoin wallet.



```
'><input hwnd='2166702' id='255' pw='0' value='BitCoinWallet
Login:  secretWalletLogin
Password:  secretWalletPassword
private key E9873D79C6D87DC0FB6A5778633389 SAMPLE '/><input hw
```

Figure 20. Extracting a bitcoin private key from a locked instance of LastPass.

Conclusion:

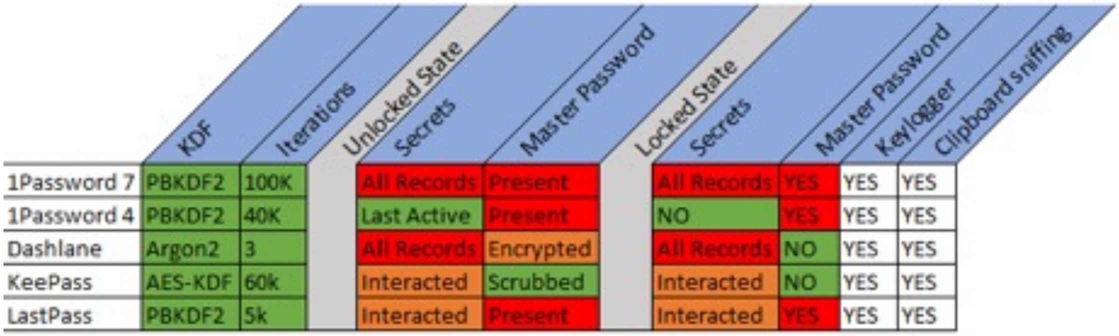
All password managers we examined sufficiently secured user secrets while in a 'not running' state. That is, if a password database were to be extracted from disk and if a strong master password was used, then brute forcing of a password manager would be computationally prohibitive.

Each password manager also attempted to scrub secrets from memory. But residual buffers remained that contained secrets, most likely due to memory leaks, lost memory references, or complex GUI frameworks which do not expose internal memory management mechanisms to sanitize secrets.

This was most evident in 1Password7 where secrets, including the master password and its associated secret key, were present in both a locked and unlocked state. This is in contrast to 1Password4, where at most, a single entry is exposed in a 'running unlocked' state and the master password exists in memory in an obfuscated form, but is easily recoverable. If 1Password4 scrubbed the master password memory region upon successful unlocking, it would comply with all proposed security guarantees we outlined earlier.

This paper is not meant to criticize specific password manager implementations; however, it is to establish a reasonable minimum baseline which all password managers should comply with. It is evident that attempts are made to scrub and sensitive memory in all password managers. **However, each password manager fails in implementing proper secrets sanitization for various reasons.**

The image below, Figure 21, summarizes the results of our evaluation:



	KDF	Iterations	Unlocked State		Locked State			
			Secrets	Master Password	Secrets	Master Password	Keylogger	Clipboard sniffing
1Password 7	PBKDF2	100K	All Records	Present	All Records	YES	YES	YES
1Password 4	PBKDF2	40K	Last Active	Present	NO	YES	YES	YES
Dashlane	Argon2	3	All Records	Encrypted	All Records	NO	YES	YES
KeePass	AES-KDF	60k	Interacted	Scrubbed	Interacted	NO	YES	YES
LastPass	PBKDF2	5k	Interacted	Present	Interacted	YES	YES	YES

Figure 21. Summary of each password managers security items we examined.

Keylogging and Clipboard sniffing are known risks and only included for user awareness, that no matter how closely a password manager may adhere to our proposed 'Security Guarantees', victims of keylogging or clipboard sniffing malware/methods have no protection. However, significant violations of our proposed security guarantees are highlighted in red. In an unlocked state, all or a majority of secret records should not be extracted into memory. Only a single one, being actively viewed, should be extracted. Also, in an unlocked state, the master password should not be present in either an encrypted or obfuscated form.

A locked running state that exposes interacted with or all records puts users' secret records unnecessarily at risk. **Most egregious is the presence of a master password in a locked state.**

It is unknown how widespread this knowledge is amongst adversaries. However, up to 60 million users of these password managers potentially are at risk of a targeted attack directed at the software that is meant to safeguard their secrets.

In our opinion, the most urgent item is to sanitize secrets when a password manager is placed into a locked state. Typically, most password managers place themselves into this locked state after a certain period of user inactivity, after this the process may remain indefinitely either until the OS is restarted, the process is terminated by the user, or the process restarts itself as part of a self-update workflow when a new version is published. This creates a large window of time in which secrets for certain password managers reside cleartext in memory and available for extraction.

In addition to providing a minimum set of guarantees users can rely on, creators of password managers should employ additional defenses to protect secrets by:

- Detecting or employing methods to, by default, thwart software based keyloggers
- Preventing secrets exposure in an unlocked state
- Employing hardware-based features (such as SGX) to make it more difficult to extract secrets
- Employing trivial malware and runtime process modification detection mechanisms
- Employing per-install binary scrambling during the install phase to make each instance a unique binary layout to thwart trivial and advanced targeted malware
- Limiting the traversal of secrets to OS provided APIs by implementing custom GUI elements and memory management to limit secrets exposure to well-known APIs that can be targeted by malware authors

End users should, as always, employ security best practices to limit exposure to adversarial activity, such as:

- Keeping the OS updated
- Enabling or utilizing well known and tested anti-virus solutions
- Utilizing features provided by some password managers, such as "Secure Desktop"
- Using hardware wallets for immediately exploitable sensitive data such as crypto currency private keys
- Utilizing the auto lock feature of their OS to prevent 'walk by' targeted malicious activity
- Selecting a strong password as the master password to thwart brute force possibilities on a compromised encrypted database file
- Using full disk encryption to prevent the possibility of secrets extraction in the event of crash logs and associated memory dumps which may include decrypted password manager data
- Shutting a password manager down completely when not in use even in a locked state (If using one that doesn't properly sanitize secrets upon being placed into a locked running state)

Future Research:

Password managers are an important and increasingly necessary part of our lives. In our opinion, users should expect that their secrets are safeguarded according to a minimum set of standards that we outlined as ‘security guarantees’. Initially our assumption and expectation were that password managers are designed to safeguard secrets in a ‘non-running state’, which we identified as true. However, we were surprised in the inconsistency in secrets sanitization and retention in memory when in a running unlocked state and, more importantly, when placed into a locked state.

If password managers fail to sanitize secrets in a locked running state then this will be the low hanging fruit, that provides the path of least resistance, to successful compromise of a password manager running on a user’s workstation.

Once the minimum set of ‘security guarantees’ is met then password managers should be re-evaluated to discover new attack vectors that adversaries may use to compromise password managers and examine possible mitigations for them.

References:

[1] "1Password," [Online]. Available: https://1password.com.

[2] "Dashlane," [Online]. Available: https://www.dashlane.com/.

[3] "KeePass," [Online]. Available: https://keepass.info/.

[4] "LastPass," [Online]. Available: https://www.lastpass.com/.

[5] T. Hunt. [Online]. Available: https://www.troyhunt.com/password-managers-dont-have-to-be-perfect-they-just-have-to-be-better-than-not-having-one/.

[6] "https://twitter.com/roustem," [Online].

[7] "https://blog.dashlane.com/10-million-users/," [Online].

[8] "https://keepass.info/help/kb/trust.html," [Online].

[9] "https://www.lastpass.com/," [Online].

[10] D. Florencio, C. Herley and P. C. v. Oorschot, "An Administrator’s Guide to Internet Password Research," [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2014/11/WhatsaSysadminToDo.pdf.

[11] T. L. Bras, "Online Overload – It’s Worse Than You Thought," [Online]. Available:

https://blog.dashlane.com/infographic-online-overload-its-worse-than-you-thought/.

[12] "Smart Taylor," [Online]. Available: https://smarttaylor.io/.

[13] Taylor. [Online]. Available: https://medium.com/smarttaylor/updates-on-the-taylor-hack-incident-8843238d1670.

[14] [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf.

[15] [Online]. Available: https://support.1password.com/pbkdf2/.

[16] "https://support.1password.com/pbkdf2/," [Online].

[17] "https://www.dashlane.com/download/Dashlane_SecurityWhitePaper_October2018.pdf," [Online].

[18] "https://keepass.info/help/base/security.html," [Online].

[19] "LastPass," [Online]. Available: https://blog.lastpass.com/2018/07/lastpass-bugcrowd-update.html/.

[20] J. Goldberg, "Using Intel's SGX to keep secrets even safer," [Online]. Available: https://blog.1password.com/using-intels-sgx-to-keep-secrets-even-safer/.

[21] "1Password support forum," [Online]. Available: https://discussions.agilebits.com/discussion/87834/intel-sgx-stopped-working-its-working-but-the-option-is-not-in-yet.

Values
Leadership
Methodology

Assessments
Programs
Training

Studies & Papers
Talks
Hacking Events

Contact

Careers