Pentest-Report Padlock.io 04.2016

Cure53, Dr.-Ing. M. Heiderich, Dipl.-Ings. A. Aranguren & A. Inführ, F. Fäßler, Filedescriptor

Index

Introduction

"Passwords are everywhere. We need them to access our computers, our email accounts and a ton of other services that we rely on. Unfortunately, passwords are a poor way of authentication. Strong passwords are hard to remember and the sheer number of different services makes it practically impossible to keep track of all of your credentials. Most people try to mitigate this by composing their passwords out of familiar things like names or birthdays, but these can be guessed easily and do not provide sufficient protection. Password reuse, which is also wide-spread, is even more dangerous and makes a lot of people easy targets for hackers. Password managers help you keep track of your passwords by providing a secure storage that is protected by a single master password. Good password managers keep your data safe by encrypting it with a strong cipher like AES."

From http://padlock.io/

This report documents the findings of a penetration test and source code audit carried out by the Cure53 team against the Padlock.io password manager application. Performed in the first half of April 2016 over the course of eleven days, the assignment involved five members of the Cure53 team.

The test's scope encompassed the app's Chrome extension, mobile applications and the server API. Since the tasks were guided by the white-box approach, the Cure53 testers had access to application sources. Github was used for sharing sources, as well as functioned as a tool for facilitating communication exchanges between the Padlock's development team and the Cure53 testers. Throughout the test, dialogue with the application maintainers was professional and productive, evidencing a commitment to improving the state of security at the Padlock.io.

The test had yielded a total of seventeen security issues, fifteen classified as vulnerabilities and two as general weaknesses. A particularly positive finding is that no issues that could be described as "Critical" with regard to their severity and impact have been found. Simultaneously, several issues marked as "High" paired with the rather numerous diverse issues that pose security risks, need to be taken into account going forward. In sum, the problems need to be quickly addressed and the recommendations should be implemented as soon as possible.

Scope

- **Chrome Browser Extension (Build shared)**
  - https://github.com/MaKleSoft/padlock
- **Mobile Apps (Built versions shared)**
  - https://github.com/MaKleSoft/padlock-cordova
- **Server API (Binary shared)**
  - https://github.com/MaKleSoft/padlock-cloud
- **All shared files are here**
  - https://drive.google.com/drive/folders/0B3mqh4_aOsKzQWtSdFdXYU1aRlU

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a 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. Each vulnerability is additionally given a unique identifier (e.g. *PIO-01-001*) for the purpose of facilitating any future follow-up correspondence.

PIO-01-001 Android, iOS and Extension: Insecure PRNG usage *(Medium)*

It was found that the mobile apps and the Chrome extension make use of a cryptographically insecure pseudo-random number generator (PRNG). An attacker could leverage this weakness to predict previous and past randomly generated usernames and passwords, as long as he has knowledge of a single randomly generated username or password. This is the case because, given the current PRNG implementation, there will be one randomly generated value per character of either a username or a password in question. For example, a known generated password that is 10 characters in length will provide the attacker with 10 sequential randomly generated values. From there, the attacker should be able to brute-force the PRNG seed and calculate the full sequence of the randomly generated values.

In file *src/rand.js*, the function *randomItem()* uses *Math.random()* to generate a random character, which is then used as a component of a random username and password. Since *Math.random()* is not cryptographically secure, an attacker can predict subsequent random strings as long as they know one of the usernames or passwords generated from it. What makes the implications of this issue worse, the JavaScript V8 engine in Chromium was proven to employ a broken implementation[1] of *Math.random()*, which makes the task of brute-forcing it much easier.

To mitigate this problem, it is recommended to use the CSPRNG[2] *window.crypto. getRandomValues()* from Web Crypto API.

PIO-01-002 API: Email enumeration and unauthenticated reset initiation *(Medium)*

It was found that the API endpoint, normally intended for resetting passwords on the cloud server, can be used to enumerate email addresses. If an email address does not exist, the API will respond with a *"User does not exist"* error. This endpoint fails to require the Authorization header for sending the reset email, which makes it easy to spam a user with delete requests. In effect, one accidental click from a user could in fact delete their password storage.

The following examples demonstrate this trivial issue:

---

[1] https://medium.com/@betable/tifu-by-using-math-random-f1c308c4fd9d#.9s2sddydi
[2] https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator

Example 1: Non-existing user

Request:
```
DELETE https://cloud.padlock.io/non-existent@cure53.de
```

Response:
```
HTTP/1.1 404 Not Found
User non-existent@cure53.de does not exists
```

Example 2: Existing user

Request:
```
DELETE https://cloud.padlock.io/fabian@cure53.de
```

Response:
```
HTTP/1.1 202 Accepted
```

It is recommended for the API to provide generic messages regardless of user existence. In addition, sensitive user actions, such as deleting the password vault from the cloud, should necessitate a server authentication.

Retest notes: The Cure53 team has verified a fix for this issue in the branch made available by the maintainer for this pentest.

PIO-01-003 Android: Lack of screen capture protections *(High)*

It was found that the Android app does not currently implement screen capture protections. In the latest Android versions this allows malicious apps with root privileges to take screenshots or even full videos. This takes place in the background, running without any user-interaction whatsoever. The ways in which malicious apps usually gain root privileges entail either simply prompting the user on a rooted phone, or exploiting a known vulnerability on an outdated device. Malicious apps without root privileges can also accomplish this but require a user-prompt, which might only be shown once, assuming that the user taps on the "*Do not show again*" message. As a consequence, all usernames and passwords displayed while the Padlock app is in the foreground could in fact be stolen by malicious apps using screenshots as a side-channel.

It is relatively easy to verify this issue with the use of the following commands from a non-root *adb* shell. This needs to take place while the Padlock app is open and displaying some sensitive information, though yielding a normal screenshot because screen capture protections are currently not implemented:

Commands:
```
adb shell screencap -p /mnt/sdcard/screenshot1.png
adb pull /mnt/sdcard/screenshot1.png
```

Since the passwords are not even obfuscated on the UI by default, this screenshot leakage weakness can be used to steal a 'username and password' pair in a single

screenshot. This is possible in both cases of the user adding a new entry, and the user just viewing an entry:
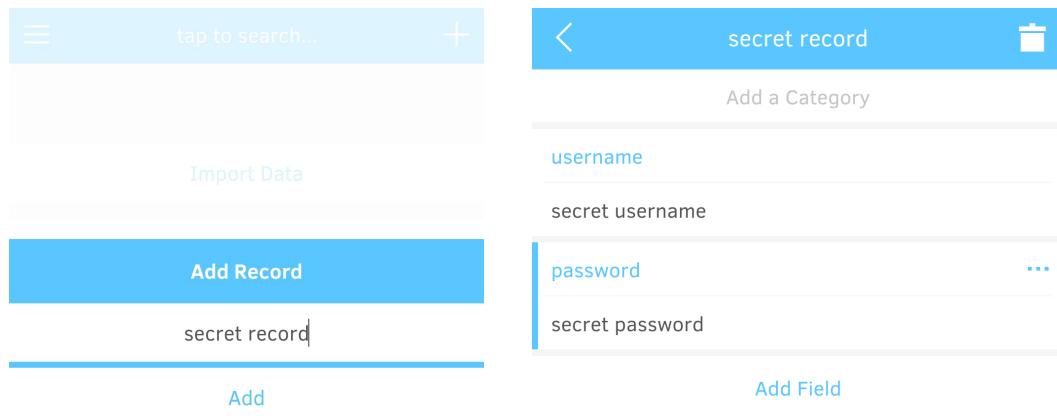
Fig.: Record, Username and Password leak when the user adds/views entries

The verification of the permission prompt required by malicious apps without root privileges can be observed via the popular Android screenshot and video recording apps. Some of the apps that do not require root privileges are *AZ Screen Recorder - No Root*[3] or *Screenshot Easy*[4]. Please note that if the user taps on the "*Don't show again*" checkbox, no further prompts will ever be shown again to the user prior to a video recording or a screenshot coming to fruition:
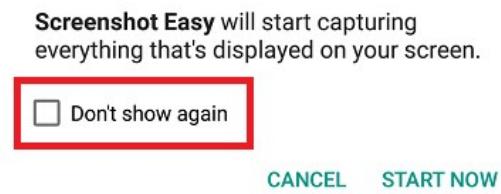
Fig.: Permission prompt required by non-root apps

---

[3] https://play.google.com/store/apps/details?id=com.hecorat.screenrecorder.free&hl=en
[4] https://play.google.com/store/apps/details?id=com.icecoldapps.screenshoteasy

As mentioned above, a more pressing concern is that, given the reality of the outdated and compromised Android ecosystem[5], absolutely no permission prompts will be shown to the user when an app with root privileges captures everything that is happening on the phone (i.e. record a full video or continuously take screenshots). In the paper *Security Metrics for the Android Ecosystem*[6]*,* published on October 2015, researchers from the *University of Cambridge* showed that root privileges can in fact be gained on 87.7% of Android phones through a security vulnerability. The authors specifically caution:

> *"We find that on average 87.7% of Android devices are exposed to at least one of 11 known critical vulnerabilities"*

What further impacts the severity of this issue is that, on a rooted phone, a malicious app can simply prompt the user for root privileges, which is a common practice in Android. Many apps like *Easy Screenshot*[7] and others behave this way. By default, upon the root privileges prompt, the root privileges will simply be indefinitely granted if the user just taps on "Grant" :
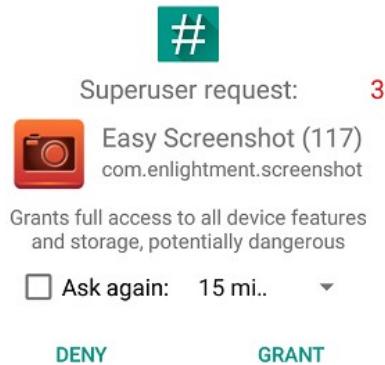
Fig.: Root privileges are granted indefinitely by default

What needs to be underscored is that a malicious app with root privileges can also fake interaction using commands such as "*input tap <x> <y>*", etc. Therefore, with the ability to view what is displayed on the screen, malicious apps with root privileges can also interact with the UI. The latter means that malicious apps can fully impersonate users when they are not actively using their phones and leave the Padlock app unlocked and open in the foreground.

In the context of the tested application, which handles sensitive information, it is recommended to ensure that all webviews have the Android *FLAG_SECURE* flag[8] set. This will guarantee that even apps running with root privileges cannot capture the information displayed by the app. It is advised that a fix similar to the following is implemented and ideally treated as a base activity that all other activities inherit:

---

[5] https://public.gdatasoftware.com/Presse/Publikatio...s/G_DATA_MobileMWR_Q1_2015_US.pdf
[6] https://www.cl.cam.ac.uk/~drt24/papers/spsm-scoring.pdf
[7] https://play.google.com/store/apps/details?id=com.enlightment.screenshot&hl=en
[8] http://developer.android.com/reference/android/view/Display.html#FLAG_SECURE

Proposed Fix:

```
public class BaseActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /**
         * approach 1: create a base activity and set the FLAG_SECURE in it,
         * Extend all other activities, Fragments from this activity
         */
        getWindow().setFlags(LayoutParams.FLAG_SECURE,
                LayoutParams.FLAG_SECURE);
    }
```

If the proposed solution is considered unfeasible, another possible approach could be to amend the *onCreate* method of the Cordova View on Android, making a corrective adjustment depicted below:

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getWindow().setFlags(LayoutParams.FLAG_SECURE,
        LayoutParams.FLAG_SECURE);
}
```

PIO-01-005 Server: POODLE and DH key exchange weaknesses *(Medium)*

It was found that the *cloud.padlock.io* and *cloud-dev.padlock.io* servers are vulnerable to the POODLE attack[9] and implement weak Diffie-Hellman key exchange parameters, which may lead to a *Logjam* attack.[10]

This can be trivially verified with the use of the relevant SSL Labs link:
https://www.ssllabs.com/ssltest/analyze.html?d=cloud.padlock.io&hideResults=on
https://www.ssllabs.com/ssltest/analyze.html?d=cloud-dev.padlock.io&hideResults=on

Improving the TLS configuration is a necessary step for solving this problem. The OWASP Transport Layer Protection Cheat Sheet[11] provides detailed instructions on how to rollout TLS securely, while the Duraconf templates[12] supply a great starting point for introducing advancements in this area.

Furthermore, the SSL Labs test facility[13] can be used to examine the TLS configuration, keeping in mind that acquiring an A-grade result should be considered the correct objective. Finally, a permanent redirect should be implemented from port 80 to port 443

---

[9] https://www.us-cert.gov/ncas/alerts/TA14-290A

[10] https://weakdh.org/

[11] https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

[12] https://github.com/ioerror/duraconf

[13] https://www.ssllabs.com/ssltest/

CURE·53

and the HSTS header[14] needs to be sent along all requests to mitigate potential channel downgrade attacks.

PIO-01-006 Android and iOS: Lack of Pinning *(Medium)*

It was found that the Android and iOS apps do not currently implement any form of certificate or public key pinning whatsoever. A malicious attacker with a certificate trusted by the relevant iOS and Android certificate stores (i.e. most governments and attackers with considerable resources) could therefore intercept network communications and hijack user sessions. In effect they could become able to interact with the Padlock API as the logged-in user, without any user-warnings occurring at the application level.

The following examples illustrate some of the information available to the attackers once communications are intercepted through this vulnerability.

Example 1: User email and authentication token in every API request

Every API request contains an *Authorization* header which reveals the email of the user contacting the server.

API Request:
```
GET https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
Authorization: AuthToken abraham+1@cure53.de:26e31c6f-2122-42d2-b749-
60a333f333fb
[...]
```

Example 2: Email and token leak on initial setup

This example shows how the user's email and authentication token are processed when the account is initially set up. This mechanism will leak the token relayed to the user by the API server:

Request:
```
POST https://cloud-dev.padlock.io/auth/ HTTP/1.1
[...]
email=abraham%40cure53.de
```

Response:
```
{"email":"abraham@cure53.de","token":"af28e2de-3ddc-44be-ae19-efe907f52579"}
```

API server are leaked, inclusive of the encrypted entries. This takes place when the user synchronizes with the API server.

---

[14] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

Example API Request:

```
PUT https://cloud-dev.padlock.io/store/ HTTP/1.1
Authorization: AuthToken abraham+1@cure53.de:26e31c6f-2122-42d2-b749-
60a333f333fb
[...]
```

```
{"cipher":"AES","mode":"ccm","iv":"NTn1QBqBUYQ09qANRJ60Vw==","salt":"xNF27i+ANO5
vLP/8U//4lg==","keySize":256,"iter":10000,"ct":"nVVX6oAylP2ej1ZXXm0Exn9I/pLW6Jk/
o9qFdVl9YnzxIrGNNT+HM6tjDP3pn1R8NNVheCvzu8jX5sCwHXWnmBJoUXKAqJMZg2TWkBX8Bg6B3VUe
0dHaJ2ke7WORluTkmVxCTZ55QtmyNkyAYndixeJ5pH/eW89zfAJD4+LXSgbMm8B/2UOixdFyX5si28L9
qGhYUzudjvZ2K+Af21ptrMSJ1iXVOesOJixPhjRmtL9ahtYdRjOepHYZ/Q+XUczxka4UoJf6ONgWh3cT
uX9gs7+pPvLPsTefF8tOBrWOGa4eQ0ceNbzETLfmhclzlbyssAYhSK/NgV4ASpjt42EyzeE3I50=","a
data":"02ofmsVhZqdRp5n5hH67sQ==","ts":64}
```

Although neither user password nor the password vault travel in clear-text over the network, this weakness is enough for a Man-In-The-Middle (MitM) to interact on the user's behalf. This signifies an attempt to crack the password online or offline and, quite possibly, performing some destructive actions like deleting the password vault from the API server.

It is recommended to implement pinning on both the Android and iOS apps, at least for the default *padlock.io* servers. While this will not prevent researchers from inspecting API calls, it will protect average users from governments and well-funded attackers able to forge the widely trusted SSL certificates. For more information about pinning, including Android and iOS examples, please see the *OWASP Pinning Cheat Sheet*[15] and the *OWASP Certificate and Public Key Pinning*[16] technical guide.

PIO-01-007 Activation link spoofing via Host header *(Low)*

A minor problem was found within the process of sending the activation links. Specifically the Host header supplied by the client was discovered to be directly used as part of the activation URL in the email sent to the user. This may lead to HTTP Host header attacks.[17] If the user clicks on the link, the activation token would be leaked to the attacker.

Affected Code:
```
err = app.Templates.ActivateAuthTokenEmail.Execute(&buff, map[string]string{
        "email":           authToken.Email,
        "activation_link": fmt.Sprintf("%s://%s/activate/?v=%d&t=%s",
schemeFromRequest(r), r.Host, version, actToken),
})
```

Since the Host header is a user-supplied value and activation requires no authentication, attackers can poison the activation link. They can do so by sending an activation request with a victim's email address using a forged Host header. When the victim receives the activation email, the intended host will be changed to an attacker-controlled destination, thus allowing the attacker to steal the activation token once the victim clicks on the link.

---

[15] https://www.owasp.org/index.php/Pinning_Cheat_Sheet
[16] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning
[17] http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html

Request (the server's host is 127.0.0.1:3000):
```
POST /auth/ HTTP/1.1
Host: attacker.com
Connection: keep-alive
Content-Length: 30
Accept: application/vnd.padlock;version=1
Origin: chrome-extension://clcklbifedcbhmhjhgghcppcpoebccic
Require-Subscription: NO
Content-Type: application/x-www-form-urlencoded

email=filedescriptor@cure53.de
```

Activation email:
```
Hi there!

You are receiving this email because you requested to a device with the Padlock
Cloud account filedescriptor@cure53.de. To complete the process, simply visit
the link below:

https://attacker.com/activate/348244e7-832a-482d-be75-d2072deef504

Best,
The Padlock Team
```

It was later confirmed that this attack depends on the cloud server hosting configuration. For example, the same attack against the *cloud-dev.* server does not work because supplying a crafted Host header fails to reach the *auth* endpoint. This stems from the fact that the *cloud.* and *cloud-dev.* servers benefit from a configuration which makes this issue not exploitable. However, the attack was confirmed against the penetration-test branch and would work against servers that are hosted on their own.

Example 1: With the correct Host header

Request:
```
curl -k --request POST https://cloud-dev.padlock.io/auth/  --header "Accept:
application/vnd.padlock;version=1"  --header "Content-Type: application/x-www-
form-urlencoded"  --header "Host: cloud-dev.padlock.io"  --include --data
'email=abraham%2B2%40cure53.de'
```

Response:
```
HTTP/1.1 202 Accepted
[...]

{"email":"abraham+2@cure53.de","token":"aecd2227-37e8-43eb-9dac-059e31c596a6"}
```

Example 2: With a crafted Host header

Request:
```
curl -k --request POST https://cloud-dev.padlock.io/auth/  --header "Accept:
application/vnd.padlock;version=1"  --header "Content-Type: application/x-www-
```

```
form-urlencoded"  --header "Host: cure53.de"  --include --data 'email=abraham
%2B2%40cure53.de'
```

Response:
```
HTTP/1.1 404 Not Found
[...]

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /auth/ was not found on this server.</p>
</body></html>
```

It is recommended to either validate the value obtained from the Host header, or, alternatively, to fixate it with a pre-defined configuration value.

PIO-01-008 Android: Complete lack of Tapjacking mitigations *(Medium)*

It was found that the Padlock app currently fails to mitigate tapjacking attacks. This allows malicious apps with neither root privileges nor special permissions to be able to fool users into tapping and typing. As a result they see screens controlled by the malicious app while taps and keystrokes reach the Padlock app. In this case, since the Padlock app will auto-lock when it goes into the background, this issue could be used to exploit PIO-01-005 by rendering a transparent overlay on top of the Padlock app. As a consequence, the user is led to believe the original malicious app has been closed while in fact it is rendered on top. Continuously, the user interacts with the Padlock app while screenshots are taken for everything happening underneath.

This issue was verified on a rooted phone running Android 6.0.1. The crafted tapjacking PoC APK opens the Padlock app in the background while the user sees something else in the foreground. A short video was also recorded and further elaborates on how taps to the PoC screen (rendered on top) reach the Padlock app underneath.

Aside for the above issue, it was also discovered that global searches for tapjacking protections return no matches in the source code provided, hence confirming a complete lack of these protections on all views and buttons used by the app. It is recommended to implement the *filterTouchesWhenObscured*[18][19] attribute on the Android Cordova WebView[20] to ensure that taps from potential malicious apps rendered on top are ignored.

---

[18] http://developer.android.com/reference/android/view/View…rTouchesWhenObscured(boolean)

[19] http://developer.android.com/reference/android/view/View...#attr_android:filterTouchesWhenObscured

[20] https://cordova.apache.org/docs/en/latest/guide/platforms/android/webview.html

PIO-01-009 Android, iOS and Extension: Permanent sync DoS *(High)*

It was found that the Android, iOS, and Chrome extension apps will blindly attempt to decrypt the blob provided by the server, using the number of iterations provided, regardless of how high this number is. When the apps receive a crafted reply with a very high number of iterations from the server, they will attempt to decrypt indefinitely. The sync function will never work again, even after the phone is restarted. In other words, once the sync starts, it will never finish.

In addition it was discovered that any non-*JSON* data can break the sync functionality as well. This is achieved by pushing arbitrary data via a *PUT* request to the server. During the sync process, the application first retrieves the data from the server to compare it with the local database. Because *JSON.parse* instruction will fail with the corrupted data, the application is no longer able to send new updates to the server.

This attack could be executed using a broad number of attack vectors. Some of the options include taking advantage of a malicious server, a compromised server, an attacker able to manipulate network communications (i.e. using PIO-01-006), or a client-side XSS vulnerability. Selected specifics are discussed below.

Attack Variant 1-a: Store the iterations on the server first

The most effective way to permanently break the sync is to store a high number of iterations on the server first. This could be obtained by, for example, using a client-side XSS issue on the mobile apps, or by manipulating network communications in a MitM scenario. A malicious or compromised server could directly modify this in the database.

For the sake of brevity, the MitM scenario will be used to illustrate this issue here. A normal API request to store the user-encrypted blob looks like this:

Request:
```
PUT https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
{"cipher":"AES","mode":"ccm","iv":"kxwbdh07IP004icta5cmGg==","salt":"vkUoqPsu+w7
c+1h+OgN6ZQ==","keySize":256,"iter":10000,"ct":"nppHMnv7aaYFaUYjRqtn1jbbD6sCkxsd
8gN45/gLLyQCCec8RMt0v247St+nuqd6IBHnFfBUoin6j6q2KdyaI65/U/jse/He4+y3s6aWJi4gQgi2
FmEvGgb9dYIaCnznd7hO/cD7t8cEyY9CEMfG737cB5oQDnD7XeNu8P/xbb42mURnVntafGuy7szvhIAx
8s+630rwpGI5Nzhlv6UIscM66ssZ8oCPkz4KvZmsVB+s5n2hXeeMh1SZhB60mT9CtyIaBUNs5Qh8AwZk
vgVhJ7LQ5ngxNhERSSmxFgSdzcuoP8wCvXwh7l2HQ4H3+fe8X45Tbugq8CzoYvJ0QwimnZXo8SR3Q/u9
edN1cB0PHqm/6zFS/Sumk4+QkU6LAyAIh2g1eOoPWKUBV3ft056GF5yvsbBRv4fBxgF5/JCcixHv1h0f
66j+ttiPONG+omLeeUb0+FEXl1o=","adata":"a3tN8XGwnc4FYQS8NAfC0g==","ts":64}
```

A malicious attacker with the ability to manipulate network communications can change the number of iterations to a much higher number, for example:

Request:
```
PUT https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
{"cipher":"AES","mode":"ccm","iv":"kxwbdh07IP004icta5cmGg==","salt":"vkUoqPsu+w7
c+1h+OgN6ZQ==","keySize":256,"iter":10000000000000,"ct":"nppHMnv7aaYFaUYjRqtn1jb
bD6sCkxsd8gN45/gLLyQCCec8RMt0v247St+nuqd6IBHnFfBUoin6j6q2KdyaI65/U/jse/He4+y3s6a
WJi4gQgi2FmEvGgb9dYIaCnznd7hO/cD7t8cEyY9CEMfG737cB5oQDnD7XeNu8P/xbb42mURnVntafGu
y7szvhIAx8s+630rwpGI5Nzhlv6UIscM66ssZ8oCPkz4KvZmsVB+s5n2hXeeMh1SZhB60mT9CtyIaBUN
s5Qh8AwZkvgVhJ7LQ5ngxNhERSSmxFgSdzcuoP8wCvXwh7l2HQ4H3+fe8X45Tbugq8CzoYvJ0QwimnZX
o8SR3Q/u9edN1cB0PHqm/6zFS/Sumk4+QkU6LAyAIh2g1eOoPWKUBV3ft056GF5yvsbBRv4fBxgF5/JC
cixHv1h0f66j+ttiPONG+omLeeUb0+FEXl1o=","adata":"a3tN8XGwnc4FYQS8NAfC0g==","ts":6
4}
```

From that moment onwards, every time the phone requests to sync with the server, it will get a reply like this:

Request:
```
GET https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
```

Response:
```
{"cipher":"AES","mode":"ccm","iv":"kxwbdh07IP004icta5cmGg==","salt":"vkUoqPsu+w7
c+1h+OgN6ZQ==","keySize":256,"iter":10000000000000,"ct":"nppHMnv7aaYFaUYjRqtn1jb
bD6sCkxsd8gN45/gLLyQCCec8RMt0v247St+nuqd6IBHnFfBUoin6j6q2KdyaI65/U/jse/He4+y3s6a
WJi4gQgi2FmEvGgb9dYIaCnznd7hO/cD7t8cEyY9CEMfG737cB5oQDnD7XeNu8P/xbb42mURnVntafGu
y7szvhIAx8s+630rwpGI5Nzhlv6UIscM66ssZ8oCPkz4KvZmsVB+s5n2hXeeMh1SZhB60mT9CtyIaBUN
s5Qh8AwZkvgVhJ7LQ5ngxNhERSSmxFgSdzcuoP8wCvXwh7l2HQ4H3+fe8X45Tbugq8CzoYvJ0QwimnZX
o8SR3Q/u9edN1cB0PHqm/6zFS/Sumk4+QkU6LAyAIh2g1eOoPWKUBV3ft056GF5yvsbBRv4fBxgF5/JC
cixHv1h0f66j+ttiPONG+omLeeUb0+FEXl1o=","adata":"a3tN8XGwnc4FYQS8NAfC0g==","ts":6
4}
```

When the iOS or Android app receives a reply like the one depicted, then an indefinite process of decryption will begin. This will consume a high percentage of CPU resources and the password vault blob will never be decrypted since it was encrypted with a different number of iterations. As a consequence, the usage of the sync becomes indefinitely impossible after this. Even if the phone is restarted or the local password vault is reset, the problem will not be solved. The only way in which a user can fix this issue is to go to *Settings* / *Padlock Cloud* / *Reset Data* (which will delete the entire password vault in the cloud server), then enable the Padlock Cloud again for the same email address, only then lastly restarting the phone.

Attack Variant 1-b: Store the corrupted data on the server side
This variant differs from the one described above as it already causes an error in the *JSON* parsing on the client-side. The client always retrieves data from the server before pushing any changes via a *PUT* request. This is the reason for the client-side being no longer able to sync any changes:

Request:
```
PUT https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
JUST_NON_JSON_DATA
```

Request:
```
GET https://cloud-dev.padlock.io/store/ HTTP/1.1
[...]
```

Response:
```
HTTP/1.1 200 OK
[...]
JUST_NON_JSON_DATA
```

Attack Variant 2: Single malicious reply
In this scenario the attacker, using a malicious server, a compromised server or network traffic manipulation, simply spoofs a single reply (or all replies) from the API sync server. The final result is concurrent to the one for Variant 1, as a sync will never end. While it is similar to the previous variant, it does not make storing of the high number of iterations on the server first a necessary precondition.

Overall, in hopes of mitigating the problems described for this issue, it is recommended to reduce the level of trust that the client must place on the server. For that purpose, Padlock clients could be modified so that they cryptographically sign their password vault modifications before sending them to the server. When Padlock clients retrieve the vault from the server, decryption should not be attempted if the signatures do not match. Furthermore, the user should be warned about potential compromise or tampering from the server-side.

PIO-01-010 iOS: Possible hijacking via unprotected files at rest *(Medium)*

It was found that the Padlock iOS app fails to take full advantage of the built-in iOS platform protections, which provide encryption at rest and allows caching of URLs in clear-text. Although other files are protected, the URL caching database contains the Authorization HTTP header in clear-text, which is enough to impersonate a user. The fact that the header never expires should also be noted.

This issue can perhaps be best described with the output of the *tar* command from a jail-broken device at a stage where an iPhone was on the lock screen. In this illustrative case all files fail to copy, with the exception being the *nsurlcache* files, which cache HTTP requests and response in clear-text:

Command:
```
tar cvfz files_locked.tar.gz *
```
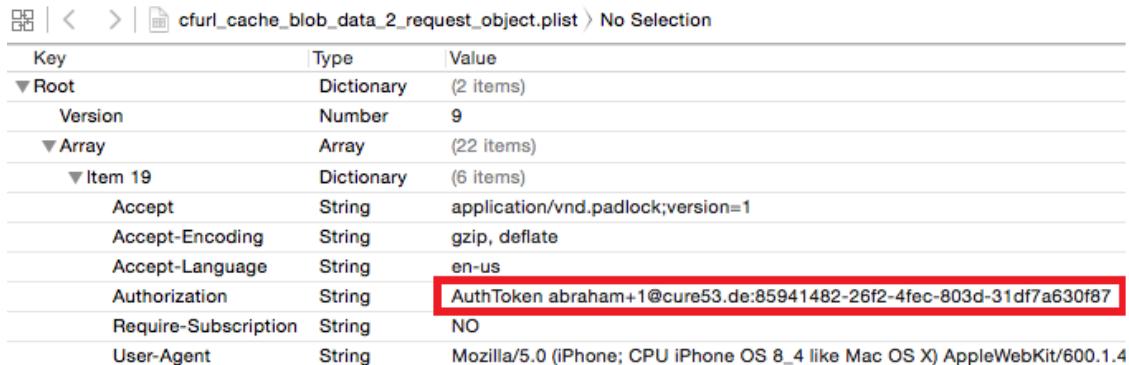
Output:
```
Documents/
Library/
Library/Caches/
```

```
Library/Caches/Snapshots/
Library/Caches/Snapshots/com.maklesoft.padlock/
Library/Caches/Snapshots/com.maklesoft.padlock/com.maklesoft.padlock/
tar:
...
tar: Library/Caches/com.maklesoft.padlock/com.apple.opengl/shaders.data: Cannot
open: Operation not permitted
tar: Library/Caches/com.maklesoft.padlock/com.apple.opengl/shaders.maps: Cannot
open: Operation not permitted
Library/Caches/com.maklesoft.padlock/nsurlcache/
Library/Caches/com.maklesoft.padlock/nsurlcache/Cache.db
Library/Caches/com.maklesoft.padlock/nsurlcache/Cache.db-shm
Library/Caches/com.maklesoft.padlock/nsurlcache/Cache.db-wal
Library/Preferences/
tar: Library/Preferences/com.maklesoft.padlock.plist: Cannot open: Operation not
permitted
Library/WebKit/
Library/WebKit/LocalStorage/
tar: Library/WebKit/LocalStorage/file__0.localstorage: Cannot open: Operation
not permitted
tmp/
tar: Exiting with failure status due to previous errors
```

The *nsurlcache* SQLite database was inspected. As a result, on one of the records in the *cfurl_cache_blob_data* table, a *.plist* file containing the HTTP request sent to the API was found to include the Authorization header:

Fig.: Authorization token leak via nsurlcache on a locked device

In order to solve this problem it is recommended to implement the *NSFileProtectionComplete* entitlement at the app level[21].

---

[21] https://developer.apple.com/library/ios/documentation/iP...App/StrategiesforImplementingYourApp.html

PIO-01-011 API: Possible Denial of Service via replay *(High)*

During testing of the mobile apps it was found that certain unauthenticated requests can be replayed indefinitely. Although this weakness could be abused to minimally increase the chances of finding a valid Authorization token, a possible better use for malicious purposes could be to cause a Denial of Service (DoS) of the web server: For each unauthenticated request the server will email a new token to the user.

An attacker can therefore force the webserver to send emails indefinitely and constantly, even for a single email address. With a resulting large email output, the server would have quickly been banned by email providers. This would in turn lead to issues with sending legitimate users emails and, very possibly, a crash if the attacker coordinated this attack from multiple hosts simultaneously.

The following rudimentary script was crafted to demonstrate this issue:

File:
token_generator.sh

Code:
```
#!/usr/bin/env bash

while [ 1 ]; do
        curl -i -s -k --noproxy cloud-dev.padlock.io --request PUT
https://cloud-dev.padlock.io/auth/  --header "Accept:
application/vnd.padlock;version=1" --data 'email=abraham%40cure53.de'
done
```

Leaving this script running for about one minute resulted in 291 emails sent in 60 seconds. This translates to about 5 emails per second without any performance optimizations in place:

Command:
```
time bash token_generator.sh > output.txt
^C
```

Output:
```
real    1m0.109s
```

Command:
```
grep abraham@cure53.de output.txt | wc -l
```

Output:
```
291
```

Speed Rate Without Optimizations:
At a rate of 291 tries per minute, 419,040 tries should be expected in one day. This means almost 3 million attempts in one week.

Speed Rate With Optimizations:
As a quick test following the script above, yet running each curl command in the background (i.e. adding a "&" at the end of the line where curl is executed), resulted in as many as 2664 emails sent in 20 seconds. In other words, this signifies about 8000 emails sent over a course of a single minute. At this speed rate, 11.5 million emails should be expected in 24 hours, while sending 80.5 million emails would be achieved in one week. All calculations rely on a pattern of a single destination email address and only one IP address for an attacker.

However, it cannot be discarded that an attacker would spread this replay attack from multiple IP addresses at the same time for additional effectiveness, especially when his or her goal is to take down the server. In addition to this, it seems plausible that with so many emails sent, the upstream email provider would have blocked the *padlock.io* server from operating. Hence all email functions would be rendered unavailable to all users who remained able to use the service. The latter would persist even after the attack has been finished and last until the email provider restores the service.

It is recommended to implement some throttling mechanisms to avoid these issues. For example, the number of emails that can be sent to any given user and from any given IP should be limited to encompass only what is deemed reasonable. Consideration could be given to additional security measures like limiting the number of pairing device requests per email address to only one at any given time, wherein the user has to accept or deny the request from the email before a new request can be sent from the server.

PIO-01-012 API: Email enumeration via invalid tokens *(Medium)*

As already mentioned under the PIO-01-006, the *Authorization* header contains the email address as well as the authentication token. It was discovered that by manipulating the email address it is possible to enumerate registered users. Before the token is checked, the application looks up whether the sent email address belongs to an existing user. If not, an error "*User does not exists*" is thrown.

This issue can be trivially verified with the following commands:

Command:
```
curl -H "Authorization: ApiKey alex@cure53.de:f" https://cloud.padlock.io
```

Output:
```
The provided key was not valid
```

Command:
```
curl -H "Authorization: ApiKey random@cure53.de:f" https://cloud.padlock.io
```

Output:
```
User random@cure53.de does not exists
```

It is recommended for the server to always consistently respond with the same error message, regardless of an invalid email address or an invalid token. This makes sure that an attacker is no longer able to retrieve information about the registered email addresses.

Retest notes: The Cure53 team has verified a fix for this issue in the branch made available by the maintainer for this pentest.

PIO-01-013 Android and iOS: User information stored in clear-text *(Medium)*

It was found that both the Android and the iOS mobile apps store the sync URL, sync key, email address, and crypto parameters used to encrypt the password vault in clear-text on the filesystem. As demonstrated in other findings of this report, like PIO-01-009 or PIO-01-014, the email address and the sync token are enough to delete or incorrectly encrypt the password vault on the server. This effectively renders the password vault stored in the server unusable.

This issue is limited in impact because the data is stored on the private app storage on both Android and iOS. However, apps with root privileges will be able to easily access this information by browsing the filesystem. Moreover, Android is further impacted since the backup element is not specified in the manifest, meaning that it defaults to true. This information could also be extracted by malicious attackers with physical access to the device, particularly on the phones that have USB debugging enabled[22]. On Android the relevant *LocalStorage* SQLite database is stored in the following location:

File:
/data/data/com.maklesoft.padlock/app_xwalkcore/Default/Local Storage/file__0.localstorage

On iOS the location will be similar to the following:

```
/private/var/mobile/Containers/Data/Application/F53D3415-959D-48FF-8486-
F6E1E4B59D84/Library/WebKit/LocalStorage/file__0.localstorage
```

In both cases of iOS and Android the database contains a "*Settings*" and a *"coll_default"* *JSON* blobs which reveal this information:

Example Settings BLOB:
```
{"sync_host_url":"https://cloud-
dev.padlock.io","sync_custom_host":true,"sync_email":"abraham@cure53.de","sync_k
ey":"7558f4af-d66f-42fb-9239-
4326ad99ae85","sync_device":"","sync_connected":true,"sync_auto":true,"sync_read
only":false,"default_fields":
["username","password"],"obfuscate_fields":false,"showed_backup_reminder":0,"syn
c_require_subscription":false}
```

---

[22] http://developer.android.com/reference/android/R.attr.html#allowBackup

Example Coll_Default BLOB:

```
{"cipher":"AES","mode":"ccm","iv":"YQdHDp6nPLTTUmFzwo+R3A==","salt":"PTMHqy8KmrK
FxVS9zcJMiA==","keySize":256,"iter":10000,"ct":"l6TKoTOF4Pxe3A==","adata":"Obqf4
4vE44aA2+GvEHmJ6A==","ts":64}
```

Sensitive information that could be used to impersonate or delete the password vault for a user absolutely requires to be better-protected on the device. For example, the iOS *KeyChain*[23] could be used for this purpose on iOS, and the *Android KeyStore*[24] on Android.

PIO-01-014 Android and iOS: Use of external clear-text HTTP URLs *(Medium)*

It was found that the Android and iOS apps will themselves open external URLs over clear-text HTTP whenever the user taps on certain locations. A malicious attacker with the ability to modify network communications could abuse this weakness to forge a webpage that looks as close as possible to the Padlock passphrase screen. This signifies a potential capacity to fool some users into sending their passphrase to the attacker. The attack could be combined with PIO-01-017, which gains access to the encrypted user-vault, and, in result, yield an ability to decrypt the password vault as well.

Although these external URLs are opened using the iOS or Android browsers, they could be viewed as coming from the Padlock app by a less-advanced or technically-versed user. From their perspective, a Padlock login could make sense in this scenario. Other attack possibilities include attempting to break into the user's phone (i.e. leveraging the outdated Android ecosystem problem, as mentioned in PIO-01-003) or prompting the user to ring premium numbers.

The following external URLs were found to be opened insecurely by both the iOS and Android apps:

File: *assets/www/index.js*
Affected Code:
```
14335   _openWebsite: function() {
14336       window.open("http://padlock.io", "_system");
14337   },
14338   _sendMail: function() {
14339       var url = "mailto:support@padlock.io";
14340       window.open(url, "_system");
14341   },
14342   _openGithub: function() {
14343       window.open("http://github.com/maklesoft", "_system");
14344   },
14345   _openHomepage: function() {
14346       window.open("http://maklesoft.com/", "_system");
```

---

[23] https://developer.apple.com/library/mac/documentation/Security/Conceptu...eTasks/iPhoneTasks.html
[24] http://developer.android.com/training/articles/keystore.html

It is recommended to ensure that all external URLs start with HTTPS://. Moreover, wherever Padlock controls the servers, the SSL configuration should be hardened using the *OWASP Transport Layer Protection Cheat Sheet*[25].

PIO-01-015 Android, iOS and Extension: XSS via innerHTML *(Info)*

It was discovered that the Android, iOS and Chrome extension apps use *innerHTML* in an insecure way. The *_openForm* function is employed for opening forms and filling their contents. The *title* parameter is loaded into a HTML form via *innerHTML*. This makes it possible to inject malicious HTML code into the form as soon as the *title* parameter contains user-controlled data

Code Line:
https://github.com/MaKleSoft/padlock/blob/pentesting/src/components/app/app.js#L662

Code:
```
_openForm: function(components, title, submitCallback, cancelCallback,
allowDismiss) {
        [...]
        var dialog = this.$.formDialog1.isShowing ? this.$.formDialog2 : this.
             $.formDialog1;
        dialog.allowDismiss = allowDismiss !== false;
        // Get form and title element associated with the selected form
        var form = Polymer.dom(dialog).querySelector("padlock-dynamic-form");
        var titleEl = Polymer.dom(dialog).querySelector(".title");
        // Close both forms
        this.$.formDialog1.open = this.$.formDialog2.open = false;
        // Update title
        titleEl.innerHTML = title || "";
```

It was discovered that it is possible to reach the vulnerable code with user-controlled data on two occasions. Firstly, a category name which contains HTML code will get parsed as soon as a user tries to edit its name. Secondly, when a user tries to connect the extensions with an email address which contains HTML code, it will be parsed and displayed in the following dialog:

Affected Code:

https://github.com/MaKleSoft/padlock/blob/pentesting/src/components/categories-view/categories-view.js#L58

```
_editCategory: function(e) {
            var category = e.model.item;
            this.fire("open-form", {
            title: "Edit '" + category + "'",
```

---

[25] https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

Affected Code:
https://github.com/MaKleSoft/padlock/blob/pentesting/src/components/start-view/start-view.js#L202

```
_promptConnecting: function() {
      this.fire("open-form", {
      title: "Almost done! An email was sent to " + this.settings.sync_email +
      " with further instructions. Hit 'Cancel' to abort the process.",
```

The impact of this vulnerability is highly reduced because of the current Content-Security Policy in place. It makes it impossible for an attacker to steal any information from the installed extension or apps. Nevertheless it is recommended to change the *innerHTML* to *innerText* to prevent the injection of arbitrary HTML code.

PIO-01-017 API: Possible vault takeover via activation *(High)*

Since the initial activation process requires no authentication, it was determined possible for an attacker to take over the password vault of any user with as little knowledge as that of the email address paired with a user-click on the email. If the user clicks on the email, he or she grants the attacker access to the encrypted password vault. Although the attacker will not be able to decrypt the vault without the passphrase, it will become possible to attempt brute-forcing of the passphrase offline. What is more, simply deleting the password vault from the cloud server becomes a real risk through this vulnerability.

This issue can be trivially verified as follows:

Step 1: Trigger account activation

Command:
```
curl -k --request POST https://cloud-dev.padlock.io/auth/  --header "Accept:
application/vnd.padlock;version=1"  --header "Content-Type: application/x-www-
form-urlencoded"  --header "Host: cloud-dev.padlock.io"  --include --data
'email=abraham%40cure53.de'
```

Output:
```
HTTP/1.1 202 Accepted
[...]

{"email":"abraham@cure53.de","token":"330df7f2-4c5b-4afe-a364-fa3034d6f53c"}
```

Step 2: Wait for the user to click on the link

The user gets an email like the one depicted below. If the user clicks on the link, the attacker's device gains the ability to sync with the password vault of the victim-user:

Hi there!

You are receiving this email because you requested to a device with the Padlock Cloud account abraham@cure53.de. To complete the process, simply visit the link below:

http://cloud-dev.padlock.io/activate/?v=1&t=ab2e533d-4f55-444f-974d-35e4f66e7e9a

Best,
The Padlock Team

Step 3: Read or Delete the encrypted password vault for this user

At this stage, the attacker has:

- Read-access to the encrypted password vault, the attacker will receive the encrypted BLOB for this user and every time the BLOB is updated, the attacker's device will receive a new encrypted BLOB with the new updates.
- Ability to invoke any other API function authenticated as the user in question.

The attacker can attempt to:

- Brute-force the user's passphrase offline. These attempts are not time-constrained and can occur indefinitely.
- Delete or corrupt the password vault, for example using an attack like the one described in PIO-01-009.

It is recommended to modify the email sent to the user and adopt more precise wording about the consequences derived from clicking on the link provide. More specifically, this is the current version:

> *You are receiving this email because you requested to a device with the Padlock Cloud account abraham@cure53.de. To complete the process, simply visit the link below:*
>
> http://cloud-dev.padlock.io/activate/?v=1&t=ab2e533d-4f55-444f-974d-35e4f66e7e9a

It could be changed to the following shape and form for the new message:

> *You are receiving this email because you requested to a **[Device Information here]** device with the Padlock Cloud account abraham@cure53.de. To complete the process, simply visit the link below:*
>
> ***WARNING: This device will gain access to your password store! if you did not send this request, DO NOT click the link below.***
>
> http://cloud-dev.padlock.io/activate/?v=1&t=ab2e533d-4f55-444f-974d-35e4f66e7e9a

When the user clicks on that link, the web server currently shows a page similar to:

> You have successfully connected your device with
> the account **abraham@cure53.de**! You can now
> use the app to synchronize with **Padlock Cloud!**
>
> Close Window

Fig.: Successful Device Pairing message

This page should also be altered to:

1. Include an additional warning, so that the user realizes that this will give the new device the ability to sync and interact with the password vault;
2. Require another user's confirmation click. This could help reduce the chances of an accidental clicking on a sensitive action like this.

In addition to this, users should be given an administration panel from where they can see and modify the list of devices that are able to access the password vault at any point in time. This would let the users revoke access from a device that has been stolen or is no longer used.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals 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.

PIO-01-004 API: No error check if random fails *(Info)*

The code to generate a UUID() fails to check possible errors from *rand.Read(b)*[26]. If the system has a problem with generating random data, for example through a faulty system setup, the random bytes might end up to be just 0.

It is recommended to follow the official example[27], which includes a check to verify if an error was returned.

PIO-01-016 Missing Password Manager Best Practices *(Info)*

Password managers have been analyzed by security researchers over the past years and several papers on the topic have been published and discussed at various conferences[28] [29] [30]. It was found that Padlock is missing some features, like *autofill*,

---

[26] https://github.com/MaKleSoft/padlock-cloud/blob/master/app.go#L36
[27] https://golang.org/pkg/crypto/rand/#Read
[28] https://crypto.stanford.edu/~dabo/papers/pwdmgrBrowser.pdf
[29] https://www.cs.ox.ac.uk/files/6487/pwvault.pdf
[30] https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-li-zhiwei.pdf

which have constituted main attack surfaces for other password managers. This simplicity is clearly a strength in this regard. Nonetheless, there are minor things that should be addressed to improve user-protection.

Example 1: Trick a user into syncing with an attacker's device

In the current version of the Padlock, a malicious user can use the *Device name* field to inject additional sentences to the email message, thus having a chance to make it sound like a legitimate email for syncing the device. If the user clicks the authentication URL, the attacker gets a valid token to request the user's store. Once the store is acquired, the attacker can brute-force the master password locally.

In the development version the *Device name* got removed, though a user might still accidentally click on the link. This attack is described in more detail in PIO-01-017.

Example 2: Lock the password store after a certain inactivity threshold

Padlock will not lock and wipe the cleartext passwords from memory once unlocked. If a user forgets to lock the computer, a physical attacker can access the cleartext passwords, for example in an office environment or a hotel.

Following the example of other password managers it is recommended to automatically lock the device after a certain time of inactivity. In addition attempting to wipe the plaintext passwords through overwriting them would be a good idea as far as hindering other attacks is considered. A easily possible solution might be to simply use *window.close()* to exit the application process, thus wiping its memory. However, the actual behavior and whether this path is a reliable way for clearing the plaintext passwords has not been fully investigated.

Conclusion

As a foreword to conclusions, it needs to be underscored that developing bullet-proof and security-wise robust password manager and encrypted vault is an extremely difficult, if not impossible, task. This penetration test assignment, conducted by the Cure53 team against the Padlock.io application in April 2016, has demonstrated this point and highlighted some of the issues that need to be further considered in the development process of the application in question.

The tests have led to seventeen security problems being discovered. The vast majority of them, namely fifteen, have been classified as security vulnerabilities. The remaining two were determined to be just general weaknesses that the Padlock.io application's maintainers need to be aware of in future developments. Probably the most important and impressive conclusion is that no "Critical" issues have been found, which can overall be read as a positive result. This good outcome is also strengthened by the fact that all findings are seemingly not very hard to fix. It needs to be pointed out that the problems can be tied mostly to implementation flaws, since no serious design bugs were noted.

On a less positive note, there is still a claim to be made that while the API is presently a simple password vault store, the developers need to keep in mind that using a compound of software tools means that Padlock.io is exposed to a considerably large attack surface. Findings classified as "High" demonstrate some of the problems, as, for instance in PIO-01-017, an attacker is able to take over the password vault due to missing authentication and ability to make the user click on hostile links. Further, PIO-01-009 shows how a permanent DoS can be achieved through permanent sync stemming from a design flaw. Here the unsigned malicious users make changes that others blindly trust and become attack-victims. Again, handling trust relationships is difficult, yet must be at the core of the development. This means that previous research on the attacks against password managers absolutely needs to be consulted. As reviewed and referenced in PIO-01-016, security best practices known from earlier works can foster a development climate in which future alterations to the app's design do not cause vulnerability to attack vectors already described by security researchers. Another clear recommendation for Padlock.io is to work on improving the quality of user-communication, notifications and emails, as the current state of these exchanges leaves the tool vulnerable to numerous phishing attacks. More specifically, user warnings need to be deployed and issued whenever a critical transaction is about to take place. To just give two examples, better warnings policy needs to be applied to giving new devices access and capacity to perform a sync with the password vault, or a process of deleting the password store entirely. More focus should also be dedicated to possible Man-In-The-Middle issues, and mitigations against rogue applications plaguing the same phone that the Padlock.io is operating on, as this has led to tapjacking and screenshot app stealing user-data.

To conclude, the Padlock.io maintainers' attitude to positioning security at the center of the future development process will be crucial. Considering the already good state of security and the willingness to implement necessary changes and fixes, the Padlock.io is on its way to becoming a password manager that is as safe and secure as possible.