Penetration Test Report

Padloc

V 1.2
Amsterdam, August 14th, 2022
Public

Document Properties

Client	Padloc
Title	Penetration Test Report
Targets	Padloc App/PWA Padloc Server Padloc Browser Extension Padloc Tauri App Padloc Cordova App
Version	1.2
Pentesters	Tillmann Weidinger, Fabian Freyer
Authors	Tillmann Weidinger, Fabian Freyer, Stefan Vink, Tillmann Weidinger
Reviewed by	Stefan Vink
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	May 27th, 2022	Tillmann Weidinger, Fabian Freyer	Initial draft
0.2	May 31st, 2022	Stefan Vink	Reviewed
1.0	May 31st, 2022	Tillmann Weidinger, Fabian Freyer	Final version
1.1	August 10th, 2022	Tillmann Weidinger	Retest
1.2	August 14th, 2022	Stefan Vink	Reviewed Retest

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between March 23, 2022 and May 27, 2022, Radically Open Security B.V. carried out a penetration test for Padloc

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following target(s):

- Padloc App/PWA
- Padloc Server
- Padloc Browser Extension
- Padloc Tauri App
- Padloc Cordova App

The scoped services are broken down as follows:

- Auditing application component: 8 days
- Auditing Cordova component of Padloc: 2 days
- Auditing Tauri component of Padloc: 2 days
- Auditing Web extensions: 4-5 days
- Project management, coordination and review: 2 days
- Retest and fix verification before publication: 1-2 days
- **Total effort: 19 - 21 days**

1.3 Project objectives

ROS will perform a penetration test of the Padloc Mono repository with Padloc in order to assess the security of the scoped packages. To do so ROS will access the V4 branch of the Padloc repository (https://github.com/padloc/padloc/tree/v4) and guide Padloc in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The Security Audit took place between March 23, 2022 and May 27, 2022.

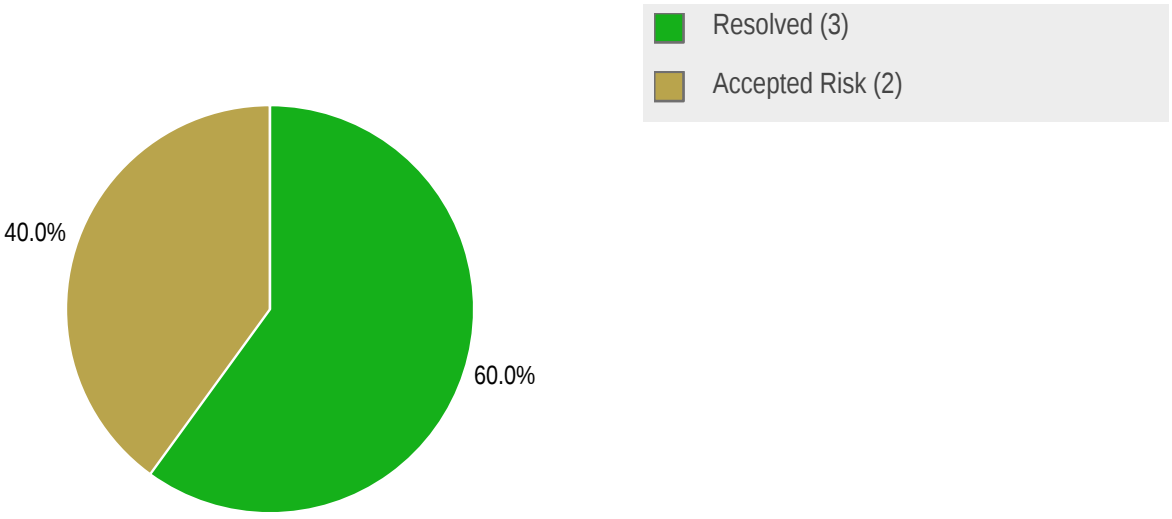
1.5 Results In A Nutshell

Input sanitization issues were found in the invitation email system PDL-008 (page 13) , which could be used to trick new and existing users to visit credential stealing phishing sites. Network adversaries would be able to intercept and modify database changes PDL-006 (page 16) and could also play a role in the tracking of Padloc users PDL-004 (page 15) .

The architectural design choices of Padloc limit the capabilities for wiping secrets in the device memory PDL-001 (page 11) . This is a general constraint of web applications.

An issue regarding a possible timing oracle for session tokens from a previous security audit was found to be improperly fixed PDL-007 (page 18).

1.5.1 Findings by Retest State

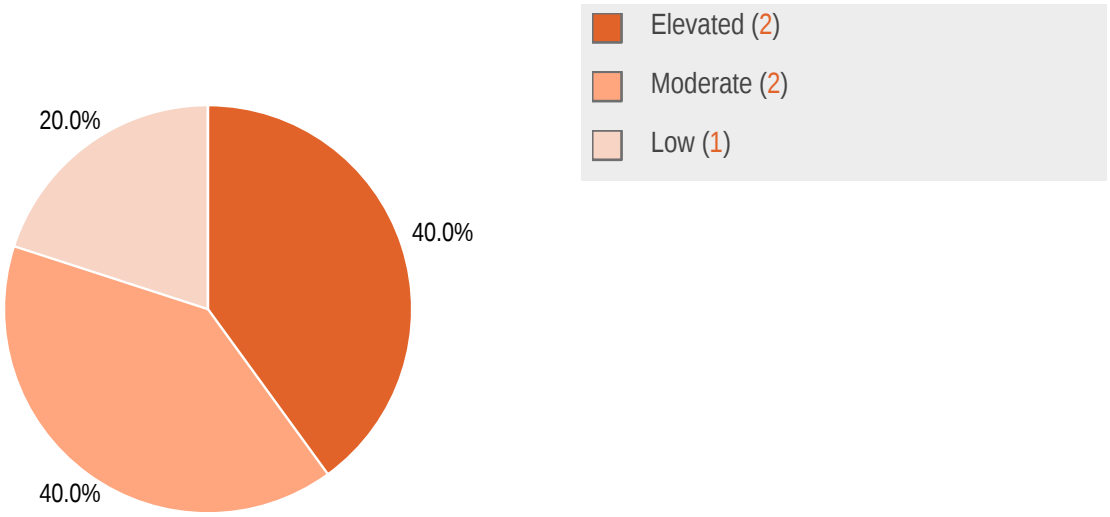


1.6 Summary of Findings

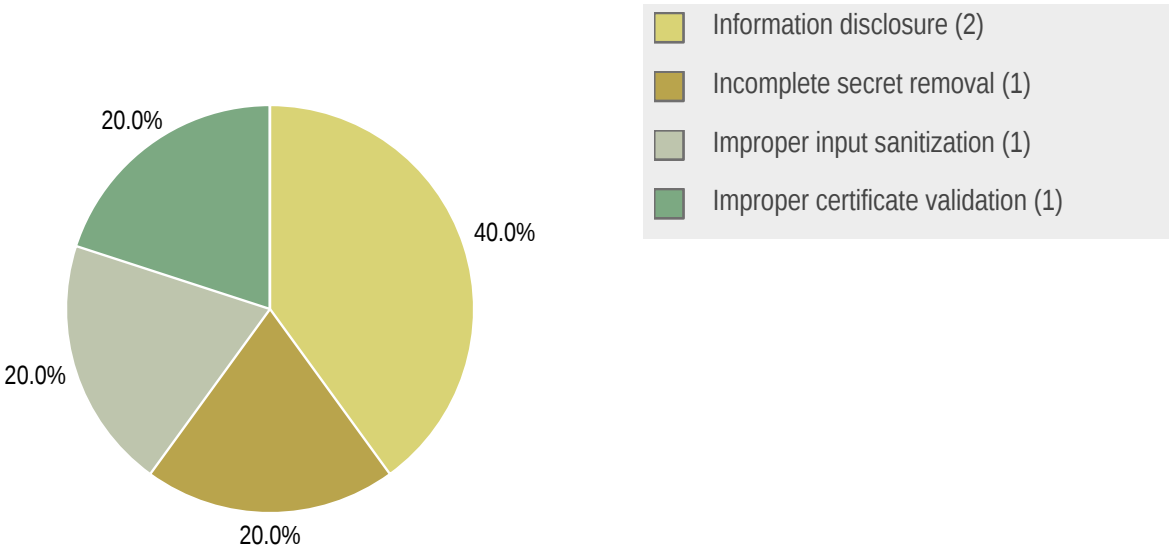
ID	Type	Description	Threat level
PDL-001	Incomplete Secret Removal	The password and master secret are kept in memory after locking the application.	Elevated

PDL-008	Improper Input Sanitization	It was found possible to inject arbitrary HTML into email templates sent for organization invites.	Elevated
PDL-004	Information Disclosure	The optional password audit feature has several privacy implications which requires proper documentation or code changes.	Moderate
PDL-006	Improper Certificate Validation	The PostgreSQL connection is not validating TLS certificates, allowing for Machine in the Middle attacks.	Moderate
PDL-007	Information Disclosure	The HMAC verification function is not implemented in constant time and leaks information through timing side channel.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
PDL-001	Incomplete Secret Removal	• Add documentation clearly stating the risk. • Investigate desktop application features for secure memory wipe and secret storage. • Consider removing this feature as it suggests false sense of security.
PDL-008	Improper Input Sanitization	• Sanitize username and organization before passing them into the email template. • Reduce available characters for usernames and organizations.
PDL-004	Information Disclosure	• Make this feature configurable. (has been implemented during the engagement). • Consider creating bloom filter based solutions. • Inform the user about the data transmission to Cloudflare. • Consider TLS pinning on the platforms that support it.
PDL-006	Improper Certificate Validation	• Set `rejectUnauthorized` to `true` .
PDL-007	Information Disclosure	• Implement a double HMAC based string comparison.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**
   We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.
2. **Enumeration**
   We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.
3. **Scanning**
   Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.
4. **Obtaining Access**
   We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately though provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2017) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Reconnaissance and Fingerprinting

We were able to gain information about the software and infrastructure through the following automated scans. Any relevant scan output will be referred to in the findings.

- Semgrep – https://semgrep.dev/
- Burp Suite Professional – https://portswigger.net/burp/pro

4 Findings

We have identified the following issues:

4.1 PDL-001 — After Locking The Application Secrets Retain in Memory

Vulnerability ID: PDL-001	**Status:** Accepted Risk
Vulnerability type: Incomplete Secret Removal	
Threat level: Elevated	

Description:

The password and master secret are kept in memory after locking the application.

Technical description:

The PWA and desktop applications allow locking the application. This is a commonly implemented functionality in password managers, which users expect to ensure that passwords are no longer accessible and wiped from memory.

Due to design choices (webview/browser), the application has no control over the retention of these secrets in the process memory.

A simple test involved taking memory dumps (`procdump`) of the browser or webview process and using `strings` to extract known secrets.

It was found that secrets were prevalent in these dumps, even after locking the application. After closing the browser tab it was unpredictable when these memory areas were overwritten/inaccessible, but in practice they were retained at least several seconds to minutes.

Reproduction:

1. Start the application or login into the website.
2. (Optionally) interact with secrets to check against.
3. Lock the application with the lock symbol.
4. (Optionally) Wait.
5. Create a process memory dump (`procdump -p 1337`).
6. Search for secret strings in the created dump `strings dump.dmp | grep "mysecret"`

For example, for a known password of `recipient patronage semester flagpole` :

```
strings ./chromium___enable_crashpad_time_2022-04-14_20\:46\:14.6103 | grep "recipient patronage
 semester flagpole"
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
recipient patronage semester flagpole
...
```

More sophisticated attacks do not require the above tooling and can be possibly executed from a malicious website or application opened in the same browser or system, or using memory side channels such (e.g. https://leaky.page/). Other scenarios would be shared devices (library, internet cafe, office, ...).

Further references, which could be found in the Webcrypto standard documentation:

```
This specification places no normative requirements on how implementations handle key material
once all references to it go away. That is, conforming user agents are not required to
zeroize key material, and it may still be accessible on device storage or device memory,
even after all references to the CryptoKey have gone away.
```

https://w3c.github.io/webcrypto/#security-developers

Furthermore, no protections are in place to prevent secrets from being written to swap under memory pressure. A threat model where this is is relevant is e.g. an evil maid attack -- a computer may be turned off but secrets may have been written to unencrypted swap space, allowing them to be extracted.

Impact:

A privileged adversary with remote or physical access to the device can extract secrets from memory dumps or leaks, even after the database was locked in the graphical interface. Although the attack complexity is relatively high, we believe that the threat level is elevated, due to the high impact of a compromised master secret. If unencrypted swap space is used, secrets may be written in plaintext to disk.

Recommendation:

- Add documentation clearly stating the risk.
- Investigate desktop application features for secure memory wipe and secret storage.
- Consider removing this feature as it suggests false sense of security.

Update :

As the locking mechanism will prevent casual attempts to extract passwords from a unlocked device, but not protect against advanced adversaries it was decided by Padloc to keep this feature and note down the issue as an accepted risk and properly document this in the user manual to highlight the risk.

4.2 PDL-008 — Email Template Injection

Vulnerability ID: PDL-008	**Status:** Resolved
Vulnerability type: Improper Input Sanitization	
Threat level: Elevated	

Description:

It was found possible to inject arbitrary HTML into email templates sent for organization invites.

Technical description:

The invite system for organizations can be abused to send manipulated email invites.

`packages/core/src/server.ts#L1086`

```
// Send invite link to invitees email address
                      await this.messenger.send(
                           invite.email,
                           new messageClass({
                               orgName: invite.org.name,
                               invitedBy: invite.invitedBy!.name || invite.invitedBy!.email,
                               acceptInviteUrl: `${this.config.clientUrl}${path}?
${params.toString()}`,
                           })
                       );
```

The send implementation is the following:

`packages/server/src/email/smtp.ts#L86`

```
async send<T extends MessageData>(email: string, message: Message<T>) {
        const { html, text } = this._getMessageContent(message);

        let opts = {
            from: this.config.from || this.config.user,
            to: email,
            subject: message.title,
            text,
            html,
        };
```

```
        return this._transporter.sendMail(opts);
```

The underlying templating system used in `_getMessageContent` replaces the template variables with the organization and username or email without sanitization or rejection of HTML content. It was also found that the organization name or username could contain several homoglyphs or invisible non-whitespace characters.

`packages/server/src/email/smtp.ts#L70`

```
private _getMessageContent<T extends MessageData>(message: Message<T>) {
        let html = this._templates.get(`${message.template}.html`);
        let text = this._templates.get(`${message.template}.txt`);

        if (!html || !text) {
            throw new Err(ErrorCode.SERVER_ERROR, `Template not found: ${message.template}`);
        }

        for (const [name, value] of Object.entries({ title: message.title, ...message.data })) {
            html = html.replace(new RegExp(`{{ ?${name} ?}}`, "gi"), value);
            text = text.replace(new RegExp(`{{ ?${name} ?}}`, "gi"), value);
        }

        return { html, text };
    }
```

The organization name length is not restricted but depending on the email client HTML content is visible in the subject of the email.

Impact:

An adversary could manipulate the email to trick the recipient into visiting an adversary controlled website, allowing for very convincing phishing attacks. Furthermore, it could impersonate another organization or sender. It would also be possible to perform denial of service attacks, where the email content is filled with spam links, triggering spam detection mechanisms. This would result in reputation degradation or loss for the sending party.

Recommendation:

- Sanitize username and organization before passing them into the email template.
- Reduce available characters for usernames and organizations.

Update :

The issue was fixed by Padloc. The length of organizations and users is now limited configurable and the message content and header is sanitized before passing it into the template. The available characterset for organizations and

users was not limited to specific 'safe' characters but should be considered in future iterations, even though exploitation is unlikely as it would require user interaction after a successful phishing attempt.

4.3 PDL-004 — Privacy Concerns for Password Audit

Vulnerability ID: PDL-004 **Status:** Accepted Risk

Vulnerability type: Information Disclosure

Threat level: Moderate

Description:

The optional password audit feature has several privacy implications which requires proper documentation or code changes.

Technical description:

The password audit feature was implemented to check the vault passwords for possible compromise. This is facilitated by the Haveibeenpwned range API. Padloc transmits truncated (5 characters) SHA-1 hashes of each password in the users vault to the Havibeenpwned API, which is provided by Cloudflare.

The implications are described here https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/ and here https://www.troyhunt.com/ive-just-launched-pwned-passwords-version-2/ . Please note the caveats section in the Cloudflare blogpost. These publications assume or imply a trust model where Cloudflare is a fully trusted party and TLS requests to Cloudflare are not intercepted (e.g. Corporate TLS interception proxy).

In the latter cases it is possible for the adversary to capture the truncated hashes.

While such a truncated hash may not seem to contain sensitive information about the original password, it will allow an adversary to:

- infer whether the password is known to be compromised and narrow it down to approximately 487 options.
- perform offline brute-force attacks to find matching non-leaked passwords, to use in a later dictionary based attack.
- determine re-use of passwords across multiple services.

Further information about the client are leaked to the API provider. This includes the IP-Address, location, browser fingerprint and browsing activity (checks are performed on every vault load).

Such (meta-)data can be used to identify the user across requests and services. These privacy leaks may be reduced by using a (corporate-owned) internal service or proxy, as suggested by Haveibeenpwned . However, end-users are unlikely to have access to such infrastructure and it would need to be considered in the trust model.

Transmitting password information to a trusted third party conflicts with the Padloc trust model, which requires end-to-end encryption of passwords and aims to avoid the trust of third parties.

A possible solution to this problem could be a set of bloom filters with a small file size, so it can be downloaded by the user. This could be used to check for possible compromise locally and then (optionally) validate likely compromised passwords over the network.

Impact:

The compromised passwords checks produce (meta-)data which can be used to infer leaked passwords, reduce password brute-force complexity, identify and track the user across requests and services. It breaks the trust model of Padloc.

Recommendation:

- Make this feature configurable. (has been implemented during the engagement).
- Consider creating bloom filter based solutions.
- Inform the user about the data transmission to Cloudflare.
- Consider TLS pinning on the platforms that support it.

Update :

It was decided by Padloc to keep this feature, make it fine grained configurable and document the feature limitation. Therefore we see this issue as an accepted risk which, depending on the individual threat model, can be completely avoided by disabling the feature.

4.4 PDL-006 — PostgreSQL Certificate Validation Disabled

Vulnerability ID: PDL-006	**Status:** Resolved
Vulnerability type: Improper Certificate Validation	
Threat level: Moderate	

Description:

The PostgreSQL connection is not validating TLS certificates, allowing for Machine in the Middle attacks.

Technical description:

The PostgreSQL connection configuration disables TLS certificates checks by setting `rejectUnauthorized` to `false`.

`server/src/storage/postgres.ts#L37`

```
constructor(public config: PostgresConfig) {
        const { host, user, password, port, database, tls, tlsCAFile } = config;
        const tlsCAFilePath = tlsCAFile && resolve(process.cwd(), tlsCAFile);
        const ca = tlsCAFilePath && readFileSync(tlsCAFilePath).toString();
        this._pool = new Pool({
            host,
            user,
            password,
            port,
            database,
            ssl: tls
                ? {
                      rejectUnauthorized: false,
                      ca,
                  }
                : undefined,
        });
    }
```

The same configuration was also found inside the `mongo2postgres` ts file (`packages/server/src/tools/mongo2postgres.ts#L22`).

Impact:

An adversary with control over the network between the server and database instance can intercept and modify requests due to broken certificate validation.

Recommendation:

- Set `rejectUnauthorized` to `true` .

Update :

The issue was fixed by Padloc. The database connection now does enforce TLS certificate checks.

4.5 PDL-007 — HMAC Verification is not Constant Time

Vulnerability ID: PDL-007 **Status:** Resolved

Vulnerability type: Information Disclosure

Threat level: Low

Description:

The HMAC verification function is not implemented in constant time and leaks information through timing side channel.

Technical description:

The function `equalCT` is used to compare session token and is not constant time, as advertised by the function name.

`packages/core/src/encoding.ts#L507`

```
export function equalCT<T extends ArrayLike<any>>(a: T, b: T): boolean {
    let match = true;

    for (let i = 0; i < a.length; i++) {
        match = match && a[i] === b[i];
    }

    return a.length === b.length && match;
}
```

Impact:

An adversary could facilitate the timing oracle provided by the session token comparison to leak a valid session token. As the comparison time difference between a valid and invalid character is relatively short, an adversary would need millions of requests. It is less likely to stay undetected by logging or rate limiting and has a very small chance of success, therefore a low threat level.

Recommendation:

- Implement a double HMAC based string comparison.

Update :

This issue was already fixed by Padloc during the engagement, by implementing a double HMAC scheme for string comparison.

5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

5.1 NF-002 — HTML and Markdown Content is Properly Sanitized

The application code properly sanitizes user controlled HTML via `dompurify.sanitize()` . Markdown code is first converted to HTML and then also sanitized. It is possible to disable this feature and introduce XSS but the global flag is enabled per default.

`packages/app/src/lib/markdown.ts#L14`

```
export function mardownToHtml(md: string, san = true) {
    let markup = marked(md, {
        headerIds: false,
    });
    if (san) {
        markup = sanitize(markup);
    }
    return html`${unsafeHTML(markup)}`;
}
```

`packages/app/src/elements/rich-content.ts#L99`

```
render() {
        switch (this.type) {
            case "markdown":
                return mardownToHtml(this.content, this.sanitize);
            case "html":
                const content = this.sanitize
                    ? sanitize(this.content, { ADD_TAGS: ["pl-icon"], ADD_ATTR: ["icon"] })
                    : this.content;
                return html`${unsafeHTML(content)}`;
            default:
                return html`${this.content}`;
        }
    }
```

6 Future Work

- **Regular security assessments**
  Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

7 Conclusion

We discovered 2 Elevated, 2 Moderate and 1 Low -severity issues during this penetration test.

The communication between ROS and Padloc was excellent, which resulted in discussions in the chat environment of ROS and fixes from Padloc before the conclusion of the audit.

No high or critical issues were found and the amount of elevated, moderate and low severity findings indicate a robust code base at the implementation level.

The `server` , `app` , `pwa` , `cordova` , `extension` , `core` and `tauri` packages were audited and were found to implement solid cryptographic primitives and hardened rendering libraries. The code seem to be well written and no major implementation issues were found.

However, architecturally, there are fundamental challenges Padloc still faces in order to achieve the design goals of a zero-trust and secure password storage. These challenges are inherent to the chosen technologies: As a web-based password manager, users have to trust the served JavaScript code. This contradicts the claim in the documentation that "unlike other products, Padloc does not require explicit trust between the end user and the host" . Furthermore, the chosen web technologies disallow explicit memory management and do not provide ways of securely clearing memory and disallowing it from being written to disk.

This is a known issue with other projects with similar design constraints, where end-to-end encryption is used in the context of a web browser. A common example is Protonmail: https://eprint.iacr.org/2018/1121 Section 4.1.1 and Section 1 of the company response https://proton.me/news/cryptographic-architecture-response .

As these issues are commonly accepted risks, we recommend qualifying these in the documentation and adjusting the claims of "no explicit trust" accordingly or investigate moving to other, native technologies.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Retest Notes:
The filed issues were all marked as resolved. Either by fixing the issue completely, or by accepting and documenting certain risks. Further improvements were made by hardening the CSP and implementing a helper tool, which publishes hashsums of all files generated during releases. This tool and instructions how to verify server side files can help to assert if server side assets or code were changed beyond the official release. This tool has certain limitation, as it will not protect against targeted attacks and is not working with local builds, but it is a good step forward to automated compromise detection and therefore a useful addition to the padloc ecosystem. We further encourage Padloc to engage

in research for further hardening against advanced adversaries, but the current implementation seems like a reasonable secure implementation and we can recommend usage in productive environments.

Appendix 1 Testing team

Tillmann Weidinger	Tillmann Weidinger is a trained full-stack developer with a strong emphasis on security. He started tinkering with computers in his early teens. Due to this he has multiple years of experience in (reverse-)engineering hard- and software, software architecture and breaking things. His main interests evolve around Secure Coding, Automation, Web Applications, WiFi, DMA attacks and other topics between hard- and software with a focus on red-teaming. He enjoys programming in multiple languages and recently has chosen rust as his new favorite. He started studying IT-Security at Ruhr University Bochum and switched to Computer Science at FH Bochum and will graduate in 2022. Due to his broad experience in application development and system's security he can quickly adapt to new IT-Security related topics and is always happy to learn lesser known facts.
Fabian Freyer	After winning multiple high-profile international CTF tournaments and qualifying for events such as DEF CON CTF and 0CTF with his CTF team Tasteless, Fabian is now focusing on code auditing and pentesting. Among his public work, he has identified critical vulnerabilities in iTerm2 (e.g. CVE-2019-9535 together with Stefan Grönke), Homebrew and RocketChat. While he prefers reviewing low-level and native code in Rust, C and Assembly and currently focuses on Apple software ecosystems such as iOS and macOS, Fabian has a well-founded understanding of the security pitfalls of web-frontend desktop applications and has identified a number of security flaws in Electron-based applications (e.g. https://hackerone.com/reports/899964). Recently, Fabian has been part of a collaborative effort to investigate the security of the Apple AirTags and has held a [talk](https://hardwear.io/netherlands-2021/speakers/jiska-and-fabian-and-stacksmashing.php) on this topic at a hardware security conference.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.