Penetration Test Report

Rauthy maintainers

V 1.0
Amsterdam, September 15th, 2025
Confidential

Document Properties

Client	Rauthy maintainers
Title	Penetration Test Report
Target	• Rauthy identity provider
Version	1.0
Pentesters	Frank Plattel, Morgan Hill
Authors	Morgan Hill, Frank Plattel, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	August 22nd, 2025	Morgan Hill, Frank Plattel	Initial draft
0.2	August 25th, 2025	Marcus Bointon	Review
1.0	September 15th, 2025	Marcus Bointon	1.0

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

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:

- Rauthy identity provider

1.3 Project objectives

ROS will perform a code audit of the Rauthy identity provider with Rauthy in order to assess its security. To do so ROS will access the source code and guide Rauthy 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 July 14, 2025 and August 22, 2025.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 1 Elevated, 3 Low and 3 N/A-severity issues.

This project involved a crystal-box analysis of the Rauthy identity provider, developed in Rust. We identified several findings of non-applicable to low severity, and a single elevated-severity finding.

Some of the findings affect logic bugs or timing related issues in the code, such as a non-constant time string comparison, which may allow for timing attacks to learn the client secret RAUTHY-005 (page 11). Another wrapped around a divide-by-zero problem, which may have led to unexpected behavior in the client RAUTHY-004 (page 17).

We identified a denial-of-service vulnerability in a home-grown dependency of this project, which provides database functions. RAUTHY-006 (page 13) explains this issue that could have caused an uncaught exception, which may result in a crash.

Other findings only target the development environment, such as RAUTHY-008 (page 18), which shows that ports are opened on all interfaces during a default development setup.

We found a static certificate authority chain and private in the repository which are shipped by default RAUTHY-009 (page 14).

In RAUTHY-007 (page 10) we showed that it was possible to inject a JavaScript payload into a profile image, which could have been abused in combination with user-interaction to hijack the web frontend of the application.

A known vulnerability in a dependency is still present in the latest version, but it's not exploitable in this app's context RAUTHY-001 (page 15).

1.6 Summary of Findings

Info	Description
RAUTHY-007 **Elevated** **Type:** Cross-site scripting **Status:** resolved	It was possible to inject a JavaScript payload into an SVG image which was persistently stored under the profile image of a user.
RAUTHY-005 **Low** **Type:** Timing oracle **Status:** resolved	Client secrets are compared with a non-constant time string comparison, theoretically enabling timing attacks to learn the client secret.
RAUTHY-006 **Low** **Type:** Reachable unwrap **Status:** resolved	It is possible to trigger a panic inside the hiqlite library by making HTTP requests to rauthy.
RAUTHY-009 **Low** **Type:** Exposed credentials **Status:** resolved	The repository contains a CA-chain with a private key.
RAUTHY-001 **N/A** **Type:** Vulnerable Dependency **Status:** not_retested	rauthy uses the latest version of the rsa crate, but it is vulnerable to CVE-2023-49092.
RAUTHY-004 **N/A** **Type:** Logic bug **Status:** resolved	Setting the page_size parameter to 0, on get users and get sessions endpoints, results in unexpected behavior.

RAUTHY-008 **N/A** **Type:** Insecure configuration **Status:** not_retested	The default configuration of the development environment requires the use of Docker containers which open ports on all interfaces.

1.7 Summary of Recommendations

Info	Recommendation
RAUTHY-007 **Elevated** **Type:** Cross-site scripting **Status:** resolved	• Perform input validation and sanitization on all untrusted input.
RAUTHY-005 **Low** **Type:** Timing oracle **Status:** resolved	• Switch to a constant-time string comparison for secrets.
RAUTHY-006 **Low** **Type:** Reachable unwrap **Status:** resolved	• Return an error rather than unwrapping.
RAUTHY-009 **Low** **Type:** Exposed credentials **Status:** resolved	• Remove the certificate and key from the repo. • Consider generating a new certificate during the setup process.
RAUTHY-001 **N/A** **Type:** Vulnerable Dependency **Status:** not_retested	• No immediate action is required. Keep dependencies up-to-date as usual.
RAUTHY-004 **N/A** **Type:** Logic bug **Status:** resolved	• Ensure `page_size` is at least `1` and consider imposing a lower page size limit than `u16::MAX`.
RAUTHY-008 **N/A** **Type:** Insecure configuration **Status:** not_retested	• Consider modifying the default listen interface to the loopback interface (`127.0.0.1`).

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 (2021) 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

Through automated scans we were able to gain the following information about the software and infrastructure. Detailed scan output can be found in the sections below.

3.1 Automated Scans

As part of our active reconnaissance we used the following automated scans:

- Burp Suite – https://portswigger.net/burp/pro

4 Pentest Technical Summary

4.1 Findings

We have identified the following issues:

4.1.1 RAUTHY-007 — Persistent cross-site scripting in profile picture

Vulnerability ID: RAUTHY-007	**Status:** Resolved
Vulnerability type: Cross-site scripting	
Threat level: Elevated	

Description:

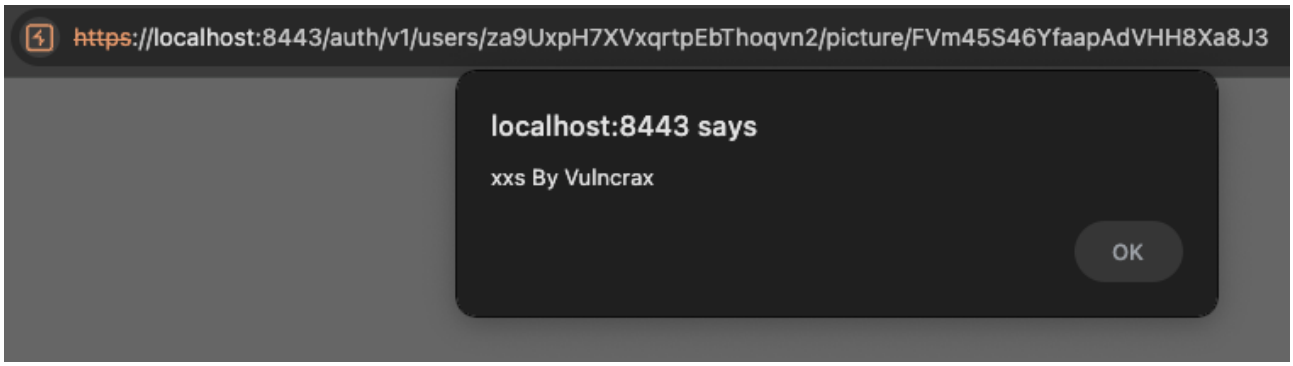
It was possible to inject a JavaScript payload into an SVG image which was persistently stored under the profile image of a user.

Technical description:

By injecting a JavaScript payload into an SVG image, such as here: https://github.com/vulncrax/svg-xss/blob/main/alert.svg, it was possible to make the JavaScript code persistent, accessible and executable by other users when they try to load the image.

This vulnerability was present in the following endpoint: `PUT /auth/v1/users/<userid>/picture`

The payload was successfully executed when directly browsing to the image:



The following code should have prevented this, but it handles the input validation incorrectly: https://github.com/sebadob/rauthy/blob/a0143e9e1babe68c24b75c12316d91d9a9e59277/src/data/src/entity/pictures.rs#L183 https://github.com/sebadob/rauthy/blob/a0143e9e1babe68c24b75c12316d91d9a9e59277/src/data/src/entity/logos.rs#L421

Impact:

An adversary could have exploited this issue and tricked another user or admin to browse to their profile picture directly, which may have allowed the adversary to take over full control over the frontend of the application.

Recommendation:

- Perform input validation and sanitization on all untrusted input.

4.1.2 RAUTHY-005 — Device endpoint non-constant time client_secret comparison

Vulnerability ID: RAUTHY-005	**Status:** Resolved
Vulnerability type: Timing oracle	
Threat level: Low	

Description:

Client secrets are compared with a non-constant time string comparison, theoretically enabling timing attacks to learn the client secret.

Technical description:

The standard Rust string comparison performed by the `==` operator exits early, providing a timing oracle that reveals how much of the string has matched. This becomes security-relevant when comparing secrets as done here: https://github.com/sebadob/rauthy/blob/48f36282ae08b5244bed87260b3d9e941687f5ca/src/api/src/oidc.rs#L552.

In practice, string comparison in Rust on a modern system takes tens of nanoseconds, and picosecond-level measurement precision would be required to cary out the attack. Even minimal jitter makes exploiting this oracle impractical.

Impact:

On modern systems this issue is more theoretical than practical. There is the theoretical possibility that the secret can be learned if an attacker has a sufficiently accurate method of measuring the timing.

Recommendation:

- Switch to a constant-time string comparison for secrets.

Update 2025-08-15 13:35:

https://github.com/sebadob/rauthy/pull/1166 should fix this.

Update 2025-08-15 16:35:

Commentary from the developer:

I did a lot of tests now, with my own impl, with `simd` comparisons, with external crates that are derived from the Linux kernels impl, and so on. In the end, when I just do a simple `left == right` in Rust, it is so fast, that it's even impossible to measure any difference on the same host even only in memory. I compared an empty string, 64 char string and a 16640 chars string, each version with a perfect match, a fail on the first character and a fail on the last one. All Strings, even the 16640 character long string were finished in the "same" (unmeasurable) time as the others.

So now my question, is this even something to worry about? Even when we were sending on the same host via a UDS, the latency would be multiple orders of magnitude higher than the time it takes to compare 2 simple strings.

Edit:

Measurements were between 10 and 30ns for all of these cases, no matter if empty strings or 16640 chars. These timings were changing all the time, in a way that it was not 30ns for 16640 chars, but it would appear at any position. Most of the time it was 20ns for all of them and I think it's even too fast to measure properly inside highly optimized Rust `release` code.

Update 2025-08-15 17:14:

From the developer:

I pushed an improved version of the constant time comparison:

https://github.com/sebadob/rauthy/pull/1167

This now also includes the comparison for `PamHost`s in constant time.

4.1.3 RAUTHY-006 — hiqlite unwrap in kv handler

Vulnerability ID: RAUTHY-006 **Status:** Resolved

Vulnerability type: Reachable unwrap

Threat level: Low

Description:

It is possible to trigger a panic inside the hiqlite library by making HTTP requests to rauthy.

Technical description:

While running Burp suite against rauthy we hit an unwrap in hiqlite, resulting in a panic.

The bug is not triggered by a single request but rather a combination of requests. This feels like a race condition that can be reproduced by running a Burp scan against rauthy.

```
2025-08-15T11:20:21.871499Z DEBUG handle{self=Index}: rauthy_data::html: encoding="br"
2025-08-15T11:20:21.871543Z DEBUG handle{self=Index}: rauthy_data::language: common_languages=["en-
US", "en"]
2025-08-15T11:20:21.871560Z DEBUG handle{self=Index}: rauthy_data::html: language=En

thread 'tokio-runtime-worker' panicked at /home/test/.cargo/registry/src/
index.crates.io-1949cf8c6b5b557f/hiqlite-0.10.0/src/store/state_machine/memory/kv_handler.rs:66:87:
called `Result::unwrap()` on an `Err` value: None
```

```
stack backtrace:
   0: __rustc::rust_begin_unwind
             at /rustc/29483883eed69d5fb4db01964cdf2af4d86e9cb2/library/std/src/panicking.rs:697:5
   1: core::panicking::panic_fmt
             at /rustc/29483883eed69d5fb4db01964cdf2af4d86e9cb2/library/core/src/panicking.rs:75:14
   2: core::result::unwrap_failed
             at /rustc/29483883eed69d5fb4db01964cdf2af4d86e9cb2/library/core/src/result.rs:1761:5
   3: core::result::Result<T,E>::unwrap
             at /rustc/29483883eed69d5fb4db01964cdf2af4d86e9cb2/library/core/src/result.rs:1167:23
   4: hiqlite::store::state_machine::memory::kv_handler::kv_handler::{{closure}}::{{closure}}
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/hiqlite-0.10.0/src/
store/state_machine/memory/kv_handler.rs:66:87
   5: hiqlite::store::state_machine::memory::kv_handler::kv_handler::{{closure}}
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/hiqlite-0.10.0/src/
store/state_machine/memory/kv_handler.rs:52:1
   6: tokio::runtime::task::core::Core<T,S>::poll::{{closure}}
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/tokio-1.47.1/src/
runtime/task/core.rs:365:24
   7: tokio::loom::std::unsafe_cell::UnsafeCell<T>::with_mut
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/tokio-1.47.1/src/
loom/std/unsafe_cell.rs:16:9
   8: tokio::runtime::task::core::Core<T,S>::poll
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/tokio-1.47.1/src/
runtime/task/core.rs:354:30
   9: tokio::runtime::task::harness::poll_future::{{closure}}
```

```
             at /home/test/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/tokio-1.47.1/src/
runtime/task/harness.rs:535:30
```

Impact:

Denial of service.

Recommendation:

- Return an error rather than unwrapping.

Update 2025-08-15 16:53:

Commentary from the developer:

That `unwrap()` is in a location where it should never `panic` because it means that the receiver for a Cache GET is gone, while they should always wait until they get some result back.

Update 2025-08-15 17:27:

Fixed in https://github.com/sebadob/hiqlite/pull/243

4.1.4 RAUTHY-009 — Static CA and private key in source code

Vulnerability ID: RAUTHY-009 **Status:** Resolved

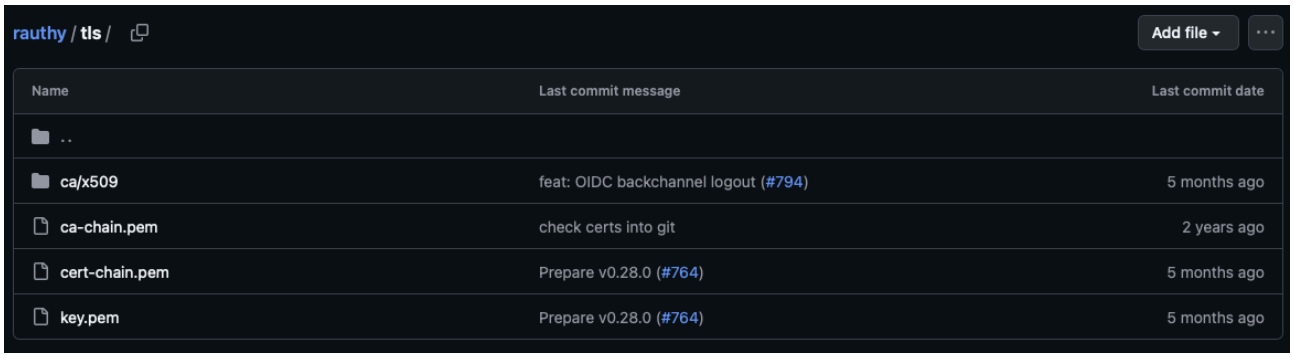
Vulnerability type: Exposed credentials

Threat level: Low

Description:

The repository contains a CA-chain with a private key.

Technical description:



Impact:

A deployment may use this certificate, which may allow adversaries to successfully perform a machine-in-the-middle attack, as the private key is public.

Recommendation:

- Remove the certificate and key from the repo.
- Consider generating a new certificate during the setup process.

4.1.5 RAUTHY-001 — Upstream weakness in rsa crate

Vulnerability ID: RAUTHY-001 **Status:** Not Retested

Vulnerability type: Vulnerable Dependency

Threat level: N/A

Description:

rauthy uses the latest version of the rsa crate, but it is vulnerable to CVE-2023-49092.

Technical description:

Cargo audit shows the following information:

```
Crate:     rsa
Version:   0.9.8
```

```
Title:     Marvin Attack: potential key recovery through timing sidechannels
Date:      2023-11-22
ID:        RUSTSEC-2023-0071
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0071
Severity:  5.9 (medium)
Solution:  No fixed upgrade is available!
Dependency tree:
rsa 0.9.8
├── rauthy-error 0.32.0
│   ├── rauthy-service 0.32.0
│   │   ├── rauthy-schedulers 0.32.0
│   │   │   └── rauthy 0.32.0
│   │   ├── rauthy-handlers 0.32.0
│   │   │   └── rauthy 0.32.0
│   │   └── rauthy 0.32.0
│   ├── rauthy-schedulers 0.32.0
│   ├── rauthy-notify 0.32.0
│   │   └── rauthy-data 0.32.0
│   │       ├── rauthy-service 0.32.0
│   │       ├── rauthy-schedulers 0.32.0
│   │       ├── rauthy-middlewares 0.32.0
│   │       │   └── rauthy 0.32.0
│   │       ├── rauthy-jwt 0.32.0
│   │       │   ├── rauthy-service 0.32.0
│   │       │   ├── rauthy-handlers 0.32.0
│   │       │   └── rauthy 0.32.0
│   │       ├── rauthy-handlers 0.32.0
│   │       └── rauthy 0.32.0
│   ├── rauthy-middlewares 0.32.0
│   ├── rauthy-jwt 0.32.0
│   ├── rauthy-handlers 0.32.0
│   ├── rauthy-data 0.32.0
│   ├── rauthy-common 0.32.0
│   │   ├── rauthy-service 0.32.0
│   │   ├── rauthy-schedulers 0.32.0
│   │   ├── rauthy-notify 0.32.0
│   │   ├── rauthy-middlewares 0.32.0
│   │   ├── rauthy-jwt 0.32.0
│   │   ├── rauthy-handlers 0.32.0
│   │   ├── rauthy-data 0.32.0
│   │   ├── rauthy-api-types 0.32.0
│   │   │   ├── rauthy-service 0.32.0
│   │   │   ├── rauthy-jwt 0.32.0
│   │   │   ├── rauthy-handlers 0.32.0
│   │   │   ├── rauthy-data 0.32.0
│   │   │   └── rauthy 0.32.0
│   │   └── rauthy 0.32.0
│   ├── rauthy-api-types 0.32.0
│   └── rauthy 0.32.0
└── rauthy-data 0.32.0
```

A fix has been worked on upstream: https://github.com/RustCrypto/RSA/issues/390.

Impact:

rauthy uses the rsa module only for JWT signatures. We do not believe that the timing of the verification is observable via the network therefore the Maven attack should not be possible in this case.

Recommendation:

- No immediate action is required. Keep dependencies up-to-date as usual.

Update 2025-08-15 12:36:

An issue has been opened to track the upstream issue: https://github.com/sebadob/rauthy/issues/197.

4.1.6 RAUTHY-004 — Get sessions/users divide by zero page size

Vulnerability ID: RAUTHY-004	**Status:** Resolved
Vulnerability type: Logic bug	
Threat level: N/A	

Description:

Setting the `page_size` parameter to `0`, on get users and get sessions endpoints, results in unexpected behavior.

Technical description:

The get users and get sessions endpoints both implement pagination in the same way. As part of the implementation they accept a `page_size` parameter intended to control the number of items returned in a page.

The `page_size` parameter is accepted and validated as an unsigned 16-bit integer. The `x_page_count` is then calculated using the `page_size` as the divisor. For this division both operands are cast to 64-bit floats this is the key detail of the finding.

With an integer division when `page_size` is zero we would expect a division by zero panic. However, division by zero is defined for IEEE 754 floating points as infinity we therefore do not see a panic. Instead `inf` is returned from the division which we then take the ceiling of and cast to a 32-bit unsigned integer. This results in `x_page_count` being `u32::MAX`.

`x_page_count` is returned to the client in the `x-page-count` HTTP header. This header is used by the client to determine how many pages of results there are to obtain from the server.

These divisions can be found here:

https://github.com/sebadob/rauthy/blob/48f36282ae08b5244bed87260b3d9e941687f5ca/src/api/src/sessions.rs#L66

https://github.com/sebadob/rauthy/blob/48f36282ae08b5244bed87260b3d9e941687f5ca/src/api/src/users.rs#L110

Impact:

Clients may not anticipate this behavior which may lead to further unexpected behavior in the client.

Recommendation:

- Ensure `page_size` is at least `1` and consider imposing a lower page size limit than `u16::MAX`.

Update 2025-08-15 12:36:

Fixed in https://github.com/sebadob/rauthy/pull/1165.

4.1.7 RAUTHY-008 — Dev ports listening on all interfaces

Vulnerability ID: RAUTHY-008 **Status:** Not Retested

Vulnerability type: Insecure configuration

Threat level: N/A

Description:

The default configuration of the development environment requires the use of Docker containers which open ports on all interfaces.

Technical description:

The following ports are opened in the development environment by default configuration of the docker containers:

```
tcp            LISTEN          0               4096                          0.0.0.0:1025
               0.0.0.0:*            users:(("docker-proxy",pid=1165,fd=7))
tcp            LISTEN          0               4096                          0.0.0.0:1080
               0.0.0.0:*            users:(("docker-proxy",pid=1179,fd=7))
tcp            LISTEN          0               4096                          0.0.0.0:5432
               0.0.0.0:*            users:(("docker-proxy",pid=1207,fd=7))
tcp            LISTEN          0               1024                          0.0.0.0:8080
               0.0.0.0:*            users:(("rauthy",pid=1839,fd=28))
```

```
tcp            LISTEN          0               1024                            0.0.0.0:8100
               0.0.0.0:*            users:(("rauthy",pid=1839,fd=22))
tcp            LISTEN          0               1024                            0.0.0.0:8200
               0.0.0.0:*            users:(("rauthy",pid=1839,fd=23))
```

Impact:

When a developer is setting up their development environment, these ports may be opened with the default configuration. This expands the attack surface of the development server, which may be unnecessary in some cases.

Recommendation:

- Consider modifying the default listen interface to the loopback interface (`127.0.0.1`).

4.2 Non-Findings

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

4.2.1 NF-002 — Unsafe code

rauthy explicitly disables unsafe code at the module level. The only unsafe blocks are for reading environment variables; it is good to see such minimal use of unsafe.

4.2.2 NF-003 — Downcasting

There are a handful of locations where downcasts between integer types are made. In Rust the behavior of these casts is well defined (unlike in C). For security purposes we are therefore interested in whether the well-defined behavior creates any exploitable bugs in the application logic. The casts are made in the context of timestamps. We have established that the timestamps come from the system and are therefore not under the control of an attacker, so it is not possible to abuse these downcasts remotely. An attacker would require control of the system clock, and we think it is reasonable to trust the system clock in applicable threat models.

4.2.3 NF-011 — Default Kubernetes configuration is sane

The default configuration for the Kubernetes deployment is secure. While there are no Kubernetes object specifications in the repository, they can be referenced from the documentation, which provides clear and detailed guidance. All configuration options in the source code, including the default user in the container, are specified securely.

5 Future Work

- **Additional component audits**
  Since they were not directly part of the initially scoped services, the home-grown Hiqlite dependency and the PAM module should receive targeted security audits to confirm secure integration.
- **Retest of findings**
  When mitigations for the vulnerabilities described in this report have been deployed, perform a repeat test to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

6 Conclusion

We discovered 1 Elevated, 3 Low and 3 N/A-severity issues during this penetration test.

The pentest revealed several non-applicable or low-severity issues, along with a single elevated-severity risk. However, it was clear that this project had security in mind from the start, and generally does a very good job in that regard. The project is active and well-managed; most of the vulnerabilities we reported were addressed within hours. In other instances, the findings were categorized as accepted risks or were already scheduled for resolution in future releases. The team's swift response demonstrates a strong commitment to security and a proactive approach to risk management. Overall, this reflects a robust security posture and a dedication to maintaining the integrity of the project.

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 that must be continuously evaluated and improved – this penetration test is just a one-time snapshot. 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.

Appendix 1 Testing Team

Frank Plattel	Frank is a pentester and developer. With a background in programming and a passion for hacking since childhood, he transitioned from software development to security. He specializes in red-team assessments, identity and access management and application security.
Morgan Hill	Morgan is a seasoned security consultant with a strong background in DevOps and IoT. He played a pivotal role in designing and implementing significant portions of Holoplot's professional audio products, which are prominently used at the MSG Sphere in Las Vegas. His expertise in media security was showcased when he presented at the MCH2022 event. Morgan's exceptional performance in the field is further demonstrated by his position on the SANS advisory board, where he achieved a high score and emerged victorious in the CTF in SEC-488. Aside from his IT accomplishments, Morgan has also made substantial contributions to the rail sector. He successfully orchestrated the delivery of new signaling schemes and station remodeling, granting him a unique perspective on operational technology. Currently, he is committed to utilizing his skills and innovative mindset to elevate the security landscape.
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.