Radically Open Security

Penetration Test Report

GNU Taler

V 1.0
Amsterdam, November 29th, 2024
Public

Document Properties

Client	GNU Taler
Title	Penetration Test Report
Targets	EBICS ISO20022 interfaces Web banking interface
Version	1.0
Pentester	András Veres-Szentkirályi
Authors	András Veres-Szentkirályi, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	November 5th, 2024	András Veres-Szentkirályi	Initial draft
0.2	November 6th, 2024	Marcus Bointon	Review
1.0	November 29th, 2024	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

Between October 1, 2024 and October 11, 2024, Radically Open Security B.V. carried out a penetration test for GNU Taler.

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 targets:

- EBICS ISO20022 interfaces
- Web banking interface

The scoped services are broken down as follows:

- Pentest (incl. reporting): 6 days
- **Total effort: 6 days**

1.3 Project objectives

ROS will perform a penetration test of the libeufin component of GNU Taler, with a focus on regional currency setups, in order to assess the security this component. To do so ROS will access the targets and guide the GNU Taler developer community 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 October 1, 2024 and October 11, 2024.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 2 High, 3 Moderate and 4 Low-severity issues.

The findings affect both the EBICS ISO20022 interfaces and the web banking interface. The good news is that none of these are issues that could be easily exploited by unauthenticated attackers randomly scanning the internet. However,

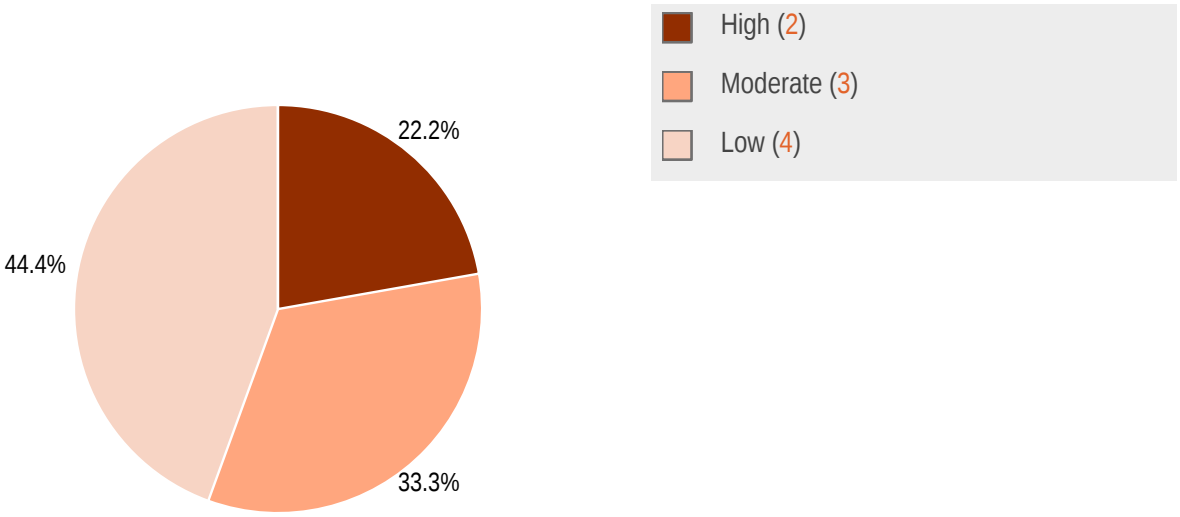
more prepared attackers or those with access to banks could guess the admin password, read files on the server, launch attacks against other servers via SSRF, or tamper with signed transactions.

Less severe bugs can make it possible to guess user passwords, abort withdrawals belonging to other users, verify whether a user account with a given name exists, or abuse revoked certificates.

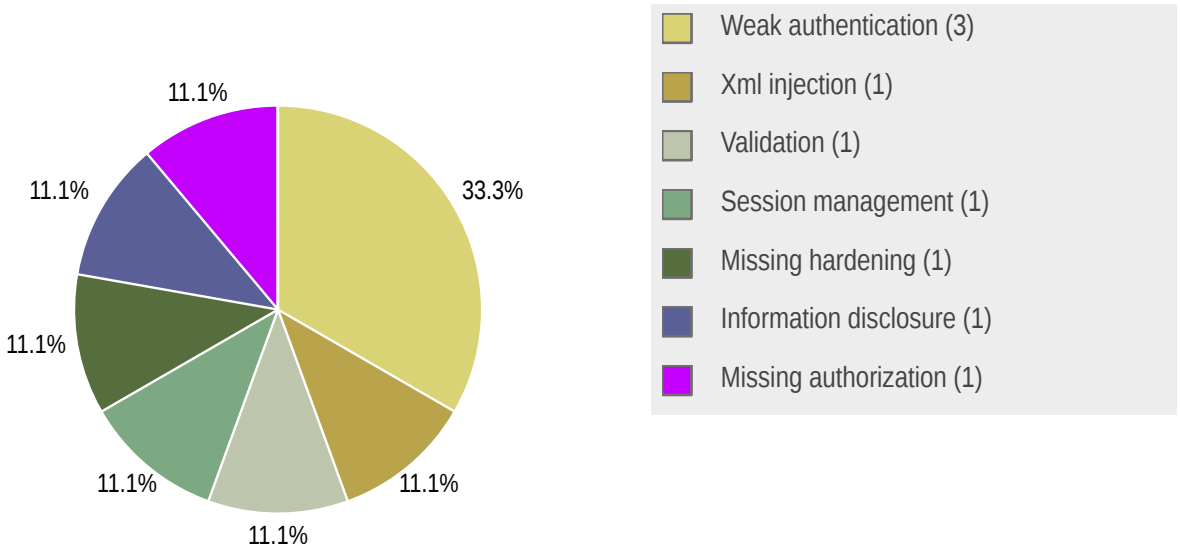
1.6 Summary of Findings

ID	Type	Description	Threat level
TLR-001	XML Injection	libeufin:nexus parses XML from untrusted sources using a parser with an insecure default configuration, making XML external entity attacks possible.	High
TLR-004	Weak Authentication	Initial admin account passwords and token values are generated using a Random Number Generator (RNG) that was not designed to be sufficiently unpredictable.	High
TLR-003	Validation	XML signature checks can be bypassed.	Moderate
TLR-005	Weak Authentication	The bank component enforces no password or lockout policy for password-based authentication. Users can pick any password upon password change and have unlimited numbers of guesses to log in.	Moderate
TLR-008	Session Management	The session token is not invalidated when the user logs out.	Moderate
TLR-002	Missing Hardening	The TLS layer of the HTTPS client used for EBICS traffic has the default Java configuration, which is a good baseline but misses some opportunities for hardening.	Low
TLR-006	Weak Authentication	The authentication endpoint POST /accounts/<username>/token responds differently for unsuccessful attempts depending on the existence of the username.	Low
TLR-007	Information Disclosure	Passwords used on the command line are visible to unprivileged users.	Low
TLR-012	Missing Authorization	The withdrawals abortion API endpoint lacks proper authorization.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
TLR-001	XML Injection	• Enable the secure processing feature in the parser.

TLR-004	Weak Authentication	• Use a cryptographically secure random number generator.
TLR-003	Validation	• Check explicitly whether processed nodes have the authenticate attribute set. • Alternatively, limit processing to only validated subtrees.
TLR-005	Weak Authentication	• Implement a password policy according to latest NIST recommendations. • Rate-limit login attempts, and block persistent failed attempts.
TLR-008	Session Management	• Issue tokens with a reasonable expiration (e.g., 30 minutes for an interactive financial application). • Refresh the token if the user stays active, providing a sliding expiration window. • If the user explicitly logs out, invalidate the token on the server by invoking the `delete("/accounts/{USERNAME}/token")` endpoint.
TLR-002	Missing Hardening	• Check OCSP and/or CRL, along with the presence of SCT to further harden the certificate validator.
TLR-006	Weak Authentication	• Make failed authentication requests return the same message, regardless of the validity of the username.
TLR-007	Information Disclosure	• Ask for the password on the TTY only, invoking `java.io.Console.readPassword()` (twice for confirmation) to hide the password as it is being entered.
TLR-012	Missing Authorization	• Implement strict authorization to ensure that users can only abort their own withdrawals.

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

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.

- nmap – https://nmap.org

4 Findings

We have identified the following issues:

4.1 TLR-001 — XML external entity injection

Vulnerability ID: TLR-001

Vulnerability type: XML Injection

Threat level: High

Description:

`libeufin:nexus` parses XML from untrusted sources using a parser with an insecure default configuration, making XML external entity attacks possible.

Technical description:

The `parseIntoDom` function is used by all parts of `libeufin:nexus` to parse XML into a DOM tree and it invokes `DocumentBuilderFactory` using its insecure default configuration that processes XML external entities.

Impact:

Since untrusted XML content is processed by this function (e.g. from EBICS responses), an attacker can exploit this vulnerability to use the server running Nexus to

- cause Denial-of-Service (DoS),
- read files stored on the local file system, and
- send HTTP requests to hosts that are only reachable from this server (SSRF)

Recommendation:

There are multiple ways to mitigate such issues, the most secure way is to enable the secure processing feature:

```
diff --git a/nexus/src/main/kotlin/tech/libeufin/nexus/XMLUtil.kt b/nexus/src/main/kotlin/tech/
libeufin/nexus/XMLUtil.kt
index fa4a993f..08efb35a 100644
--- a/nexus/src/main/kotlin/tech/libeufin/nexus/XMLUtil.kt
+++ b/nexus/src/main/kotlin/tech/libeufin/nexus/XMLUtil.kt
@@ -42,6 +42,7 @@ import javax.xml.transform.stream.StreamResult
 import javax.xml.xpath.XPath
```

```
 import javax.xml.xpath.XPathConstants
 import javax.xml.xpath.XPathFactory
+import javax.xml.XMLConstants

 /**
  * This URI dereferencer allows handling the resource reference used for
@@ -86,6 +87,7 @@ object XMLUtil {
     /** Parse [xml] into a XML DOM */
     fun parseIntoDom(xml: InputStream): Document {
         val factory = DocumentBuilderFactory.newInstance().apply {
+            setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true)
             isNamespaceAware = true
         }
         val builder = factory.newDocumentBuilder()
```

4.2 TLR-004 — Insufficient entropy used for credential generation

Vulnerability ID: TLR-004

Vulnerability type: Weak Authentication

Threat level: High

Description:

Initial admin account passwords and token values are generated using a Random Number Generator (RNG) that was not designed to be sufficiently unpredictable.

Technical description:

The same issue affects two parts of the codebase:

- In `bank/src/main/kotlin/tech/libeufin/bank/helpers.kt` function `createAdminAccount` generates the initial password for the admin account using the expression `ByteArray(32).rand()` and then decoding it as UTF-8.
- In `common/src/main/kotlin/TalerCommon.kt` methods `Base32Crockford32B.rand()` and `Base32Crockford64B.rand()` generate tokens using the expression `ByteArray(32).rand()` and `ByteArray(64).rand()`, respectively. (When presented to the user, these tokens are encoded using base-32.)

In all three cases, `ByteArray.rand()` is implemented by an extension method in `common/src/main/kotlin/helpers.kt` using `kotlin.random.Random.nextBytes()`, which is not designed to be unpredictable.

Impact:

An attacker could guess the conditions present at the time of generation (wall clock time, JVM, etc.) and have a reasonable chance at a brute force attack. This chance is made higher by the fact that there is no lockout policy for user logins and tokens – especially since there is no identifier for token authentication that could be the discriminator for such a lockout.

Recommendation:

Use a strong Cryptographically Secure (Pseudo) Random Number Generator (CSPRNG) for purposes where the output must be very unpredictable, such as passwords and tokens. The extension method in `common/src/main/kotlin/helpers.kt` already has a version that accepts a `SecureRandom` instance – this would be a good choice, used by many parts of the codebase already (such as `EbicsCommon` and `PwCrypto`).

4.3 TLR-003 — XML signature check bypass

Vulnerability ID: TLR-003

Vulnerability type: Validation

Threat level: Moderate

Description:

EBICS messages received in XML format have electronic signatures that are checked upon receipt. However, the combination of the behavior of the signature checks and later processing steps result in an opportunity for an attacker to bypass this validation step.

Technical description:

Methods `XMLUtil.verifyEbicsDocument` and `EbicsSigUriDereferencer.dereference` identify signature nodes and verify that their URIs match the hardcoded string `"#xpointer(//*[@authenticate='true'])"`. This is a proper protection against attacks without a valid signature.

However, this implementation ignores the rest of the XML document, and further processing (e.g. `EbicsBTS.parseResponse`) fails to check whether the attribute `authenticate` is set to `true` on the elements it encounters.

Impact:

An attacker can alter legitimate signed XML documents in a way that changes how they are interpreted, even when the signature check completes without issue. One example can be seen below, which alters the built-in test XML document in a way that the message within between `<foo>` and `</foo>` is changed, yet the unchanged signature still passes the test case `XmlUtilTest.testRefSignature`:

```
diff --git a/nexus/src/test/resources/signature1/doc.xml b/nexus/src/test/resources/signature1/
doc.xml
index 271f8429..6a2ceb98 100644
--- a/nexus/src/test/resources/signature1/doc.xml
+++ b/nexus/src/test/resources/signature1/doc.xml
@@ -5,5 +5,6 @@
         A8KcvluV1wGEBuakHL2t3GqFPQEfKW4l8GYTjHh/w9jBve5d8tvMOjGtoyNemZGrVlzBxO9+hwbw&amp;#13;
         8UFUCDA00dCjFDUHOnyAbBYsGzoaQyZprDn3iYDvlBz243zAN98PIKDclxlUEmkuF+JhrhCRjT9l&amp;#13;
         +JJxrELGHaDkFVadR4kaPdWPsbDaV0/2Fzc4Qg==</ds:SignatureValue></ebics:AuthSignature>
-    <foo authenticate="true">Hello World</foo>
+    <foo>Generic greetings!</foo>
+    <bar><foo authenticate="true">Hello World</foo></bar>
 </myMessage>
\ No newline at end of file
```

The parser will use the top-level `<foo>` node, not checking that the attacker had stripped its `authenticate` attribute, while the signature check will find it in the original one within the `<bar>` node, and happily report that it has not been altered in any way.

Recommendation:

Further processing (such as in `EbicsBTS.parseResponse`) should check explicitly whether the XML nodes it reads information from have the attribute `authenticate="true"` either directly or on one of their parents/ascendants.

Another approach would be for `XMLUtil.verifyEbicsDocument` to return the validated nodes and further processing is limited to those subtrees.

4.4 TLR-005 — Lack of password and lockout policy

Vulnerability ID: TLR-005

Vulnerability type: Weak Authentication

Threat level: Moderate

Description:

The `bank` component enforces no password or lockout policy for password-based authentication. Users can pick any password upon password change and have unlimited numbers of guesses to log in.

Technical description:

If the user provides an invalid password in `bank/src/main/kotlin/tech/libeufin/bank/db/AccountDAO.kt`, the function `checkPassword` receives `false` from `PwCrypto.checkpw`, but it only returns `CheckPasswordResult.PasswordMismatch` to `doBasicAuth` in `bank/src/main/kotlin/tech/libeufin/bank/auth/auth.kt`. Here, the exception `unauthorized` is thrown, which sets the HTTP status and response without further database interaction.

To change the password, `reconfigPassword` is called in `bank/src/main/kotlin/tech/libeufin/bank/db/AccountDAO.kt` which saves the new hash in the database without further checks on the value itself. This method is directly called by the route `patch("/accounts/{USERNAME}/auth")` in `bank/src/main/kotlin/tech/libeufin/bank/api/CoreBankApi.kt` which passes the user input directly without any validation.

Impact:

An attacker can launch on-line brute-force and password spraying attacks to achieve unauthorized access into accounts.

Recommendation:

Passwords should be checked according to a policy. NIST 800-63B recommends that

> passwords chosen by users should be compared against a blocklist of unacceptable passwords. This list should include passwords from previous breach corpuses, dictionary words used as passwords, and specific words (e.g., the name of the service itself) that users are likely to choose

and

> Verifiers and CSPs SHALL require passwords to be a minimum of eight characters in length and SHOULD require passwords to be a minimum of 15 characters in length.

For the first part, a reasonable choice is the Have I Been Pwned API (https://haveibeenpwned.com/API/v3#PwnedPasswords) that can be queried in a private manner.

Login attempts should be throttled in a way that discourages brute-force attacks. This can be implemented on a per-user and/or per-IP basis, and the resulting action can range from full-on blocking to softer measures such as a CAPTCHA. These actions should have a reasonable timeout that offers a compromise between security and usability.

See the relevant section of the OWASP Authentication cheat sheet for more details.

4.5 TLR-008 — Web frontend implements insecure session management

Vulnerability ID: TLR-008

Vulnerability type: Session Management

Threat level: Moderate

Description:

The web frontend of the `bank` component issues forever tokens, and when the user clicks on "Log out", the token is not invalidated, even though the API does have an endpoint (`delete("/accounts/{USERNAME}/token")`) for that specific purpose.

Technical description:

Upon login, the web frontend of the `bank` component sends the following HTTP request:

```
POST /accounts/admin/token HTTP/1.1
Authorization: Basic ...
Content-Type: application/json

{"scope":"readwrite","duration":{"d_us":"forever"},"refreshable":true}
```

Later, when the user clicks on "Log out", no HTTP traffic is sent at all. Replaying a previous request that used the token confirms that no invalidation happened on the server side, thus the token remains valid indefinitely.

Impact:

An attacker that manages to obtain a session token can use it as long as they want to, as neither the "Log out" functionality, nor an expiration time will invalidate the session.

Recommendation:

- Issue tokens with a reasonable expiration (e.g., 30 minutes for an interactive financial application).
- Refresh the token if the user stays active, providing a sliding expiration window.

- If the user explicitly logs out, invalidate the token on the server by invoking the `delete("/accounts/{USERNAME}/token")` endpoint.

4.6 TLR-002 — Missing TLS client hardening measures

Vulnerability ID: TLR-002

Vulnerability type: Missing Hardening

Threat level: Low

Description:

The TLS layer of the HTTPS client used for EBICS traffic has the default Java configuration, which is a good baseline but misses some opportunities for hardening.

Technical description:

The Ktor `HttpClient` is used to send HTTP requests to EBICS, and the TLS layer used for HTTPS communication uses Java defaults, which can vary between versions and vendors. Based on our experience with `openjdk-17-jre-headless` on a Debian system during testing, several of the tests marked as *bad* on `badssl.com` failed (the request went through without exceptions). Two of these have security relevance in the context of an M2M API, as is used here:

- `revoked.badssl.com` – the revocation status was not checked
- `no-sct.badssl.com` – the presence of Signed Certificate Timestamp (SCT) was not checked

Impact:

Since these are hardening measures, exploitation is limited to highly specific scenarios with motivated and sophisticated attackers. Impact is further lowered by the fact that TLS is just one of the layers of protection in transit, EBICS messages can also have signatures.

In case of revocation, attackers compromise the server of the bank and steal the private key that matches the certificate. If the bank discovers this, they can ask the CA (Certificate Authority) to revoke the certificate but as the EBICS client does not check revocation status (CRL, OCSP), it will accept the certificate as valid until its original expiration date.

In case of SCT, attackers strong-arm or bribe a CA to sign a fraudulent certificate with the bank's FQDN but the attacker's public key. If this gets into the Certificate Transparency logs, this can be noticed quickly. If not, the lack of SCT can be a sign that the certificate was not submitted to the CT system.

Recommendation:

- Check OCSP and/or CRL, along with the presence of SCT to further harden the certificate validator.

4.7 TLR-006 — Username oracle

Vulnerability ID: TLR-006

Vulnerability type: Weak Authentication

Threat level: Low

Description:

The authentication endpoint `POST /accounts/<username>/token` responds differently for unsuccessful attempts depending on the existence of the username.

Technical description:

If the user provides an invalid password, in `bank/src/main/kotlin/tech/libeufin/bank/db/AccountDAO.kt`:

- If the user exists, the function `checkPassword` gets `false` from `PwCrypto.checkpw` and it returns `CheckPasswordResult.PasswordMismatch` to `doBasicAuth` in `bank/src/main/kotlin/tech/libeufin/bank/auth/auth.kt`.
- If the user does not exist, the function `checkPassword` returns `CheckPasswordResult.UnknownAccount` to `doBasicAuth`.

Here, the exception `unauthorized` is thrown, which sets the HTTP status and response with two possible messages that are returned in the JSON body under the attribute `hint`:

- `"Bad password"`
- `"Unknown account"`

This difference in responses allows an attacker to know whether a username exists.

Impact:

An attacker can determine whether a given username exists in the system. When combined with the lack of rate limiting inTLR-005 (page 14), it provides a method for easy user enumeration.

Recommendation:

Unless business requirements call for the current behavior, make failed authentication requests return the same message, regardless of the validity of the username. This should also apply to timing, so for non-existent users, the method `checkpw` should be called to calculate a hash based on a random salt.

4.8 TLR-007 — CLI asks for password as command line argument

Vulnerability ID: TLR-007

Vulnerability type: Information Disclosure

Threat level: Low

Description:

CLI (Command Line Interface) tools such as `libeufin-bank passwd` allow the user to pass a password as a command line argument – and in fact this is the only way it allows the password to be entered. Most common operating systems allow any user to read the full command line (including arguments) of all processes.

Technical description:

As can be seen below, user `lowlevel` can read the `admin` password (`Test1234`) as it is being changed, even though the process was launched by `root` (first column of output):

```
lowlevel@taler:~$ id
uid=1000(lowlevel) gid=1000(lowlevel)
 groups=1000(lowlevel),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),100(users),106(netdev)
lowlevel@taler:~$ ps awwux | grep passwd
root       78149  146  2.1 3619544 87868 pts/0   Sl+  16:55   0:00 java -jar /usr/lib/bank-0.14.1-
all.jar passwd admin Test1234
lowlevel   78187  0.0  0.0   6332  2048 pts/1    S+   16:55   0:00 grep passwd
```

Impact:

Other (human or technical) users can read sensitive credentials this way just as they are being set. The probability of exploitation is lowered by the fact that the attacker only has a short window of opportunity – though the startup latency of the JVM being used helps with this. On the other hand, the `/proc` file system (or its equivalent source of process

information on non-Unix-like operating systems) could be polled continuously by a persistent attacker without much difficulty.

Recommendation:

- Ask for the password on the TTY only, invoking `java.io.Console.readPassword()` (twice for confirmation) to hide the password as it is being entered.

4.9 TLR-012 — Lack of authorization

Vulnerability ID: TLR-012

Vulnerability type: Missing Authorization

Threat level: Low

Description:

The withdrawals abortion API endpoint lacks proper authorization. Even though the user must authenticate themselves to invoke this action, no check is performed to validate that the withdrawal being aborted belongs to the current user.

Technical description:

In `CoreBankApi.kt` the endpoint `post("/accounts/{USERNAME}/withdrawals/{withdrawal_id}/abort")` is within an `auth` scope, however, unlike other adjacent authenticated endpoints, it makes no use of the variable `username` that could be used as part of a SQL `WHERE` clause to ensure that users can only abort their own withdrawals.

Impact:

An authenticated user in possession of the ID of another user's withdrawal process ID could abort the withdrawal. The probability of exploitation depends on how hard it is to get or guess such an ID.

Recommendation:

- Implement strict authorization to ensure that users can only abort their own withdrawals.

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-009 — Authorization testing

We performed authorization testing in a declarative manner, using centralized decorator-like method calls with closures, which is considered best practice. Endpoints fell into the following categories:

- Public data: no authorization necessary
- Admin endpoints: administrative access was checked properly
- User-specific endpoints: username was inside URL and was matched to authenticated user
- Object-specific endpoints: same as read-only, with the addition of matching the object's relation to the user

One interesting outlier was the query of withdrawals, which can be performed without authentication, but based on information from the developers, this is by design for wallet apps.

5.2 NF-010 — SQL injection testing

We analyzed database operations one-by-one for possible SQL injection. The system uses PostgreSQL directly (that is, without an ORM). Regular statements are augmented with stored procedures. All SQL invocations (including Kotlin code and `EXECUTE` in stored procedures) either use hardcoded queries or implement templating using hardcoded snippets – no user input is incorporated into strings passed to the parser.

5.3 NF-011 — File operations testing

Most of the persistence tasks are handled by the PostgreSQL database, the few exceptions are the ones below:

- Configuration files are only read by the application, not written.
- Key files (bank public key, libeufin private key) are written to a hardcoded location, using secure file permissions; only the (technical) user that created the file can read it.

The system has no code paths where file/path operations incorporate untrusted input, thus no path traversal is possible. (Obviously, the XXE vulnerability can be used for such purposes, but that is outside the scope of this section.)

6 Future Work

- **Retest of findings**
  When mitigations for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Let us emphasize that security is a process, and this penetration test is just a single snapshot. Security must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.
- **Set up automated source code scanning**
  Automated source code scanning can help to test the code against known issues and antipatterns. Tools like Semgrep analyze source code at the AST level, meaning that patterns can be identified regardless of whitespace and semantic noise around the affected areas. Although such patterns are advanced and mature for Java, Kotlin ones are improving, and it could be an interesting exercise to implement regression test cases for the vulnerabilities identified. These could also be upstreamed to the Semgrep community repositories to help other projects as well. This does not replace manual testing, but rather complements it to catch low-hanging fruit with minimal long-term effort. Ideally, it should be integrated into a CI/CD pipeline so that issues can be caught as easily and reliably as possible.

7 Conclusion

We discovered 2 High, 3 Moderate and 4 Low-severity issues during this penetration test.

The libeufin-nexus component of the GNU Taler system has a limited attack surface, so the findings within this report can be interpreted in two ways. On the one hand, the most severe issues can be exploited by attackers with sufficient privileges (e.g., your own bank) or significant preparedness. On the other hand, many things that could go wrong within this limited attack surface did go wrong, so attackers could guess the administrative password, get access to local files on the system, launch SSRF attacks against other servers, or tamper with signed transactions, just to name a few of the most impactful vulnerabilities.

Some of these are the result of the platform chosen for the implementation (e.g., the Java XML parser's default settings), while many of them are the result of the wheel being reinvented with an approach where not enough care was taken over security aspects. Examples of the latter include session handling and authentication, where off-the-shelf solutions such as KeyCloak would have taken care of password and lockout policy, along with session expiration and logout.

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.

Appendix 1 Testing team

András Veres-Szentkirályi	András has CISSP, OSCP, GWAPT, and SISE certifications in addition to an MSc. in Computer Engineering, and has been working in IT Security since 2009.
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.