RADICALLY OPEN SECURITY

Penetration Test Report

Relaycorp, Inc.

V 1.0
Amsterdam, April 25th, 2025
Public

Document Properties

Client	Relaycorp, Inc.
Title	Penetration Test Report
Targets	• VeraID protocol • VeraID cloud native app/API • VeraID client libraries
Version	1.0
Pentesters	Sipke Mellema, Andrea Jegher
Authors	Sipke Mellema, Andrea Jegher, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	April 10th, 2025	Sipke Mellema, Andrea Jegher	Initial draft
0.2	April 23rd, 2025	Marcus Bointon	Review
1.0	April 25th, 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

Between March 27, 2025 and April 11, 2025, Radically Open Security B.V. carried out a penetration test for Relaycorp, Inc.

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:

- VeraID protocol
- VeraID cloud native app/API
- VeraID client libraries

The scoped services are broken down as follows:

- Penetration testing of VeraID Cloud Native App/API: 3 days
- Code Review / Testing of VeraID Client Libs semantics: 4 days
- Code Review / Testing of DNSSEC semantics: 2 days
- Reporting: 2 days
- **Total effort: 11 days**

1.3 Project Objectives

ROS will perform a penetration test of the VeraId protocol with Relaycorp in order to assess the security of the protocol and its implementation. To do so ROS will access the VeraId open source repositories and guide Relaycorp 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 27, 2025 and April 11, 2025.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 1 Elevated, 4 Low and 1 N/A-severity issues.

The most impactful issue was found in the VeraId Authority server. Secret tokens used to import public keys are partially predictable CLN-005 (page 12). This predictability could allow an attacker to add public keys to other users' keys.

The project uses outdated packages with dependencies that include known vulnerabilities CLN-011 (page 17). While these vulnerabilities are not easily exploitable, they still pose a moderate risk and should be addressed in future updates.

A Denial of Service (DoS) vulnerability exists in the NodeJS implementation CLN-008 (page 16). By manipulating the signature bundle, an attacker could crash the application, potentially disrupting availability.

The system VeraId Authority server URL validation only on the first of two requests made when validating a JWT token during a signature spec request. This inconsistent filtering could allow malicious content to bypass security controls CLN-007 (page 15).

We identified a NoSQL injection pattern in the VeraId Authority server CLN-006 (page 13). This pattern could be exploited to bypass authorization mechanisms, putting sensitive data at risk.

We found multiple inconsistencies in how the Java and NodeJs implementations parse data CLN-010 (page 18). These variations can lead to unpredictable behavior and potential security weaknesses if not carefully managed.

The JWT verification process does not enforce the `Issued At` claim, but this was not considered a vulnerability, as it aligns with accepted practices in certain contexts non-finding NF-009 (page 21).

We reviewed the GitHub workflow scripts for shell injection vulnerabilities through developer-supplied inputs, but did not find any exploitable patterns.

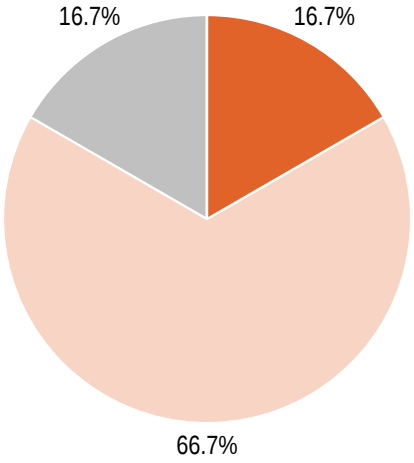
The report includes an issue regarding protocol-level security decisions too, which were not explicitly detailed in this summary but are worth deeper review non-finding NF-012 (page 21).

1.6 Summary of Findings

Info	Description
CLN-005 **Elevated** **Type:** CWE-639: Authorization Bypass Through User-Controlled Key **Status:** none	The secret tokens that are generated to import public keys via the Awala service are partially predictable, which could allow an attacker to manipulate keys belonging to others.

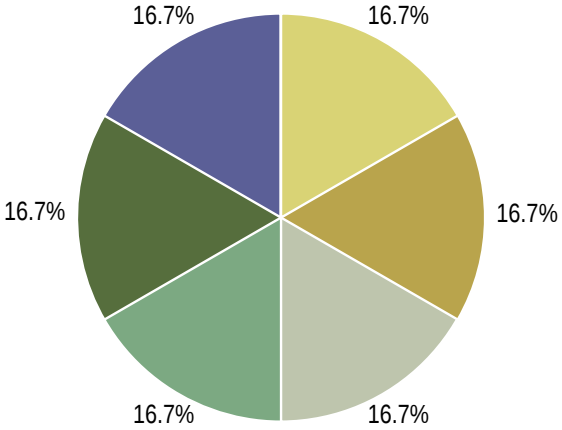
CLN-006 **Low** **Type:** CWE-943: Improper Neutralization of Special Elements in Data Query Logic **Status:** none	The veraid-authority application uses a coding pattern that allows NoSQL injection, which can result in an authorization bypass.
CLN-007 **Low** **Type:** Server-Side Request Forgery **Status:** none	Members can create a signature bundle on the Authority server. They can provide a URL to a custom identity server. Two requests are made when the bundle is retrieved, but only the initial URL is validated.
CLN-008 **Low** **Type:** Denial of Service **Status:** none	The NodeJS implementation is vulnerable to denial of service; an attacker can manipulate the signature bundle to crash the application.
CLN-011 **Low** **Type:** Vulnerable and Outdated Components **Status:** none	The VeraID packages contain outdated dependencies with known vulnerabilities that are not easily exploited.
CLN-010 **N/A** **Type:** Inconsistent Parsing **Status:** none	There are multiple differences between the parser implementations for NodeJS and Kotlin.

1.6.1 Findings by Threat Level



- Elevated (1)
- Low (4)
- N/A (1)

1.6.2 Findings by Type



- CWE-639: Authorization Bypass Through User-Controlled Key (1)
- CWE-943: Improper Neutralization of Special Elements in Data Query Logic (1)
- Server-Side Request Forgery (1)
- Denial of Service (1)
- Vulnerable and Outdated Components (1)
- Inconsistent Parsing (1)

1.7 Summary of Recommendations

Info	Recommendation
CLN-005 **Elevated** **Type:** CWE-639: Authorization Bypass Through User-Controlled Key **Status:** none	• Use cryptographically random tokens like a UUIDv4 or v7 for security-sensitive operations.
CLN-006 **Low** **Type:** CWE-943: Improper Neutralization of Special Elements in Data Query Logic **Status:** none	• Force the email address to be a string with the `toString` function.
CLN-007 **Low** **Type:** Server-Side Request Forgery **Status:** none	• Perform the same level of URL validation on both URLs (`providerIssuerUrl` and `jwks_uri`).
CLN-008 **Low** **Type:** Denial of Service **Status:** none	• Improve the way that the zones are parsed to prevent the application entering an infinite loop.
CLN-011 **Low** **Type:** Vulnerable and Outdated Components **Status:** none	• Update the packages to their latest version on a regular basis.
CLN-010 **N/A** **Type:** Inconsistent Parsing **Status:** none	• Consider adding more details about parsing to the standard if the mentioned parsing differences are of security concern to future development.

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.

- Gitleaks – https://github.com/gitleaks/gitleaks
- CodeQL – https://codeql.github.com
- OpenGrep – https://github.com/opengrep/opengrep

4 Findings

We have identified the following issues:

4.1 CLN-005 — Insecure direct object reference

Vulnerability ID: CLN-005

Vulnerability type: CWE-639: Authorization Bypass Through User-Controlled Key

Threat level: Elevated

Description:

The secret tokens that are generated to import public keys via the Awala service are partially predictable, which could allow an attacker to manipulate keys belonging to others.

Technical description:

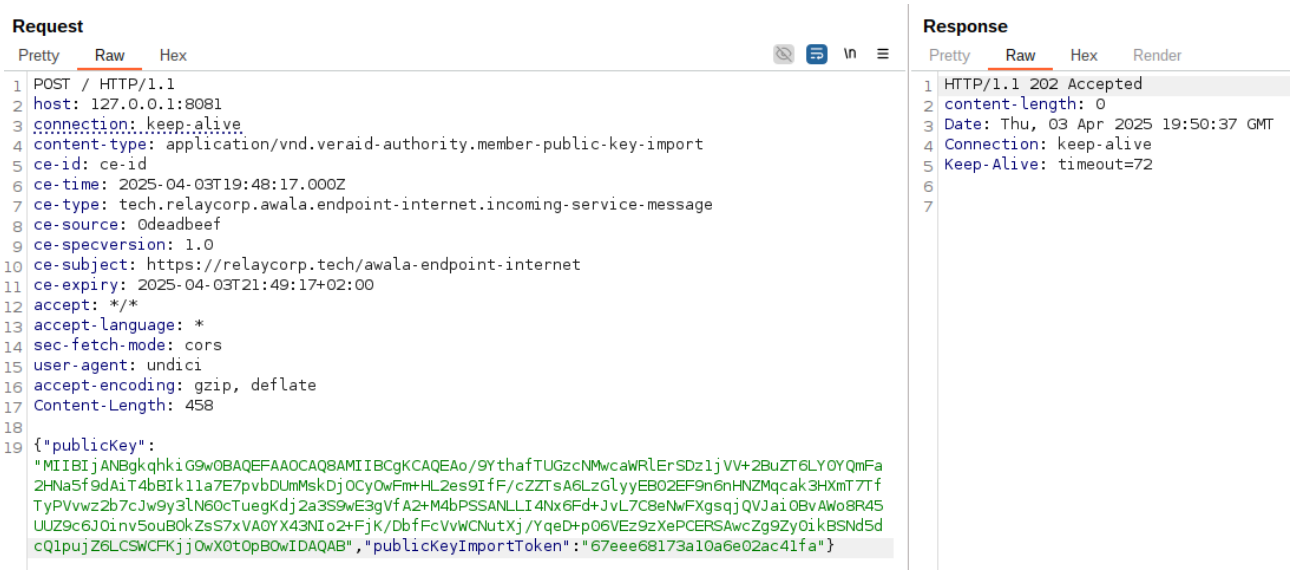
The veraid-authority application allows administrators to manage organizations and members. Members can also use the server to perform actions like set their public key.

Veraid-authority uses MongoDB IDs as object identifiers. This is used for members and public keys, but they are also used as secret tokens for changing a member's public key via the Awala endpoint `publicKeyImportToken`. MongoDB IDs are not cryptographically random; they are partially predictable. From the documentation:

> ObjectIds are small, likely unique, fast to generate, and ordered. ObjectId values are 12 bytes in length, consisting of: A 4-byte timestamp, representing the ObjectId's creation, measured in seconds since the Unix epoch. A 5-byte random value generated once per process. This random value is unique to the machine and process. A 3-byte incrementing counter, initialized to a random value.

Depending on the attacker's resources and the application deployment these IDs might not be secure. For example, if the MongoDB has only one process, the 5-byte random value is always the same. This leaves 4-7 bytes left to brute-force, which is feasible, even for an online attack.

The image below shows an attacker importing a public key for a user with a predictable import token via the Awala endpoint.



Impact:

An attacker with enough resources might be able to guess the identifier and set the public key for another user. This in itself has limited impact, because the attacker would need to also obtain the member bundle in order to spoof others. The ability to use the Awala endpoint also depends on the way the server is deployed.

Recommendation:

- Use cryptographically random tokens like a UUIDv4 or v7 for security-sensitive operations.

4.2 CLN-006 — NoSQL injection pattern

Vulnerability ID: CLN-006

Vulnerability type: CWE-943: Improper Neutralization of Special Elements in Data Query Logic

Threat level: Low

Description:

The veraid-authority application uses a coding pattern that allows NoSQL injection, which can result in an authorization bypass.

Technical description:

In `orgAuthPlugin.ts`, the organization name and the user's email are used to find a member with the `findOne` function.

```
async function decideAuthorisation(
[..]
  const member = await memberModel.findOne({ orgName, email: userEmail }).select(['role']);
```

`MemberModel` is a Mongoose model that prevents NoSQL injection by default. The snippet from the member model below shows that the `orgName` is a string, so Mongoose will convert it to a string during conversion. However, the email field allows arbitrary types. If an attacker can change the value of `userEmail` to `{"$ne":"test"}`, the query will search for all emails that are not equal to "test", instead of searching for the literal value; this will match any email address.

```
export class Member {
  @prop({ default: null, allowMixed: Severity.ALLOW })
  public name!: string | null;

  @prop({ default: null, allowMixed: Severity.ALLOW })
  public email!: string | null;
[..]
  @prop({ required: true })
  public orgName!: string;
}
```

However, in this case the email is taken from the JWT token, so a user would need to manipulate the token in order to perform the attack.

Impact:

Veraid-authority uses a vulnerable coding pattern that is currently not directly exploitable. If the pattern is repeated in future development it might allow attackers to manipulate NoSQL queries to change the application's behavior.

This example would require an attacker to find a bug in the identity server such that the email can be manipulated to any value. This level of access can be used to pose as any user, so the injection attack would only be useful to the attacker if they don't know any email addresses.

Recommendation:

- Force the email address to be a string with the `toString` function.

4.3 CLN-007 — Inconsistent URL filtering

Vulnerability ID: CLN-007

Vulnerability type: Server-Side Request Forgery

Threat level: Low

Description:

Members can create a signature bundle on the Authority server. They can provide a URL to a custom identity server. Two requests are made when the bundle is retrieved, but only the initial URL is validated.

Technical description:

A user provides a `providerIssuerUrl` in the `signatureSpec` endpoint. When a bundle is downloaded, the jwks configuration is retrieved from that URL. From that response, the server takes the `jwks_uri` and uses it to make a second request. Only the first `providerIssuerUrl` is validated.

In `jwksRetrieval.ts`, `sanitiseJwksUri` uses the `URL` class to sanitize the URL, but this can still be *any* URL. The request to `jwks_uri` seems to be done with `undici`. This client doesn't support many URL types, only `data:` and `blob:`. A scheme like `file:` returns the error `"Error: not implemented... yet..."`, as can be read in the source code here: https://github.com/nodejs/undici/blob/9dd11b8c61c95efd5459f375a196a117184230fa/lib/web/fetch/index.js#.

Impact:

The URL is inconsistently filtered. An attacking member can use a custom identity server to manipulate the second URL. This currently has limited impact, but a future HTTP client might allow more advanced URL schemes like `file` to try and read files from the local filesystem.

Recommendation:

- Perform the same level of URL validation on both URLs (`providerIssuerUrl` and `jwks_uri`).

4.4 CLN-008 — Stack overflow denial-of-service vulnerability in NodeJS implementation

Vulnerability ID: CLN-008

Vulnerability type: Denial of Service

Threat level: Low

Description:

The NodeJS implementation is vulnerable to denial of service; an attacker can manipulate the signature bundle to crash the application.

Technical description:

The DNS zone-parsing logic in `name.ts` will get stuck in an endless loop with input like `.butsers.nl.` or `\_veraid.butsers.nl.`. This causes a stack overflow, so the application crashes.

`/dnssec-js/src/lib/utils/dns/name.ts`:

```
export function getZonesInName(zoneName: string, shouldIncludeRoot = true): readonly string[] {
  if (zoneName === '') {
    return shouldIncludeRoot ? ['.'] : [];
  }
  const parentZoneName = zoneName.replace(/^[^.]+\./u, '');
  const parentZones = getZonesInName(parentZoneName, shouldIncludeRoot);
  return [...parentZones, zoneName];
}
```

Impact:

An attacker can make the bundle validation logic crash by providing a malicious CN. If VeraID validation is implemented server-side, this could be used to crash that service. Also note that the DNS zones need to be traversed before signature checking can take place, so manipulation can't be detected before the crash.

Recommendation:

- Improve the way that the zones are parsed to prevent the application entering an infinite loop.

4.5 CLN-011 — Use of outdated packages

Vulnerability ID: CLN-011

Vulnerability type: Vulnerable and Outdated Components

Threat level: Low

Description:

The VeraID packages contain outdated dependencies with known vulnerabilities that are not easily exploited.

Technical description:

The partial output below shows dangerous vulnerabilities in dependencies used by the Authority server. These could allow bypasses in JWT-validation and command injection in certain advanced attack scenarios (which would require secondary vulnerabilities).

```
fast-jwt  <5.0.6
Severity: moderate
Fast-JWT Improperly Validates iss Claims - https://github.com/advisories/GHSA-gm45-q3v2-6cf8
[..]

protobufjs  7.0.0 - 7.2.4
Severity: critical
protobufjs Prototype Pollution vulnerability - https://github.com/advisories/GHSA-h755-8qp9-cq85
```

Impact:

The Authority server and other VeraID applications use outdated packages with known vulnerabilities. The issues are currently not directly exploitable, but over time new issues might stack up, and once a vulnerability is found, the others will make exploitation easier.

Recommendation:

- Update the packages to their latest version on a regular basis. During the pentest, the VeraID developers were aware of the issues and were already busy with a patch-management plan before the pentest started.

4.6 CLN-010 — Parser differences

Vulnerability ID: CLN-010

Vulnerability type: Inconsistent Parsing

Threat level: N/A

Description:

There are multiple differences between the parser implementations for NodeJS and Kotlin.

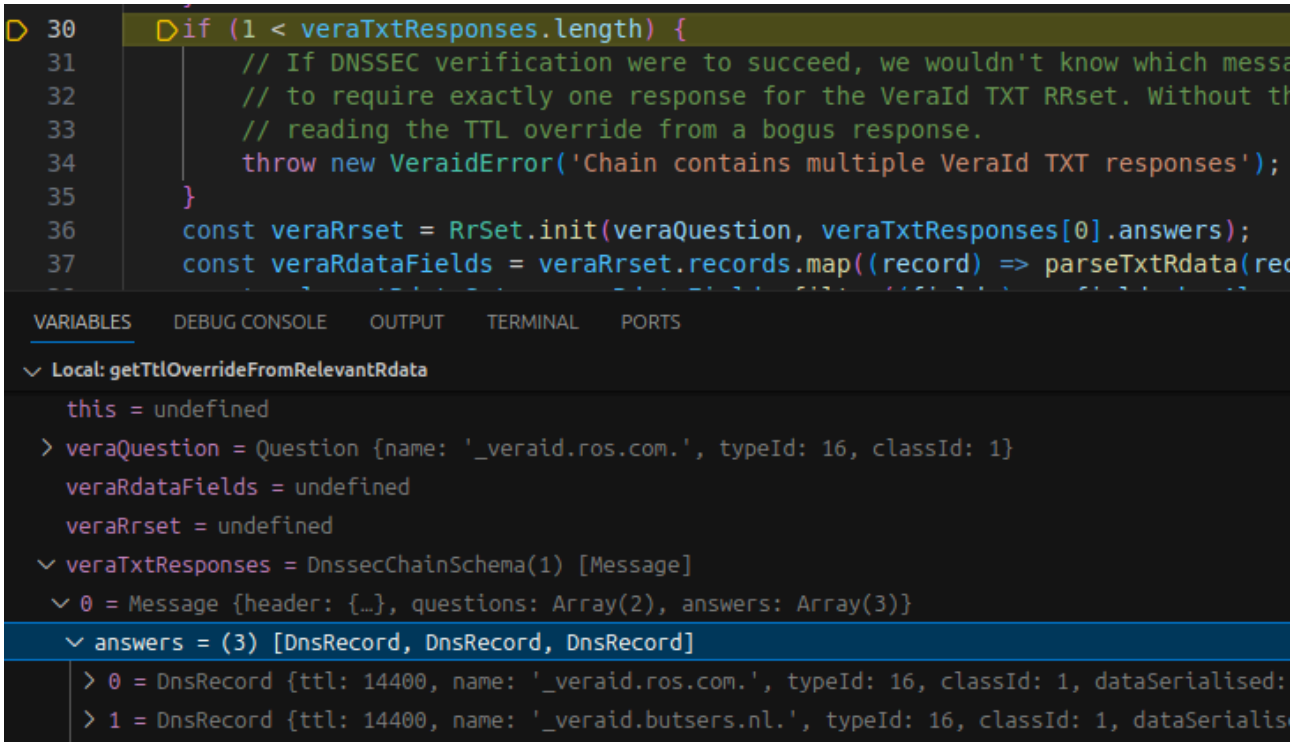
Technical description:

The CMS structure

Unsigned attributes can be included in the `signerinfo` structure (`initSignerInfo`) without throwing exceptions in the JS implementation. The BouncyCastle library for Kotlin throws an exception on this.

DNS chain filtering

Adding multiple answers in the final DNS response is ignored by Node, because it only looks at the CN from the org certificate. Kotlin throws an exception because the DNSSEC chain is insecure. An example is shown in the screenshot below (the answers for the other domain are filtered out after the rrset step below it).



Changing a DNS record type to 0 will make Node throw an exception because one of the responses is malformed (because it doesn't understand type 0), but the Kotlin implementation will say "DNSSEC verification failed: Unsigned response was proved to be validly INSECURE".

The organisation certificate

A member can manipulate the organisation certificate, like the extensions and other attributes, because only the public key and the subject are used (which can't be manipulated).

A member can add a CN to the organisation certificate. Adding the code below to `Certificate.js` will cause the Node implementation to throw an exception. The Kotlin implementation validation will pass, because it uses the first CN from a certificate.

```
pkijsCert.subject.typesAndValues.push(new AttributeTypeAndValue({
    type: COMMON_NAME,
    value: new BmpString({ value: options.commonName }),
}));
pkijsCert.subject.typesAndValues.push(new AttributeTypeAndValue({
    type: COMMON_NAME,
    value: new BmpString({ value: "www.ros.win" }),
}));
```

Impact:

These parsing differences have no security impact at the moment. They are mentioned as interesting cases that might be of concern to future development (like how the organisation certificate can be manipulated).

Recommendation:

- Consider adding more details about parsing to the standard if the mentioned parsing differences are of security concern to future development.

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-004 — GitHub run shell injection

The project shares GitHub Workflows in the repo called `shared-workflows`. Four workflows in this repo use developer inputs in the `run` step. These developer inputs are defined in the `inputs` field of a workflow trigger, in this case `workflow_call`.

1. `shared-workflows/.github/workflows/nodejs-compose-ci.yml` line 60
2. `shared-workflows/.github/workflows/nodejs-knative-ci.yml` line 164
3. `shared-workflows/.github/workflows/nodejs-server-ci.yml` line 110
4. `shared-workflows/.github/workflows/tfmodule-ci.yml` line 41

For example, see the `Export Docker image` step of the first workflow in the list:

```
- name: Export Docker image
run: |-
  set -o nounset
  set -o errexit
  set -o pipefail

  DOCKER_IMAGE_TAG="$(docker images "${{ inputs.docker_image_local_name }}" --format '{{.Tag}}' |
 tail -1)"
  TARGET_IMAGE_TAG="${GITHUB_REPOSITORY,,}:ci"
  docker tag "${{ inputs.docker_image_local_name }}:${DOCKER_IMAGE_TAG}" "${TARGET_IMAGE_TAG}"
  docker save -o /tmp/docker-image "${TARGET_IMAGE_TAG}"
```

Note that variable `DOCKER_IMAGE_TAG` is created from `"${{ inputs.docker_image_local_name }}"`. If an attacker can control the input `docker_image_local_name`, they could inject commands with strings like `` `whoami` `` or `; echo command ;`.

This is not reported as an issue because the injections are only present in actions triggered by a `workflow_call` and so only other actions can trigger them. Additionally, of course, developers might change the workflow or the code to be malicious, and there is only a remote use case where user without permissions to write on the repository but can trigger the workflows manually. Even in this case there is nothing to gain from this attack.

Nonetheless, this issue can be mitigated with an intermediate environment variable from a step `env` to store the data. We report a generic example that you can find in the GitHub documentation:

```
- name: Check PR title
env:
  TITLE: ${{ github.event.pull_request.title }}
run: |
  if [[ "$TITLE" =~ ^octocat ]]; then
  echo "PR title starts with 'octocat'"
  exit 0
```

```
else
echo "PR title did not start with 'octocat'"
exit 1
fi
```

5.2 NF-009 — JWT verification doesn't require Issued At claim

The JWT token verifier makes it optional to include the time at which the token was generated. An attacker with a token that doesn't hold such a claim can use it indefinitely.

Non finding because idp doesn't need to set `iat`.

5.3 NF-012 — Protocol considerations

Improvements

- It might be good if people can submit invalid bundles somewhere to detect attack trends.
- To detect a compromised organization, TLD or IANA, an application could, on boot, store the DNSSEC keys for the active organization and most of the TLDs. If they are changed before their expiry time, the user can be shown an alert about a potential compromise.
- Consider using an allowlist instead of a denylist for member names. ROS was informed that this is already being included in the VeraID standard draft.

Threats

- A large part of the security design relies on how the DNS parsers parse the DNSSEC chain. A bug in the parser might allow an attacker to break security assumptions. For example, a golden bug would be a record that is seen holding the public key that is not included in the rrset.
- Key management is an obvious challenge. If the private keys from the organization or the member are compromised it breaks their security abilities.
- There is also the threat of an attacker sending an invalid member bundle to a member. If they import it and override their bundle, they can't sign anything until they get online again (and download a new bundle). There is a `verify` method on the bundle model, but it was not tested because it wasn't included in the demo application.

5.4 NF-013 — Signature bundle verification fuzzing

Fuzzing is an automated software testing technique that feeds unexpected or random inputs to a program to discover security vulnerabilities. During the analysis we wrote a quick and simple fuzzer to see if it was possible to alter a

valid signature bundle in a way that it would be valid, but with different essential data (e.g. the name of the signer). A secondary goal was to find which fields in the signature bundle could be changed without invalidating it, which may help to find additional issues.

We report here the full bash script we wrote to fuzz signature bundle verification. It uses the `verify.js` script from the VeraId demo using the `veraid-js` library. We used the code from the same demo to generate the organization and member certificates, along with the member bundle and signature bundle. The only other requirement is Radamsa, a test case generator for robustness testing, a.k.a. a fuzzer.

```
#!/bin/bash
prefix="${1}"
if [ "${prefix}" == "" ]; then
    prefix="${RANDOM}"
fi

while true
do
    out="./fuzz/out${prefix}_${RANDOM}.fuzz"
    # If grep find the `successful` string in the output then the signature bundle variation was
 still valid
    cat signature.bundle | radamsa | tee "${out}" | ./verify.js "1.3.6.1.4.1.58708.1.1" | grep -q
 successful
    if [ $? -eq 0 ]; then
        echo win
        mv "${out}" "${out}.win"
    else
        rm ${out}
    fi
done
```

Results

Around 24 hours of running 7 instances of this script produced 840 variations of a valid signature bundle that were valid but different; **none** of these contained alterations of the signed data or the signer name, or had meaningful alterations to the DNSSEC chain.

Most of the changes happened in DNSSEC records that were not taken into account, like SOA, domain names of DNS queries that did not have answers, and unused padding.

Even if the results did not find a vulnerability in the signature bundle verification, they highlighted that many of the fields in the signature bundle are not essential and can be altered without changing the validity of the file. This could be important for further fuzzing since it might be necessary to create more specific checks when deciding whether a variation is meaningful or not for the use case. On the other hand, since it is fairly easy to create a valid variation from an additional file it could be an indication that fuzzing could provide hints on which parts of the signature bundle are actually used or not, perhaps helping further security research.

6 Future Work

- **DNSSEC Parsing**
  Perform dedicated testing on the DNSSEC chain parsing and validation code. The DNSSEC chain parsing is a single point of failure for the system; if an attacker can manipulate it, they might be able to forge arbitrary signature bundles.
- **Signature bundle fuzzing**
  Continue the fuzzing work with a more sophisticated process, perhaps one that supports code coverage, and with an automated means of comparing signature bundle variations. Also consider that time is an important factor in fuzzing, and it might take months to find a meaningful variation.
- **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.

7 Conclusion

We discovered 1 Elevated, 4 Low and 1 N/A-severity issues during this penetration test.

The review followed several approaches to find potential issues in the code. We started from the available VeraId demo to generate valid signature bundles and certificates, and deployed `VeraId` on our own domains. After understanding and researching the environment, we proceeded with a manual review of the code, writing custom tests to check for corner cases and see if our attacks were feasible. For example, we used the testing unit of `veraid-js` to check whether it would be possible for a member to forge new certificates with their own, which proved to not be possible unless the organization allows it. Lastly, we tried to perform fuzz testing on the signature bundle verification process to try to find possible valid variations of a genuine signature bundle.

This security audit identified a few vulnerabilities and risky implementation patterns that could impact the overall security posture of the system. The most important issue is the use of MongoDB IDs that could be predictable, depending on the deployment, as Awala public key import tokens. The remaining issues relate to input validation; they do not pose a threat to the system as a whole, but only in certain specific cases.

Nonetheless, the overall implementation and the protocol itself appear sound and robust. The design demonstrates a thoughtful approach to security, with many aspects of the threat model explicitly considered and reflected in the implementation. Furthermore, the presence of defensive coding patterns and checks across the code base suggests a clear effort to anticipate and mitigate potential vulnerabilities. While no system is entirely immune to risk, the current implementation shows a strong foundation that can be built upon with targeted improvements.

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

Sipke Mellema	Sipke is an experienced pentester with nine years of experience in the field. His specialty is crystal-box web security and providing long-term advice on security improvements. Over the years he has branched out to the cloud, IoT, ICS, network security, and security management.
Andrea Jegher	Andrea is a security engineer with experience in offensive security and secure development. He started his career focusing on Web Application as a developer and as a penetration tester. Later he studied other fields of security such as cloud, networks and desktop applications.
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.