Psono Review
for esaqa GmbH

Final Report and Management Summary

2024-07-22

CONFIDENTIAL

X41 D-Sec GmbH
Krefelder Str. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

Revision	*Date*	*Change*	*Author(s)*
1	2024-07-22	Final Report and Management Summary	Gregor Kopf, E. Sesterhenn

Contents

Dashboard

Target	
Customer	esaqa GmbH
Name	Psono
Type	Cryptographic Implementations
Version	As deployed between 2024-06-17 and 2024-07-08

Engagement	
Type	White Box Review
Consultants	1: Gregor Kopf
Engagement Effort	9 person-days, 2024-06-17 to 2024-07-08

Total issues found	4

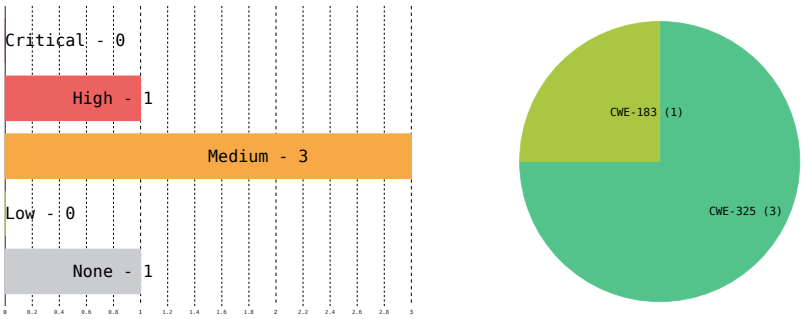

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

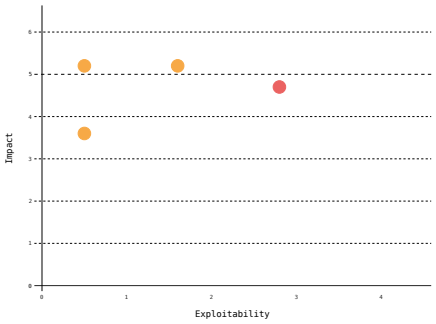

Figure 2: CVSS Impact and Exploitability Distribution

1 Executive Summary

In June 2024, X41 D-Sec GmbH preformed a source code review in combination with a dynamic test against the Psono solution, aiming to identify security vulnerabilities and weaknesses in the implementation. Special focus was requested on the correct usage of cryptographic algorithms and methods.

A total of four vulnerabilities were discovered during the test by X41. None were rated as having a critical severity, one as high, three as medium, and none as low. Additionally, one issue without a direct security impact was identified.

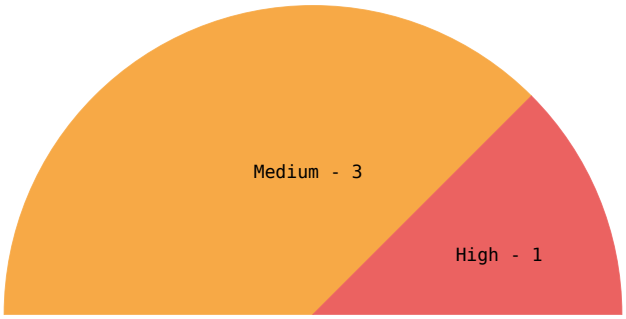

Figure 1.1: Issues and Severity

The Psono solution offers an end-to-end encrypted password manager. Given the sensitivity of the data handled within the implementation, one core aspect of the review was to identify possible cryptographic weaknesses, which could allow for passwords or other sensitive information to leak to untrusted third parties. Furthermore, the implementation was also subject to a general review, focusing on typical implementation-level issues.

In a white box penetration test, the testers receive all available information about the target, including source code. That was the case with this project. The source code has been obtained from the public Psono git repositories.

The test was performed by one experienced security expert between 2024-06-17 and 2024-07-08.

The most severe issue discovered allows an attacker to trick clients into using an encryption key that is also known to the server, thereby breaking the encryption of sensitive information. It should be noted, however, that such an attack is likely most effective against accounts that have not yet stored any data. Another issue allows malicious clients to access at least the `/tmp` directory of the server, which could - depending on the particular deployment - lead to the disclosure of sensitive information or even code execution issues.

Overall, the solution appears to be on a good security level compared to systems of similar size and complexity. While a number of vulnerabilities and weaknesses could be identified, it should be stressed that the solution appears to be developed with security in mind. This is reflected for instance by the choice of the used cryptographic library (NaCL), which offers secure and modern cryptographic primitives. Furthermore, a number of common vulnerability types such as SQL injection issues appear to have been addressed upfront.

2 Introduction

X41 reviewed the Psono solution, which provides an end-to-end encrypted password manager. The solution can be self-hosted, and the source code is freely available.

Given the sensitive nature of the data handled by Psono, a particular emphasis was put on reviewing the cryptographic design and implementation. One of the most important goals of an attacker is expected to be the extraction of the users' confidential information. Therefore, the review was conducted with this attack scenario in mind.

2.1 Methodology

The review was performed as a source code review in combination with manual dynamic testing.

A manual approach for penetration tests and for code reviews is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools[1].

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[2] standards and the *Study - A Penetration Testing Model*[3] of the German Federal Office for Information Security.

[1] `https://portswigger.net/burp`
[2] `https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards`
[3] `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1`

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Traffic Encryption not Enforced	MEDIUM	PS-CR-24-01	4.1.1
Parsing of PostScript Files Can Lead to Limited File System Access	HIGH	PS-CR-24-02	4.1.2
Possible Confusion of Secret Data	MEDIUM	PS-CR-24-03	4.1.3
Cryptographic Key Disclosure to Rogue Server	MEDIUM	PS-CR-24-04	4.1.4
Missing Domain Separation	NONE	PS-CR-24-100	4.2.1

Table 2.1: Security-Relevant Findings

2.3 Scope

The scope of the review included the Psono backend server, as well as the web and mobile client implementations. The source code was obtained from the following public git repositories:

- `git clone -depth 1 -branch v1.8.8 https://gitlab.com/psono/psono-app`
- `git clone -depth 1 -branch v3.1.1 https://gitlab.com/psono/psono-client`
- `git clone -depth 1 -branch v4.0.11 https://gitlab.com/psono/psono-server.git`

A Slack channel was made available for direct communication with the client.

2.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

While the time allocated to X41 for this assessment was sufficient to achieve coverage over the most sensitive parts of the implementation, a particular emphasis of the review was put on analyzing the used cryptographic schemes, which consumed a non-negligible fraction of the project's time budget as requested by esaqa GmbH beforehand.

During the review, the in-scope components have been analyzed based on an an internally developed threat model. As the codebase was new to X41, possibly severe vulnerability classes have been prioritized. This includes - as outlined above - possible cryptographic issues, as well as problems that could lead to code execution issues on the server side. This is due to the fact that

such issues have the potential to directly break all security guarantees offered by Psono. While client-side issues have also been investigated, less emphasis was put on them.

2.5 Recommended Further Tests

Possible future tests of the Psono solution could put additional focus on the client-side implementations, particularly focusing on platform-specific issues. Furthermore, in case the cryptographic protocols are changed (e.g., in response to the issues outlined in this report), it appears to be advisable to perform another review in order to make sure that there are no adverse side-effects of such changes.

3 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers when they are discovered during a test. Business factors and financial risks for esaqa GmbH are beyond the scope of a penetration test, which focuses entirely on technical factors. However, technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

The CVSS[1] is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

3.1 CVSS

All findings relevant to security are rated by the testers using the CVSS industry standard version 3.1, revision 1.

Vulnerabilities scored with CVSS get a numeric value based on several metrics ranging from 0.0 (least worst) to 10.0 (worst).

The score captures different factors that express the impact and the ease of exploitation of a vulnerability among other factors. For a detailed description of how the scores are calculated, please see the CVSS version 3.1 specification.[2]

The metrics used to calculate the final score are grouped into three different categories.

[1] Common Vulnerability Scoring System
[2] `https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf`

The *Base Metric Group* represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments. It captures the following metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Privileges Required (PR)
- User Interaction (UI)
- Scope (S)
- Confidentiality Impact (C)
- Integrity Impact (I)
- Availability Impact (A)

The *Temporal Metric Group* represents the characteristics of a vulnerability that change over time but not among user environments. It captures the following metrics:

- Exploitability (E)
- Remediation Level (RL)
- Report Confidence (RC)

The *Environmental Metric Group* represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment. It includes the following metrics:

- Attack Vector (MAV)
- Attack Complexity (MAC)
- Privileges Required (MPR)
- User Interaction (MUI)
- Confidentiality Requirement (MCR)
- Integrity Requirement (MIR)
- Availability Requirement (MAR)
- Scope (MS)
- Confidentiality Impact (MC)
- Integrity Impact (MI)
- Availability Impact (MA)

A CVSS vector defines a specific set of metrics and their values, and it can be used to reproduce and assess a given score. It is rendered as a string that exactly reproduces a score.

For example, the vector `CVSS:3.1/AV:N/AC:H/PR:L/UI:R/S:C/C:H/I:L/A:N` defines a base score metric with the following parameters:

- Attack Vector: Network
- Attack Complexity: High
- Privileges Required: Low
- User Interaction: Required
- Scope: Changed
- Confidentiality Impact: High
- Integrity Impact: Low
- Availability Impact: None

In this example, a network-based attacker performs a complex attack after gaining access to some privileges, by tricking a user into performing some actions. This allows the attacker to read confidential data and change some parts of that data.

The detailed scores are the following:

Metric	Score
CVSS Base Score	6.5
Impact Sub-Score	4.7
Exploitability Sub-Score	1.3
CVSS Temporal Score	Not Available
CVSS Environmental Score	Not Available
Modified Impact Sub-Score	Not Available
Overall CVSS Score	6.5

CVSS vectors can be automatically parsed to recreate the score, for example, with the CVSS calculator provided by FIRST, the organization behind CVSS: `https://www.first.org/cvss/calculator/3.1#CVSS:3.1/AV:N/AC:H/PR:L/UI:R/S:C/C:H/I:L/A:N`.

3.2 Severity Mapping

To help in understanding the results of a test, numeric CVSS scores are mapped to qualitative ratings as follows:

Severity Rating	CVSS Score
NONE	0.0
LOW	0.1–3.9
MEDIUM	4.0–6.9
HIGH	7.0–8.9
CRITICAL	9.0–10.0

3.3 Common Weakness Enumeration

The CWE[3] is a set of software weaknesses that allows vulnerabilities and weaknesses in software to be categorized. If applicable, X41 gives a CWE ID for each vulnerability that is discovered during a test.

CWE is a very powerful method for categorizing a vulnerability. It gives general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*.[4] More information can be found on the CWE site at `https://cwe.mitre.org/`.

[3] Common Weakness Enumeration
[4] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 PS-CR-24-01: Traffic Encryption not Enforced

Severity:	MEDIUM
CVSS v3.1 Total Score:	6.8
CVSS v3.1 Vector:	CVSS:3.1/AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:H/A:H
CWE:	325 – Missing Cryptographic Step
Affected Component:	psono-server/psono/restapi/parsers.py:DecryptJSONParser()

4.1.1.1 Description

While reviewing the implementation of the transport-level encryption scheme, it was found that the ***DecryptJSONParser()*** implementation does not enforce traffic to be encrypted. Please refer to the code excerpt in listing 4.1.

```
1 class DecryptJSONParser(JSONParser):
2     """
3     Decrypts data after JSON deserialization.
4     """
```

```
5
6       media_type = 'application/json'
7       renderer_class = renderers.JSONRenderer
8
9       def parse(self, stream, media_type=None, parser_context=None):
10          """
11          Takes the incoming JSON object, and decrypts the data
12          """
13
14          data = super(DecryptJSONParser, self).parse(stream, media_type, parser_context)
15
16          if 'text' not in data or 'nonce' not in data:
17              return data
18
19          decrypted_data = decrypt(parser_context['request'].auth.secret_key, data['text'],
            ↪  data['nonce'])
20
21          try:
22              data = json.loads(decrypted_data.decode())
23          except ValueError:
24              raise ParseError('Invalid request')
25
26          return data
```

Listing 4.1: DecryptJSONParser()

The above JSON[1] parser is used by the backend server for processing incoming messages. The Psono solution makes use of a separate transport encryption layer, on top of the TLS-protected communication channel between frontend and backend. When receiving a JSON-encoded message, the backend will first check whether the message is encrypted (this is determined by inspecting the *text* and *nonce* fields of the JSON object). If the message is not encrypted, it will simply be returned as-is.

This is problematic insofar, as it allows attackers with access to the used TLS[2] communication channel to inject their own commands into a client's communication stream. Such a scenario could for instance happen in situations where the client is behind a proxy server that breaks the TLS connection, or in case an attacker obtained access to the TLS-encrypted channel by other means (e.g., by compromising the server's certificate). While this issue is not directly exploitable in the absence of other vulnerabilities, it should be noted that it bypasses Psono's additional cryptographic protection layer, and hence should be addressed in order to maintain defense in depth.

Furthermore, please observe that the same pattern is used in `psono-client/src/js/services/`

[1] JavaScript Object Notation
[2] Transport Layer Security

`user.js` in the client implementation.

4.1.1.2 Solution Advice

X41 recommends to address this issue by making encrypted communication mandatory once a session key has been negotiated between the client and the backend server. One option could be to first inspect the state of the current session, and if it turns out that the key negotiation phase already finished, to throw an exception if plain-text JSON data is received.

4.1.2 PS-CR-24-02: Parsing of PostScript Files Can Lead to Limited File System Access

Severity:	HIGH
CVSS v3.1 Total Score:	7.6
CVSS v3.1 Vector:	CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:L
CWE:	183 – Permissive List of Allowed Inputs
Affected Component:	psono-server/psono/restapi/serializers/create_avatar.py:CreateAvatarSerializer()

4.1.2.1 Description

While reviewing the implementation of the backend server, it was found that there is support for users uploading custom avatar images. Such images are however not only uploaded and stored; they are also parsed using the ***PIL*** library. Please consider the code excerpt in listing 4.2.

```
1  class CreateAvatarSerializer(serializers.Serializer):
2      data_base64 = serializers.CharField(required=True)
3
4      def validate(self, attrs: dict) -> dict:
5          data_base64 = attrs.get('data_base64', '')
6
7          try:
8              img_data = base64.b64decode(data_base64, validate=True)
9          except (base64.binascii.Error, ValueError):
10             raise exceptions.ValidationError('INVALID_BASE64')
11
12         file = io.BytesIO(img_data)
13         mime_type = None
14
15         try:
16             # Load the image and verify it's not corrupted
17             image = Image.open(file)
18             image.verify()  # Verify the image (checks integrity but not decoded)
19             file.seek(0)
20             image = Image.open(file)
21             image.load()
22             mime_type = Image.MIME.get(image.format)
```

Listing 4.2: CreateAvatarSerializer()

It can be observed that the above code fragment aims to identify the MIME[3] type of the uploaded

[3] Multipurpose Internet Mail Extensions

image. In order to do so, ***PIL*** will internally parse the provided file. The ***PIL*** library generally supports a wide variety of image formats, including possibly problematic format such as PostScript. The PostScript format is problematic insofar, as it allows for Turing-complete computations to be performed, and furthermore exposes access to the local file system of the system parsing the file. The ***PIL*** library internally uses GhostScript for parsing PostScript files. While GhostScript is invoked with the `-Dsafer` switch (which will prevent most of the problematic interfaces of the PostScript interpreter), it should be noted that this still will not prevent the PostScript interpreter from accessing files in the `/tmp` directory.

The particular behavior of the PostScript interpreter furthermore depends on the system the Psono server is running on. In case the respective system runs other services that rely on files in the `/tmp` directory, a system-level compromise might be possible. As Psono is intended to be self-hostable, the situation might be difficult to control in general.

In order to reproduce this issue, the PostScript files on `https://github.com/RedTeamPentesting/postscript_blog_examples` can be used.

4.1.2.2 Solution Advice

X41 recommends to either not parse uploaded files at all, in order to eliminate emerging server-side risks in general. An alternative option (in case parsing uploaded images is required, for instance in order to determine their MIME type), it is recommended to restrict the formats accepted by ***PIL*** to a trustworthy subset (e.g., to only accept PNG[4] and JPG files).

[4] Portable Network Graphics

4.1.3 PS-CR-24-03: Possible Confusion of Secret Data

Severity:	MEDIUM
CVSS v3.1 Total Score:	4.2
CVSS v3.1 Vector:	CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:U/C:H/I:N/A:N
CWE:	325 – Missing Cryptographic Step
Affected Component:	Symmetric Encryption Scheme

4.1.3.1 Description

A rather general pattern identified during the review of the used encryption scheme is that secrets (such as passwords or other confidential information) are first encrypted by the client using a randomly-generated key. This key is then stored in a data store, which is in turn again encrypted using a symmetric key derived from the user's password.

This is not generally a weakness; in fact, it is rather recommendable to use separate cryptographic keys for storing individual objects. However, when using such a scheme, it is important to keep in mind that in the presence of a rogue server, clients should still be able to double-check whether an encrypted secret they request from the server actually matches the data they stored.

To illustrate this concern, please consider the following example scenario: A client stores a secret on the server, and the server assigns a specific UUID[5] to that secret. The secret could for instance be a password required to upload a build artifact to an FTP[6] server from within a CI/CD[7] pipeline. Later on, the server's API[8] is used to access this secret again from the CI/CD pipeline. For the sake of simplicity, assume that a restricted API key is used for this purpose. When receiving the API request for accessing the secret, a rogue server could now swap the encrypted secret with another, more sensitive secret that is tied to the respective API key. This would result in this sensitive password to be leaked when initiating a plaintext FTP connection.

Please be aware that this is a general concern, which is not exclusively tied to the scenario above. Generally speaking, clients should not only be able to verify that data received from the server has not been directly tampered with (Psono ensures this by using authenticated encryption), but also that the received data actually matches what they requested.

While reviewing the implementation, no such mechanism could be identified. A rogue server could - for instance in the above example scenario - therefore swap encrypted secrets and possibly trigger a disclosure of their plaintext versions. Pinpointing this issue to a particular piece of

[5] Universally Unique Identifier
[6] File Transfer Protocol
[7] Continous Integration/Continous Delivery
[8] Application Programming Interface

code is not directly possible, as it is rather referring to the general practice used in the analyzed encryption scheme.

4.1.3.2 Solution Advice

Addressing this issue in general likely requires adjusting the used cryptographic scheme. One way of approaching the problem could be do make use of the fact that the ***NaCL*** library that Psono uses for encryption offers not only authenticated encryption, but particularly also an AEAD[9] mode. That is, it allows for additional plaintext data to become part of the authentication tag that is generated during the encryption process. This ties the respective ciphertext to the used additional data. Therefore, one could switch to an AEAD cipher and include the ID[10] of the respective secret in the additional data.

[9] Authenticated Encryption with Associated Data
[10] Identifier

4.1.4 PS-CR-24-04: Cryptographic Key Disclosure to Rogue Server

Severity:	MEDIUM
CVSS v3.1 Total Score:	5.7
CVSS v3.1 Vector:	CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:U/C:H/I:H/A:N
CWE:	325 – Missing Cryptographic Step
Affected Component:	psono-server/psono/restapi/views/login.py:LoginView()

4.1.4.1 Description

While reviewing the implementation of the login functionality and of the cryptographic key derivation in general, it was found that a rogue Psono server could initiate an attack against clients, by which it would trick a client into using encryption keys that are known to the server.

In a first step, please observe that a user's password is used for both, authentication with the server and for encrypting the user's secrets. For authentication, the password is subject to the ***scrypt()*** function. The required salt for hashing the password is the SHA-256[11] hash of the username. This is also called the user's *AUTHKEY*. Please observe that during each login, the client has to send the *AUTHKEY* to the server, which - if it was acting maliciously - could store this value.

For performing symmetric encryption, a client would then derive a different key from the password, by using the same procedure as described above. However, instead of using the hash of the username as salt, the client would this time use another parameter, called *user_sauce*. Needless to say, the result of this operation is kept secret by the client and is not sent to the server.

Given the fact that the Psono solution allows for multiple clients to use the same account, the server will supply most required information to each client upon login. This information also includes the *user_sauce* parameter, as highlighted in the below code excerpt:

```
1  class LoginView(GenericAPIView):
2      permission_classes = (AllowAny,)
3      serializer_class = LoginSerializer
4      allowed_methods = ('POST', 'OPTIONS', 'HEAD')
5      throttle_scope = 'login'
6
7      def get(self, *args, **kwargs):
8          return Response({}, status=status.HTTP_405_METHOD_NOT_ALLOWED)
9
10     def put(self, *args, **kwargs):
11         return Response({}, status=status.HTTP_405_METHOD_NOT_ALLOWED)
12
```

[11] Secure Hashing Algorithm 2, 256-bit

```
13      def post(self, request, *args, **kwargs):
14  [...]
15          response = {
16              "token": token.clear_text_key,
17              "session_valid_till": token.valid_till.isoformat(),
18              "required_multifactors": required_multifactors,
19              "session_public_key": server_session_public_key_hex.decode('utf-8'),
20              "session_secret_key": session_secret_key_hex.decode('utf-8'),
21              "session_secret_key_nonce": session_secret_key_nonce_hex.decode('utf-8'),
22              "user_validator": user_validator_hex.decode('utf-8'),
23              "user_validator_nonce": user_validator_nonce_hex.decode('utf-8'),
24              "user": {
25                  "username": user.username,
26                  "language": user.language,
27                  "public_key": user.public_key,
28                  "private_key": user.private_key,
29                  "private_key_nonce": user.private_key_nonce,
30                  "user_sauce": user.user_sauce,
31                  "authentication": user.authentication,
32                  'hashing_algorithm': user.hashing_algorithm,
33                  'hashing_parameters': user.hashing_parameters,
34              }
35          }
```

Listing 4.3: LoginView()

A malicious server could therefore send the SHA-256 hash of the user's username as their *user_sauce*. This would trick a client into essentially deriving the *AUTHKEY* value again, which it would then use for symmetric encryption. As outlined above, the *AUTHKEY* is however in principle known to the server. The server would therefore possess the key material required to decrypt a user's secrets.

Please observe that this attack is most effective against users who just created a new account with Psono. This is because replacing *user_sauce* with another value will render all stored encrypted information unreadable to the client, and hence clients might detect that an attacck is going on. However, please also note that it might be possible to conceal this attack, for instance by notifying the respective user about „Technical Difficulties" and ask then to export and reimport all stored data.

4.1.4.2 Solution Advice

Ideally, the *user_sauce* parameter should not be sent to the clients by the server. Instead, clients should store this value locally. However, X41 recognizes that this might be difficult to achieve, as it would require users to type in the *user_sauce* during the setup of the application on a

new device. It might therefore be more applicable to hard-code *user_sauce* to a static value that cannot coincide with the username. X41 recognizes that this will reduce the entropy of the derived key material down to the entropy of the user's password. However, without making the user remember a source of more entropy (such as a random *user_sauce*), increasing the entropy of the derived key material does not seem to be easily possible.

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 PS-CR-24-100: Missing Domain Separation

Affected Component:	Traffic Encryption

4.2.1.1 Description

While reviewing the overall implementation of the traffic encryption scheme, it was found that the communication between clients and the server is protected by symmetric encryption. However, both directions (i.e., from client to server and from server to client) currently use the same cryptographic key. This means that on a cryptographic level, it is currently not possible to distinguish between a message sent by a client, and a message sent by the server. While this might appear to be a benign issue, it should be noted that it allows for so-called reflection attacks, where an attacker would record a message, say sent by a client, and then later play back that very message to the client again. Depending on the implementation, the client might confuse this message with one that originates from the servers.

In the current implementation, this has not found to be exploitable. However, it is not considered best practice and might lead to exploitable situations in the future (e.g., when additional features and new message types are introduced).

4.2.1.2 Solution Advice

X41 recommends to use different cryptographic keys for each direction (from/to server/client). Such keys can be generated for instance by making use of a key derivation function such as ***hkdf()***

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

AEAD Authenticated Encryption with Associated Data 19
API Application Programming Interface 18
CI/CD Continous Integration/Continous Delivery 18
CVSS Common Vulnerability Scoring System 9
CWE Common Weakness Enumeration 12
FTP File Transfer Protocol 18
ID Identifier 19
JSON JavaScript Object Notation 14
MIME Multipurpose Internet Mail Extensions 16
PNG Portable Network Graphics 17
SHA-256 Secure Hashing Algorithm 2, 256-bit 20
TLS Transport Layer Security 14
UUID Universally Unique Identifier 18