Security assessments for software that matters.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report ERA Wallet mobile apps, crypto, API & TX flows 08.2026

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, C. Lüders, Dr. D. Bleichenbacher, M. V. S. Lima

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

"At ERA, we believe real security starts with understanding. That's why our products are built to be open, verifiable, and intuitive - combining strong cryptography with human-centered design. We don't hide complexity behind marketing - we make it understandable. Because only when you know how your wallet works, you can truly trust it."

From https://era-wallet.com/about-era

This report describes the results of a penetration test and source code audit conducted against the ERA wallet mobile applications and Serverpod RPC API.

To give some context regarding the assignment's origination and composition, ERA Wallet (HWLT FZE) contacted Cure53 in June 2026. The test execution was scheduled for August 2026, namely in CW32. A total of eighteen days were invested to reach the coverage expected for this project, and a team of five senior testers was assigned to its preparation, execution, and finalization.

The methodology conformed to a white-box strategy, whereby assistive materials such as sources, builds, documentation, as well as all further means of access required to complete the tests were provided to facilitate the undertakings.

The work was split into four separate work packages (WPs), defined as:

- **WP1**: White-box pen.-tests & code audits against ERA mobile applications
- **WP2**: White-box pen.-tests & code audits against ERA Transaction Construction
- **WP3**: White-box pen.-tests & code audits against ERA QR & BC-UR flows
- **WP4**: White-box pen.-tests & code audits against ERA Serverpod RPC API

All preparations were completed in late July 2026, specifically during CW31, to ensure a smooth start for Cure53. Communication throughout the test was conducted through a dedicated and shared Slack channel, established to combine the teams of ERA and Cure53. All personnel involved from both parties were invited to participate in this channel. Communications were smooth, with few questions requiring clarification, and the scope was well-prepared and clear. No significant roadblocks were encountered during the test. Cure53 provided frequent status updates, shared its findings, and provided live reporting for all findings through the aforementioned Slack channel.

The Cure53 team achieved good coverage over the scope items, and identified a total of twenty-eight findings. Of the twenty-eight security-related findings, fifteen were classified as security vulnerabilities, and thirteen were categorized as general weaknesses with lower exploitation potential.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It should be noted that a dedicated hardware wallet is required in order to perform real signing operations; however, Cure53 was only made aware of this fact at the start of the engagement, which left no time to ship the devices. Instead, Cure53 was provided with a simulator to work with. This did not allow the exploration of all features in a real, production-grade setting.

During the engagement, the mobile applications initially exhibited a moderate security posture, though systemic flaws increasingly surfaced as testing progressed. The plaintext storage of sensitive information was discovered within local data files (HWL-01-021), which was accompanied by trivial bypasses of the PIN lock mechanism (HWL-01-027) and platform-specific biometric API controls (HWL-01-028). While a comparatively positive impression was made by the Android application, it is advised that the overall mobile defense-in-depth posture was found to be inadequate.

It is advised that a brittle state of security was observed when testing the backend API, with pervasive authentication and authorization omissions being discovered. Critical endpoints were found to leak user data to unauthenticated entities (HWL-01-007), allow unauthorized account linking via public fingerprints (HWL-01-009), and permit complete backend Denial of Service (DoS) (HWL-01-010). Furthermore, vital authorization checks were observed to be rendered ineffective in production environments (HWL-01-011), while unauthorized transaction broadcasting was permitted through the proxy service (HWL-01-018).

The report will now shed more light on the scope and testing setup, and will provide a comprehensive breakdown of the available materials. Following this, the report will list all findings identified in chronological order, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues* unearthed. Each finding will be accompanied by a technical description, Proof-of-Concepts (PoCs) where applicable, and any fix or preventative advice to action.

In summation, the report will finalize with a *Conclusions* chapter in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the ERA wallet mobile applications and Serverpod RPC API.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pen.-tests & code audits against ERA Wallet mobile apps, crypto & TX flows**
  - **WP1:** White-box pen.-tests & code audits against ERA mobile applications
    - **Repository:**
      - URL: https://github.com/ERAWLT/ERA-SW
      - Branch: develop
      - Commit: a7aac0160d926b2ff4a66b9a72c01abe2e30b747
    - **Android Build:**
      - https://play.google.com/apps/testing/io.hwlt.era_sw
    - **iOS Build:**
      - https://testflight.apple.com/join/72m8ywSu
  - **WP2:** White-box pen.-tests & code audits against ERA Transaction Construction
    - **Repository:**
      - URL: https://github.com/ERAWLT/ERA-SW
      - Branch: develop
      - Commit: a7aac0160d926b2ff4a66b9a72c01abe2e30b747
  - **WP3:** White-box pen.-tests & code audits against ERA QR & BC-UR flows
    - **Repository:**
      - URL: https://github.com/ERAWLT/ERA-SW
      - Branch: develop
      - Commit: a7aac0160d926b2ff4a66b9a72c01abe2e30b747
  - **WP4:** White-box pen.-tests & code audits against ERA Serverpod RPC API
    - **Environment URL:**
      - https://api.era-wallet.com
    - **Repository:**
      - URL: https://github.com/ERAWLT/appserv
      - Branch: master
      - Commit: 3f07dbe2e252f6f95599ff579c3e90fdf3ba56bd
  - **Test User Credentials**
    - Via QR code file
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., HWL-01-001) to facilitate any future follow-up correspondence.

HWL-01-002 WP1: DoS of Android app via serialized intents *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While dynamically investigating the ERA Wallet Android app, it was noted that the app exports only a single activity - namely its main activity. Fuzzing this activity for security issues revealed that the ERA Wallet app crashes when sending an intent to start the main activity with unknown extra data.

This puts the ERA Wallet app at risk of crashes if an attacker manages to lure a victim to install a malicious app on their Android device. In particular, if the victim first launches the ERA Wallet app and consequently opens the attacker's app, then the attacker's app could send an intent to crash the ERA Wallet app.

It should be noted that Android apps were able to send intents while being backgrounded up to Android SDK level 28. This circumstance therefore applies to the ERA Wallet app, as it supports devices with Android SDK level 26 (see HWL-01-001).

Steps to reproduce:

1. Connect a mobile device that has the Android ERA Wallet app installed.
2. Start the ERA Wallet app, and authenticate using the configured PIN.
3. Put the ERA Wallet app to the background.
4. Open Android Studio, and open the Logcat output.
5. Start the IntentFuzzer app[1] on the mobile device, and select the ERA Wallet app.
6. Click *Serialize Fuzz All*, which results in a crash of the ERA Wallet app. The crash can be observed in the Logcat output, as shown below:

Logcat output:

```
2026-08-03 19:00:17.658  9337-9337  AndroidRuntime
io.hwlt.era_sw                        D  Shutting down VM
2026-08-03 19:00:17.658  9337-9337  AndroidRuntime
io.hwlt.era_sw                        E  FATAL EXCEPTION: main
Process: io.hwlt.era_sw, PID: 9337
```

[1] https://github.com/MindMac/IntentFuzzer

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
java.lang.RuntimeException: Unable to start activity
ComponentInfo{io.hwlt.era_sw/io.hwlt.era_sw.MainActivity}:
java.lang.RuntimeException: Parcelable encountered
ClassNotFoundException reading a Serializable object (name =
com.android.intentfuzzer.util.SerializableTest)
at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:
3764)
```

To mitigate this issue, Cure53 recommends implementing robust data input validations for all exported activities, in order to prevent unexpected crashes.

HWL-01-007 WP4: Unauthenticated endpoints leak user data *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The ERA Wallet API implements authentication in terms of HMAC verifications. To this end, a user's mobile device utilizes an installation-specific secret in each of its requests, to authenticate the user on the backend. A static code review revealed that several of the API endpoints fail to enforce such a HMAC check, instead having it optionally executed only if the *hmac* field is present in the request. This results in a data leakage to unauthenticated attackers.

Several endpoints are affected by this vulnerability. Some of these reveal user-specific information concerning Onramp and MoonPay transactions - such as fiat amount and currency, payment methods, wallet addresses, and failed or abandoned transactions. These endpoints require the knowledge of the victim's device ID. In contrast, the *swap* endpoint, together with the *getTransactions* function, require only addresses for the victim, and reveal not only all recorded swap attempts for the provided addresses, but also the associated device ID. It should be noted that addresses correspond to publicly available information. The attacker can consequently use this device ID to retrieve other data for the victim, as described above. In summary, the attacker can acquire transaction information concerning the victim, together with knowledge about which wallet addresses belong to them.

The request shown below demonstrates the leakage for the *getTransactions* function of the *swap* endpoint. It should be noted that the request's *hmac* field is set to *null*. The response reveals the entire swap, together with its *deviceId*:

Request:

```
POST /api/swap HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 105
host: api.era-wallet.com
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
Connection: keep-alive

{"hmac":null,"fromAddresses":
["GjJyeC1r2RgkuoCWMyPYkCWSGSGLcz266EaAkLA27AhL"],"method":"getTransactions"
}
```

Response:

```
HTTP/1.1 200 OK
[...]

[{"__className__":"SwapTransaction","id":224,"internalSwapId":"-
YilJMJ2TZV8M8VGrVLBOQ","deviceId":"era-sw-msdgv0h3-
5fbe40c87580c9a271d0b36fc0edc838","providerId":"rango","providerRequestId":
"10c1a82a-b961-41e5-bb6e-
31a6e44390d5","providerContext":"[...]}","fromChain":"SOLANA","fromToken":"
SOL","fromAmount":"55134468","fromAddress":"GjJyeC1r2RgkuoCWMyPYkCWSGSGLcz2
66EaAkLA27AhL","toChain":"ZKSYNC","toToken":"ETH","toAddress":"0x9858effd23
2b4033e47d90003d41ec34ecaeda94","expectedOutputAmount":"2157586400520002","
status":"awaiting_swap","createdAt":"2026-08-
03T18:56:05.650455Z","updatedAt":"2026-08-03T18:56:08.578954Z"},,[...]]
```

The list below summarizes all endpoint files that leak user-related information to an unauthenticated attacker. Cure53 stresses that other endpoints also exist which do not enforce the HMAC check, and which do not leak user-related information. However, it is advised that these endpoints should still only be accessible to users of the ERA Wallet app.

Affected files:

- *appserv/era_server/lib/src/endpoints/onramp_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/onramper_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/moonpay_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/swap_endpoint.dart*

The excerpt below demonstrates the issue for the *getTransactions* function of the *swap* endpoint. It is clear that no HMAC check is enforced:

Affected code (*swap_endpoint.dart*):

```
Future<List<SwapTransaction>> getTransactions(
  Session session,
  AuthEnvelope? hmac,
  List<String> fromAddresses,
) async {
  if (hmac != null) {
     await HmacVerifier.verify(session, hmac, 'swap.getTransactions', const
[]);
  }
  [...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
return SwapTransaction.db.find(
  session,
  where: (t) => t.fromAddress.inSet(fromAddresses.toSet()),
  orderBy: (t) => t.createdAt,
  orderDescending: true,
);
```

It is strongly recommended to revisit all endpoints that do not have the HMAC check enforced, and to ensure that all such endpoints require a valid HMAC. Furthermore, it is advised that the handlers should implement a check to prevent unauthorized access of data, by cross-checking if the sending installation has access to the data referenced by the request.

HWL-01-009 WP4: Unauthorized device linking to account via fingerprint *(High)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

When linking a device to a wallet, the ERA Wallet mobile app invokes the *registerAccount* function of the *wallet* API endpoint. The payload of this request contains several parameters, including the *walletFingerprint* and *accountIndex*. The endpoint handler checks if the database already contains a wallet account for the provided fingerprint and account index (which is unique), and if so, it links the installation ID, corresponding to the device ID, to the wallet account. This grants read-only access to the wallet account. It was discovered that solely the knowledge of the fingerprint, which is public information (see below), suffices to link a new device to a wallet account.

This enables an attacker who manages to get possession of the fingerprint for a victim's wallet account to probe for utilized *accountIndex* values used by the victim. If the attacker manages to find this *accountIndex*, then the API automatically links the attacker's device to the victim's wallet account. Therefore, the attacker gains read-only access to all of the victim's wallets without the victim needing to confirm this new device. Due to this read-only access, the attacker will acquire information about all past transactions for the wallet account, and will also receive notifications on all new transactions and live updates.

It was confirmed by the ERA Wallet development team that the fingerprint is public information, as it is used as customer ID by payment providers within the checkout link in the browser. Hence this information can not be considered secret.

The excerpt below demonstrates the issue. It is clear that the *registerAccount* function checks if the wallet account already exists, and if so, it automatically links the new device to the wallet account:

Affected file:
appserv/era_server/lib/src/endpoints/wallet_endpoint.dart

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
Future<WalletRegisterResponse> registerAccount(
  Session session,
  AuthEnvelope hmac,
  WalletRegisterRequest request,
) async {
  await HmacVerifier.verify(
    session,
    hmac,
    'wallet.registerAccount',
    utf8.encode(request.toString()),
  );
  [...]
  final existing = await WalletAccount.db.findFirstRow(
    session,
    where: (a) =>
        a.walletFingerprint.equals(fingerprint) &
        a.accountIndex.equals(request.accountIndex),
  );

  if (existing != null) {
    [...]
    final updated = existing.copyWith(
      ethAddress: _keepIfBlank(request.addresses.eth, existing.ethAddress),
      [...]
    );
    [...]
    await WalletAccount.db.updateRow(
      session,
      updated,
      columns: (t) => [
        t.ethAddress,
        t.btcAddress,
        t.btcTestnetAddress,
        t.btcXpub,
        t.btcTestnetXpub,
        t.btcAddresses,
        t.btcTestnetAddresses,
        t.solAddress,
        t.tronAddress,
      ],
    );

    // Authorize THIS device for the shared account (idempotent).
    await AccountAccess.linkDevice(session, existing.id!, hmac.installId);
    [...]
  }
  [...] }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It is strongly recommended to implement a confirmation step involving the owner of the wallet account when adding a new device to an existing wallet account. In a first step, the backend should notify the owner on their linked devices that a new registration has been attempted, and the owner of the wallet account can then either complete or deny the new registration. This will ensure that the owner retains full control over the wallet account.

HWL-01-010 WP4: Unauthenticated memory exhaustion DoS of API *(High)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While following up on HWL-01-007 (which documents that several endpoint handlers fail to enforce authentication via a HMAC check), Cure53 investigated all endpoints that did not require authentication. This led to the discovery of the */api/swap* route for the *getSupported* function. The corresponding endpoint handler returns a list of all supported chains, together with their assets, without pagination. The response is roughly 5.4MB large - which is sufficiently long to mount a DoS attack due to memory / resource exhaustion in the given rate-limiting setting.

In particular, an unauthenticated attacker could send hundreds of requests in parallel to the */api/swap* endpoint for the *getSupported* function. For example, 100 requests in parallel amount to roughly 540MB, which is feasible using several public IP addresses in the given rate-limit setting. This will eventually consume all of the application's memory within the container, and will result in a crash.

Due to the fact that testing was conducted against a production instance, the issue was dynamically investigated by the customer. These dynamic tests confirmed that load increases drastically when sending multiple such requests in parallel, effectively leading to a backend crash.

The request shown below can be used to demonstrate the issue. It queries for all supported chains for performing swaps without authentication. The response, as highlighted, is roughly 5.4MB long:

Request:

```
POST /api/swap HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 25
host: api.era-wallet.com
Connection: keep-alive

{"method":"getSupported"}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.27.3
Date: Tue, 04 Aug 2026 06:32:29 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 5331931
Connection: keep-alive
access-control-allow-origin: *
access-control-allow-methods: POST

{"__className__":"SwapSupportedResponse","chains":[...]}
```

From the excerpt below, it is evident that the *getSupported* handler fails to support pagination. Instead, the handler keeps all results in memory:

Affected file:
appserv/era_server/lib/src/endpoints/swap_endpoint.dart

Affected code:

```
Future<SwapSupportedResponse> getSupported(
  Session session,
  AuthEnvelope? hmac, {
  String? chainFilter,
}) async {
  if (hmac != null) {
    await HmacVerifier.verify(session, hmac, 'swap.getSupported', const
[]);
  }
  [...]
  for (final p in providers) {
    providerIds.add(p.providerId);
    [...]
    chains.addAll(resp.chains);
    for (final a in resp.assets) {
      final key = '${a.chain}:${(a.address ?? '').toLowerCase()}:$
{a.symbol}';
      assets.putIfAbsent(key, () => a);
    }
  }
  [...]
  final refined = SwapAssetRanking.refine(
    filteredAssets,
    canonicalSet: canonicalSet,
    catalog: catalog,
    rankByCoingeckoId: rankByCoingeckoId,
  );
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  return SwapSupportedResponse(
    chains: chains
        .where((c) => filter == null || c.toUpperCase() == filter)
        .toList(),
    assets: refined,
    providerIds: providerIds,
  );
}
```

To mitigate this issue, Cure53 strongly recommends implementing a pagination mechanism for all endpoints querying for data. It is advised that the parameters for pagination should also be constrained in such a way that the application will not load excessive amounts of data in memory. Furthermore, this approach should be paired with strict input data validation with regard to the length of strings and fields when creating new entries in the database.

HWL-01-011 WP4: Unauthenticated device attestation API due to asserts *(High)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Reviewing the source code of the *appserv* repository revealed that the endpoint handlers for device attestation utilize the Dart *assert* statement when performing security-relevant checks. It should be noted that *assert* statements will not be included in production builds. This leaves all endpoints of the device attestation endpoint unprotected.

This enables an attacker to execute all endpoints of the device attestation portion of the ERA Wallet API. For example, as also demonstrated below, an attacker could query for all devices through the *getDevices* function. Or the attacker could also add devices to the database. Lastly, the attacker could also tamper with the *verifyResponse* function, effectively bypassing the security checks for finding the correct device attestation data in the database.

The request shown below highlights the issue. It executes the *getDevices* function on the *deviceAttestation* endpoint. The *serverSecret* parameter is required for processing the request correctly, however, it can be left blank:

Request:

```
POST /api/deviceAttestation HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 41
host: api.era-wallet.com
Connection: keep-alive

{"serverSecret":"","method":"getDevices"}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Response:

```
HTTP/1.1 200 OK
[...]

[{"__className__":"Device","id":4,"deviceId":"HWABA01001768040","publicKey"
:"2D208DBB6D1BE9CB0CC5E5DC94391E1340220DEAC2EF6599BC281B2B2D27885466729062E
5C308C9EFBCBAB19BAC7859EB9EFEFBB0468AE7A9FC8C40EC9F0475"},[...]]
```

The excerpt below demonstrates the issue in the *DeviceAttestationEndpoint* class. It uses the *assert* statement several times to perform security-relevant checks. For example, the *addDevices* function authenticates the caller through the knowledge of the *serverSecret* parameter, using an *assert* statement:

Affected file:
appserv/era_server/lib/src/endpoints/device_attestation_endpoint.dart

Affected code:

```
class DeviceAttestationEndpoint extends Endpoint {
  [...]
  Future<bool> verifyResponse(
    Session session,
    DeviceAttestation attestation,
  ) async {
    assert(attestation.deviceId != null && attestation.signature != null);
    [...]
    assert(attestationDb != null &&
        attestationDb.challenge == attestation.challenge &&
        attestationDb.timestamp == attestation.timestamp &&
        attestationDb.sid == attestation.sid);
    [...]
  }

  Future<void> addDevices(Session session, AddDevicesRequest request) async
{
    assert(request.serverSecret ==
        Serverpod.instance.getPassword("serviceSecret"));
    [...]
  }

  Future<List<Device>> getDevices(Session session, String serverSecret)
async {
    assert(serverSecret ==
Serverpod.instance.getPassword("serviceSecret"));
    return await Device.db.find(session);
  }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  Future<void> updateDevicesFromDb(Session session, String serverSecret)
async {
    assert(serverSecret ==
Serverpod.instance.getPassword("serviceSecret"));
    await session.serverpod.futureCallWithDelay('fetchAttestationDbCall',
        FetchAttestationDbCallParams(isSkipPlanning: true), Duration.zero);
  }
}
```

To mitigate this issue, Cure53 recommends removing all *assert* statements from the server source code repository, and replacing them with conditional statements (e.g., *if* … *throw*), which return an error if the condition is not met.

HWL-01-014 WP4: IP spoofing via *X-Forwarded-For* header *(Info)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Validation of the code responsible for determining a client's IP address confirmed that, when present, the first value contained in the *X-Forwarded-For* header is treated as the originating client IP. Furthermore, reviewing the Nginx configuration revealed that the *X-Forwarded-For* header is populated using the *$proxy_add_x_forwarded_for* variable, which appends the connecting client's IP address to any existing header value, rather than overwriting it. As a result, an attacker can supply an arbitrary *X-Forwarded-For* header, causing the application to trust a spoofed client IP address.

The request shown below demonstrates the issue by detecting the country of the IP address from the *X-Forwarded-For* HTTP header. From the response, it is clear that the spoofing is successful, as the request was not sent from Australia.

Request:

```
POST /api/onramper HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 38
X-Forwarded-For: 1.1.1.1
host: api.era-wallet.com
Connection: keep-alive

{"hmac":null,"method":"detectCountry"}
```

Response:

```
HTTP/1.1 200 OK
[...]
"AU"
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The excerpt below highlights the issue. Both files contain this identical code snippet, which extracts the client's IP address from the *x-forwarded-for* HTTP header:

Affected files #1:

- *appserv/era_server/lib/src/endpoints/onramper_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/moonpay_endpoint.dart*

Affected code #1 :

```
String? _clientIp(Session session) {
  final headers = session.request?.headers;
  if (headers == null) return null;

  String? candidate;
  final forwarded = headers['x-forwarded-for']?.firstOrNull;
  if (forwarded != null && forwarded.isNotEmpty) {
    candidate = forwarded.split(',').first.trim();
  }
  candidate ??= headers['x-real-ip']?.firstOrNull?.trim();

  if (candidate == null || candidate.isEmpty) return null;
  if (_isLoopbackOrPrivate(candidate)) return null;
  return candidate;
}
```

The excerpt below shows the Nginx configuration:

Affected file #2:
appserv/nginx.conf

Affected code #2:

```
server {
    listen 443 ssl;
    server_name api.era-wallet.com;

    [...]

    location /api/ {
        proxy_pass http://app:8080/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    [...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 recommends configuring the reverse proxy to overwrite any client-supplied *X-Forwarded-For* header, rather than appending to an existing value.

HWL-01-015 WP4: DoS in anonymous rate-limited endpoints *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During the assessment of the application's rate-limiting implementation, it was observed that the HMAC value, the *installId*, or the global *<anon>* identifier were used as the rate-limiting key. However, several endpoints intentionally accept requests with a null HMAC value. As a result, requests targeting these endpoints are associated with the shared *<anon>* identifier, thereby allowing an attacker to exhaust the corresponding rate-limit budget and effectively deny service to all anonymous users.

It should be noted that this behavior affects every rate-limited endpoint that accepts requests with a *null* HMAC value. However, the practical impact is limited to endpoints that are explicitly intended to process unauthenticated requests, as these can be accessed without prior user interaction. Consequently, an attacker can repeatedly invoke such endpoints to exhaust the shared anonymous rate-limit budget, thereby affecting all anonymous users.

Request:

```
POST /api/onramp/getConfig HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 13
host: api.era-wallet.com
Connection: keep-alive

{"hmac":null}
```

Response:

```
HTTP/1.1 400 Bad Request
[...]

{"className":"ServerException","data":
{"__className__":"ServerException","message":"Rate limit exceeded. Please
retry shortly.","errorCode":429}}
```

The excerpt below shows the shared rate-limiting key used by all anonymous users. It should be noted that all unauthenticated requests for an endpoint will share this bucket:

Affected file #1:
appserv/era_server/lib/src/services/rate_limiter.dart

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code #1:

```
/// Key used when a nullable-`hmac` method is called without an envelope:
all
/// unauthenticated callers share one coarse bucket (a behind-nginx setup
/// collapses remote IPs anyway, so a global anon bucket is equivalent).
static const anon = '<anon>';
```

The excerpt below demonstrates the issue for the Onramp endpoints:

Affected file #2:
appserv/era_server/lib/src/endpoints/onramp_endpoint.dart

Affected code #2:

```
Future<OnrampConfig> getConfig(
  Session session,
  AuthEnvelope? hmac,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<List<OnrampPurchase>> getHistory(
  Session session,
  AuthEnvelope? hmac,
  String deviceId,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]
```

The excerpt below demonstrates the issue for the MoonPay endpoints:

Affected file #3:
appserv/era_server/lib/src/endpoints/moonpay_endpoint.dart

Affected code #3:

```
Future<MoonPaySupportedResponse> getSupported(
  Session session,
  AuthEnvelope? hmac,
  String? country,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<List<MoonPayCountry>> getCountries(
  Session session,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  AuthEnvelope? hmac,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]
Future<List<MoonPayPaymentMethod>> getPaymentMethods(
  [...]
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<String?> detectCountry(
  Session session,
  AuthEnvelope? hmac,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<MoonPayQuote> getQuote(
  [...]
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<List<MoonPayTransaction>> getTransactions(
  Session session,
  AuthEnvelope? hmac,
  String deviceId,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]

Future<MoonPayTransaction?> getTransaction(
  Session session,
  AuthEnvelope? hmac,
  String externalTransactionId,
) async {
  RateLimiter.instance
      .check(hmac?.installId ?? RateLimiter.anon, RateLimiter.onramp);
  [...]
```

Cure53 recommends deriving rate-limiting keys for anonymous requests from client-specific attributes, rather than using a single global identifier. Suitable identifiers include the originating IP address, or other values that uniquely distinguish individual clients.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-016 OOS: Open redirect in */tap/return* endpoint *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

A cursory review of the *appserv* repository revealed that the */tap/return* endpoint of the payment service suffers from an open redirect vulnerability. The endpoint allows for arbitrary *return_url* parameters in its query, which could aid an attacker in phishing attacks. If an attacker managed to convince a victim to open a specially crafted link, containing a malicious URL in the *return_url* field, pointing to the payment service of the ERA Wallet, then the ERA Wallet payment service would in turn redirect the victim to the attacker-controlled URL. The attacker could use this in attempts to mount phishing attacks.

The URL shown below demonstrates the issue. It corresponds to a URL for the ERA Wallet payment service that immediately redirects to *https://cure53.de*:

URL:
https://pay.era-wallet.com/tap/return?return_url=https%3A%2F%2Fcure53.de%23

The excerpt below demonstrates the issue. It is evident that the payment service fails to restrict the *return_url* value to URLs of the *pay.era-wallet.com* host:

Affected file:
appserv/era_payments/server.js

Affected code:

```
app.get('/tap/return', async (req, res) => {
  [...]
  try {
    [...]
    const redirectUrl = paymentStatus === 'PAID' ? return_url : `${decodeURIComponent(return_url + '&errorMsg=An error occurred while making the payment. Please try again.')}`;


    res.redirect(redirectUrl);
  } catch (err) {
    [...]
  }
});
```

To mitigate this issue, Cure53 recommends parsing the supplied *return_url*, and only allowing URLs pointing to an expected host (such as *pay.era-wallet.com*) together with a set of allow-listed paths.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-017 WP4: Insecure randomness for generating device attestation *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While reviewing the device attestation API, it was discovered that the endpoint to generate the device attestation values uses cryptographically insecure randomness for generating the attestation's challenge and *sid*. The challenge value should correspond to a single-use, non-guessable cryptographically secure random value. However, the backend currently uses Dart's *Random* class. This class internally makes use of a pseudorandom number generator, and hence, is not suitable for cryptographic purposes.

The excerpt below shows the *AttestationUtil* class. The *createDeviceAttestation* function uses the *CryptoUtil.generateRandomString* function to generate the *sid* and *challenge* values.

Affected file #1:
appserv/era_server/lib/src/attestation_util.dart

Affected code #1:
```
abstract final class AttestationUtil {
  static DeviceAttestation createDeviceAttestation(Session session) {
    return DeviceAttestation(
      sessionId: session.sessionId.uuid,
      sid: CryptoUtil.generateRandomString(4),
      challenge: CryptoUtil.generateRandomString(32),
      timestamp: DateTime.now(),
    );
  }
}
```

The excerpt below demonstrates the usage of the pseudorandom number generator *Random* to return a random string:

Affected file #2:
appserv/era_server/lib/src/crypto_util.dart

Affected code #2:
```
import 'dart:math';
[...]
abstract final class CryptoUtil {
  static String generateRandomString(int length) {
    const chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
    Random rand = Random();
    return List.generate(length, (i) => chars[rand.nextInt(chars.length)])
        .join();
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  }
  [...]
```

To mitigate this issue, Cure53 recommends instead using the *Random.secure()* function to create a cryptographically secure random number generator. This will eliminate the risk of a predictable challenge when performing device attestation.

HWL-01-018 WP4: TRON RPC proxy allows unauthenticated broadcast *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Testing confirmed that the TRON RPC proxy exposes a broad range of endpoints - including transaction creation and broadcast functionality - for forwarding requests to the underlying TRON relay. Since this endpoint is affected by the authentication issue described in HWL-01-007, which renders HMAC authentication effectively optional, an unauthenticated attacker may abuse ERA's paid TRON RPC subscription, by proxying arbitrary requests - including processing their own transactions - through the service.

Affected file:
appserv/era_server/lib/src/endpoints/rpc_proxy_endpoint.dart

Affected code:

```
/// Read-only RPC proxy. Whitelisted methods only.
class RpcProxyEndpoint extends Endpoint {

[...]

 Future<String> proxyTronRpc(
    [...]
  ) async {
    RateLimiter.instance.check(_rateKey(hmac), RateLimiter.rpc);
    if (hmac != null) {
      await HmacVerifier.verify(session, hmac, 'rpcProxy.proxyTronRpc',
utf8.encode('$path:$requestBody'));
    }
    const allowedPaths = [
      '/wallet/createtransaction',
      '/wallet/broadcasthex',
      '/wallet/broadcasttransaction',
      '/wallet/getnowblock',
      '/wallet/getaccount',
      '/wallet/getchainparameters',
      '/wallet/getaccountresource',
      '/wallet/triggerconstantcontract',
      '/wallet/triggersmartcontract',
      '/wallet/transferasset',
    ];
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    if (!allowedPaths.any((p) => path.toLowerCase() == p)) {
      throw ServerException(message: 'Tron RPC path not allowed: $path',
errorCode: 403);
    }
    [...]
    return response.body;
  }
}
```

Cure53 recommends primarily addressing the authentication issue affecting this route, so as to prevent unauthenticated access. In addition, it is advised that appropriate rate-limiting should be enforced to mitigate unrestricted use of the exposed endpoints. Finally, it is advised that the allow-list of proxied paths should be reviewed and restricted to the intended read-only TRON relay functionality, ensuring that transaction creation and broadcast endpoints are not exposed unless explicitly required.

HWL-01-019 WP1: Supabase exposes multiple internal files *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During the inspection of application secrets, an *.env* file was discovered within the iOS and Android application containing a Supabase endpoint URL and a hardcoded anonymous authentication token. Investigation of this endpoint revealed the presence of multiple files, including development firmware and plugins. Consequently, an attacker could exploit this configuration to list all files and obtain internal organizational information.

Affected URL:
https://zlcbezuxlpxaylvscggj.supabase.co

Example affected endpoints:

- *https://zlcbezuxlpxaylvscggj.supabase.co/storage/v1/object/public/hw_updates/test/1.3.3/fw/MF_1.3.3+9_prod_Q*
- *https://zlcbezuxlpxaylvscggj.supabase.co/storage/v1/object/public/hw_updates/dev/1.3.0/bl/BL_0.9.74_dev_signed_1205f32.bin*
- *https://zlcbezuxlpxaylvscggj.supabase.co/storage/v1/object/public/hw_updates/plugins/fabfc2dc-72d0-4593-8172-190358dabf30/1.0.0/plugin_solana_2.0.0+15_9d7b2d23_devQA.zip*

Cure53 recommends that no internal development files should be exposed to the Internet. Additionally, it is recommended that a separate Supabase endpoint should be utilized for development and QA environments, so as to ensure proper segregation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-021 WP1: Plaintext storage of contacts and UR wallet code *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The ERA Wallet apps persist information in several different ways. Some of these utilize Flutter's secure storage mechanism, while others store information in plaintext. It was discovered that the *LocalStorageService* stores its data within a plaintext file (*app-db*) on the mobile device. Dynamically investigating the contents of this file uncovered that the file contains sensitive information such as a user's contacts, and also the UR code necessary to link the wallets of an account to a device.

This becomes a security issue if an attacker manages to gain possession of the app data (including the *app-db*) of a victim - e.g. via a rooted device - and manages to gain physical access to this device, backups, or similar paths. In such a situation, the attacker could acquire the file backing the *LocalStorageService* on the file system and extract both the contact list and the UR code necessary for linking the wallets of an account to a device. The attacker can then acquire read-only access to the wallets of an account via the issue discussed in HWL-01-009.

Cure53 initially logged this finding as a miscellaneous issue. However, the customer clarified that confidentiality violations and data-at-rest leakages due to the app exposing sensitive information in app files, shared preferences, or alike, should be logged as vulnerabilities, rather than miscellaneous issues.

Steps to reproduce (Android):

1. Connect a rooted, Android mobile device with the ERA Wallet set up with an account, to a computer running Android Studio.
2. Download the file */data/data/io.hwlt.era_sw/app_flutter/app-db/data.mdb*.
3. Open the file using hex-editor, and search for the term *UC:CRYPTO-ACCOUNT*. This will result in a hit similar to the one indicated below. This UR code corresponds to the UR code necessary when linking the device to the wallet account:

   Hit from the search:
   ```
   UR:CRYPTO-ACCOUNT/OXAD[...]EBK
   ```

Steps to reproduce (iOS):

1. Connect to a jailbroken device via SSH with the ERA Wallet app installed.
2. Inspect the values of *data.mdb*:

   Command:
   ```
   # cat /var/mobile/Containers/Data/Application/{Data container
   UUID}/Documents/app-db/data.mdb | grep -aE CRYPTO-ACCOUNT
   ```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

3. Observe that unencrypted sensitive data is shown.

Several entities utilize the class shown below, namely the *LocalStorageService* class. From the *initialize* function, it is clear that the utilized store fails to encrypt its content.

Affected file:
ERA-SW/lib/core/services/local_storage_service.dart

Affected code:

```
@singleton
class LocalStorageService {
  final Store store;

  LocalStorageService._(this.store);

  @FactoryMethod(preResolve: true)
  static Future<LocalStorageService> initialize() async {
    final docsDir = await getApplicationDocumentsDirectory();
    // NOTE (S8-T4): the store is intentionally opened WITHOUT an
encryption
    // key. On-device data-at-rest encryption is a commercial ObjectBox
feature
    // and is not exposed by the open-source Dart edition — `Store`
(objectbox
    // 5.x) has no `encryptionKey` parameter, so there is no supported way
to
    // pass one here. Documented refusal, not an oversight: at-rest
protection
    // relies on the OS app sandbox + full-disk encryption; keys/PIN live
in
    // `flutter_secure_storage` (see SecureStorageService). Revisit only
via the
    // commercial edition or by moving user data (contacts/xpub) into
secure
    // storage.
    final store = await openStore(directory: p.join(docsDir.path, 'app-
db'));
    return LocalStorageService._(store);
  }
}
```

To mitigate this issue, Cure53 recommends encrypting all sensitive items of the *LocalStorageService* before their persistence. This can be achieved by generating a key in the respective platform key store, and using this key to encrypt the item prior to its persistence in the *LocalStorageService*.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-024 WP3: Decompression bomb DoS via crafted UR codes *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While reviewing the source code of the *ERA-SW* repository, it was noted that the UR decoder for TRON applies a decompression function after decoding the UR code. Prior to this decompression, the decoder fails to heuristically infer the expected length of the decompressed raw buffer values, which could result in arbitrary large decompressed buffers after decompression. If such a buffer becomes too large, then the ERA Wallet application will eventually crash due to an out-of-memory situation. This could aid an attacker in attempts to crash the victim's app, by sending or showing the victim a crafted UR code that decompresses to an overly-large buffer.

The excerpt below demonstrates the issue. It is evident that the *parseSignature* function does not check the length before or after decompression:

Affected file:
ERA-SW/packages/era_hw_sdk/lib/src/chains/tron_codec.dart

Affected code:

```
@override
Future<HwSignature> parseSignature(UR ur, HwSignRequest request) async {
  final map = ur.decodeCBOR();
  if (map is! CborMap) {
    throw const ChainCodecException('keystone-sign-result is not a CBOR
map');
  }
  final comp = map[CborSmallInt(1)];
  if (comp is! CborBytes) {
    throw const ChainCodecException('keystone-sign-result missing payload
(key 1)');
  }
  final raw = await GZip().decompress(Uint8List.fromList(comp.bytes));
  final base = Base.fromBuffer(raw);
  return TronSignature(base.data.signTxResult.rawTx);
}
```

It is advised that ideally, *parseSignature* should write the compressed stream to a temporary file and decompress the file on the mobile device's file system, prior to loading it in memory. This would give the app the option to check the length of the decompressed raw content prior to holding it in memory. However, if this is not possible, then it is advised that the mobile apps should heuristically guess the length of the decompressed stream, and throw an error for overly-long content.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-026 WP1/WP3: Signed QR reply accepted without validation *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Review of the QR signing round-trip revealed that the phone performs only a minimal CBOR structure check on the signed-reply QR. It does not validate security-relevant fields such as *requestId* or verify that the signature corresponds to the wallet address. For Bitcoin and TRON wallets, the phone subsequently broadcasts the signed transaction contained in the QR payload.

While this cannot be used to forge transactions, as signing still requires the private key held by the ERA hardware, it allows previously signed transactions to be re-submitted as fresh signed replies. For example, a previously signed but unbroadcast transaction from a cancelled or interrupted signing flow could be re-presented and broadcast, potentially finalizing a payment that the user did not approve in the current flow. The excerpts below summarize the main interactions relevant for this security issue:

Affected file #1:
ERA-SW/lib/features/sign_transaction/presentation/widgets/hw_signature_scan_adapter.dart

Affected code #1:

```
@override
 ScanFrameOutcome<QrFrameSink> onCode(String raw) {
   final complete = _sink.feed(raw);
   _progress.value = _sink.progress;
   if (complete) return ScanFrameOutcome.complete(_sink);
   return ScanFrameOutcome.progress(_sink.progress);
 }
```

Affected file #2:
ERA-SW/packages/era_hw_sdk/lib/src/chains/btc_codec.dart

Affected code #2:

```
@override
 Future<HwSignature> parseSignature(UR ur, HwSignRequest request) async {
   if (request is BtcPsbtSignRequest) {
     final value = ur.decodeCBOR();
     if (value is! CborBytes) {
       throw const ChainCodecException('crypto-psbt response is not
bytes');
     }
     return SignedPsbt(Uint8List.fromList(value.bytes));
   }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    // btc-signature: { 1: requestId, 2: signature }.
    final map = ur.decodeCBOR();
    if (map is! CborMap) {
      throw const ChainCodecException('btc-signature is not a CBOR map');
    }
    final sig = map[CborSmallInt(2)];
    if (sig is! CborBytes) {
      throw const ChainCodecException('btc-signature missing signature (key
2)');
    }
    return RawSignature(Uint8List.fromList(sig.bytes));
  }
```

Affected file #3:
ERA-SW/packages/era_hw_sdk/lib/src/keys

Affected code #3:

```
 @override
  Future<HwSignature> parseSignature(UR ur, HwSignRequest request) async {
    final map = ur.decodeCBOR();
    if (map is! CborMap) {
      throw const ChainCodecException('keystone-sign-result is not a CBOR
map');
    }
    final comp = map[CborSmallInt(1)];
    if (comp is! CborBytes) {
      throw const ChainCodecException('keystone-sign-result missing payload
(key 1)');
    }
    final raw = await GZip().decompress(Uint8List.fromList(comp.bytes));
    final base = Base.fromBuffer(raw);
    return TronSignature(base.data.signTxResult.rawTx);
  }
```

Affected file #4:
ERA-SW/lib/features/sign_transaction/domain/services/tx_broadcaster.dart

Affected code #4:

```
[...]
final psbtBytes = (signature as SignedPsbt).psbt;

// WalletConnect bip122 signPsbt is sign-only: hand the signed (not
// finalized) PSBT straight back to the dApp as base64.
if (_dataSource.method == 'signPsbt') {
  return SignFlowSignedOnly(base64Encode(psbtBytes));
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
final transaction = Psbt.fromBase64(base64Encode(psbtBytes));
final builder = PsbtBuilderV0(transaction);
final trx = builder.finalizeAll();
final rawTxHex = trx.toHex();
final isTestnet = blockchain.blockchainType == BlockchainType.btcTestnet;
final idempotencyKey =
    crypto.sha256.convert(utf8.encode(rawTxHex)).toString();
final result = await _broadcastRemoteService.broadcastBtc(
  rawTxHex: rawTxHex,
  isTestnet: isTestnet,
  idempotencyKey: idempotencyKey,
);
_dataSource.onBroadcasted?.call(result.txHash);
handleBroadcasted(result.txHash);
return SignFlowBroadcasted(result.txHash);
[...]
 Future<SignFlowResult> _dispatchTron(TronSignature signature) async {
    final rawTxHex = signature.rawTxHex;

    // WalletConnect / arbitrary-contract path: hand the signed raw tx back
to
    // the dApp instead of broadcasting. (Result shaping for WC Tron is
debug-
    // grade pending firmware support — see the era_hw_sdk Tron notes.)
    if (_dataSource.signMessageBytes != null) {
      return SignFlowSignedOnly(rawTxHex);
    }

    final idempotencyKey =
        crypto.sha256.convert(utf8.encode(rawTxHex)).toString();
    final result = await _broadcastRemoteService.broadcastTron(
      rawTxHex: rawTxHex,
      idempotencyKey: idempotencyKey,
    );
    _dataSource.onBroadcasted?.call(result.txHash);
    handleBroadcasted(result.txHash);
    return SignFlowBroadcasted(result.txHash);
  }
```

Cure53 recommends rejecting the signed reply and preventing transaction broadcast if the UR type does not match the expected type, the reply does not echo the unpredictable *requestId*, or the signed transaction does not match the user-approved request (inputs, outputs, amount, fee, and transaction ID). For wallets holding the relevant keys (such as BTC PSBT inputs), the corresponding signatures should also be verified. These checks should be enforced as a fail-closed gate immediately before broadcasting the transaction (*tx_broadcaster.dart*).

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-027 WP1: Multiple bypasses in PIN lock mechanism *(High)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

A comprehensive analysis was conducted on the application's PIN lock mechanism, which serves as the primary security layer. The assessment revealed multiple bypass vulnerabilities carrying varying degrees of impact. Ultimately, the current implementation relies exclusively on simple client-side comparisons, and lacks robust cryptographic safeguards.

Bypass #1:
The application's lockout mechanism, which is triggered following consecutive failed authentication attempts, relies on comparing the lockout duration against the local system time via *DateTime.now()*. Consequently, an attacker can circumvent this restriction by manually advancing the device's clock forward, thereby facilitating a brute-force attack against the six-digit PIN through purely local device interactions.

Affected file:
ERA-SW/lib/features/pin/presentation/manager/pin_cubit.dart

Affected code:

```
  Future<void> _init() async {
   [...]
      _secureStorageService.getPinFailedAttempts(),
      _secureStorageService.getPinLockedUntilMs(),
      _secureStorageService.getPinLockArmedAtMs(),
    ).wait;
    _storedPinRecord = pin;
    final now = DateTime.now().millisecondsSinceEpoch;
```

Bypass #2:
As documented in HWL-01-005, the hashed PIN is stored locally within the iOS Keychain. On a jailbroken device, it is possible to overwrite the Keychain data, thereby modifying the PIN to an attacker-known value, and circumventing the entire security mechanism.

It is advised that the PIN implementation should derive a cryptographic key from the PIN's hash, and use it to encrypt all data at rest. This measure will ensure that a lock screen bypass as described in bypass #2 does not grant access to user data, while simultaneously increasing the computational cost of brute-force attacks. To mitigate against bypass #1, it is recommended to consider using server-side time, instead of local time, for deriving back-off windows.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

HWL-01-001 WP1: Android application supports outdated SDK versions *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Dynamic testing confirmed that the ERA Wallet Android app supports an outdated SDK version of Android. In particular, the *minSdkVersion* of the app is set to 26. This introduces a security risk to users running older versions of Android, due to the fact that these versions no longer receive security updates.

The excerpt below demonstrates the issue. It corresponds to a file generated by the *apktool* of Android, and it indicates that the APK supports a min SDK version of 26:

***apktool.yml*:**

```
version: 2.12.0
apkFileName: app-arm64-v8a-release.apk
usesFramework:
  ids:
  - 1
sdkInfo:
  minSdkVersion: 26
  targetSdkVersion: 36
[...]
```

To mitigate this issue, Cure53 recommends increasing the min SDK level supported by the ERA Wallet app to at least 31. If active security patches are desired, then the min SDK version should be increased to a level that is still receiving regular security updates.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-003 WP4: Non-constant time comparison of secrets *(Info)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While reviewing the source code of the *appserv* repository, it was noted that the API contains several administrative endpoints. These endpoints require knowledge of a *serverSecret* string by the caller, effectively guarding the admin API from unauthorized usage. It was found that all admin endpoints compare the provided secret to the configured *serverSecret* value by using the *!=* operator.

String compares using *!=* return at the first mismatch of characters between the two strings. When comparing cryptographic secrets using such an operator, the compare function returns on the first mismatched character. This effectively allows an attacker to measure time-differences between matching and mismatching comparisons of characters in a secret, and hence facilitate the reconstruction of the secret, character-by-character.

Affected files:

- *appserv/era_server/lib/src/endpoints/notifications_admin_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/onramp_admin_endpoint.dart*
- *appserv/era_server/lib/src/endpoints/swap_admin_endpoint.dart*

Affected code (*swap_admin_endpoint.dart*):

```
void _requireSecret(String serverSecret) {
  final expected = Serverpod.instance.getPassword('serviceSecret');
  if (expected == null || expected != serverSecret) {
    throw ServerException(
      message: 'Unauthorized',
      errorCode: 401,
    );
  }
}
```

To mitigate this issue, Cure53 recommends using constant time comparisons for all cryptographic and secret values, as implemented for many other comparisons on the backend.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-004 WP4: Path traversals in RPC proxy & moonpay API endpoints *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During a source code review of the *appserv* repository, it was noted that the backend contains several endpoints that send outbound requests to external services. When constructing paths for these external requests, it was discovered that, on some occasions, the API uses string interpolation on unsanitized strings. This effectively results in a path traversal vulnerability by using strings containing characters such as *../*

An attacker who is a legitimate user of the ERA Wallet app could send requests containing path traversal payloads in strings to endpoints of the API that trigger outbound requests. Due to the missing sanitization of the ERA API, the attacker will traverse the outbound request. This could result in the invocation of unintended endpoints, resulting in further, unspecified harm.

The excerpt below demonstrates the issue for the BTC *blockbook* client. It is used by the RPC proxy API in some of its endpoint handlers. Here, it is evident that the *address* field is directly interpolated into the URL, without first being sanitized:

Affected file #1:
appserv/era_server/lib/src/services/wallet/btc/blockbook_btc_client.dart

Affected code #1:

```
Future<List<Map<String, dynamic>>> utxos(
  Session? session,
  String address,
) async {
  final json = await _getJson(session, '/api/v2/utxo/$address');
  if (json is! List) return const [];
  return [for (final u in json) u as Map<String, dynamic>];
}
```

The excerpt below demonstrates a path traversal issue reachable via the MoonPay API - namely the *getQuote* endpoint. It is evident that the service shown below interpolates the *cryptoCode* field into the resulting URL without sanitizing it first:

Affected file #2:
appserv/era_server/lib/src/services/moonpay/moonpay_service.dart

Affected code #2:

```
Future<MoonPayQuote> getBuyQuote({
  required String cryptoCode,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  required String fiatCode,
  required double fiatAmount,
  String? paymentMethod,
}) async {
  final qp = <String, String>{
    'apiKey': _publishableKey,
    [...]
  };
  final uri = Uri.parse(
          '$_apiBaseUrl/v3/currencies/${cryptoCode.toLowerCase()}/buy_quote')
      .replace(queryParameters: qp);

  final response = await _http.get(uri);
  [...]
}
```

To mitigate this issue, Cure53 recommends sanitizing all strings from path traversal payloads before interpolating them into URLs.

HWL-01-005 WP1: PIN brute-forcing due to weak PBKDF2 parameters *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The mobile apps protect against unauthorized usage by enforcing a PIN check - consisting of 6 digits - on being opened. The PIN is stored in the secure storage of Flutter, which utilizes secure storage mechanisms on both iOS and Android. However, before persisting the PIN, the app transforms the PIN using the password protection PBKDF2-SHA256. To this end, the apps use 100,000 iterations, which is considered insecure according to official recommendations.

The purpose of such a password hashing function is to slow down brute-force attacks attempting to recover the PIN by trying all 1M combinations. Due to the fact that the app persists the hashed PIN within Flutter's secure storage, an attacker with access to the device can dump the hash and locally brute-force using *hashcat*[2] in less than 15 seconds.

PoC command:

```
# hashcat -m 10900 -a 3 hash_pin.hash ?d?d?d?d?d?d
```

The excerpt below demonstrates the issue. It is clear that the *PinHasher* class uses 100,000 iterations:

Affected file:
ERA-SW/lib/core/services/pin_hasher.dart

[2] https://hashcat.net/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
abstract final class PinHasher {
  static const int iterations = 100000;
  static const int _saltLength = 16;
  static const int _keyLength = 32;
  static const String _prefix = 'pbkdf2-v1';
  [...]
  static String hash(String pin, {List<int>? saltOverride}) {
    final salt = Uint8List.fromList(saltOverride ??
_randomBytes(_saltLength));
    final derived = _derive(pin, salt, iterations);
    return [
      _prefix,
      '$iterations',
      base64Encode(salt),
      base64Encode(derived),
    ].join(r'$');
  }
  [...]
  static Uint8List _derive(String pin, Uint8List salt, int iterationCount)
{
    final derivator = PBKDF2KeyDerivator(HMac(SHA256Digest(), 64))
      ..init(Pbkdf2Parameters(salt, iterationCount, _keyLength));
    return derivator.process(Uint8List.fromList(utf8.encode(pin)));
  }
  [...]
}
```

To mitigate this issue, Cure53 recommends exchanging the existing password hashing function with a more secure alternative - e.g. *argon2id* - or to adopt official OWASP recommendations and increase the number of iterations to at least 600,000[3].

HWL-01-006 WP4: Docker container running with root privileges *(Info)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The app server runs as a Docker container within the deployment of the ERA Wallet backend. A static code review revealed that the container of the ERA Wallet backend runs with root privileges. Running containers with elevated privileges is considered dangerous from a security perspective, as in most cases it is not necessary, but it increases the impact of a container takeover.

[3] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

If the container runs with root privileges, then an attacker could attempt to break out from the container by using publicly-known vulnerabilities, and thereby pivot onto the host running the container. This could result in further, unspecified harm.

The excerpt below showcases the affected *Dockerfile*. It is evident that the container fails to drop the root privileges and run the container as a different user:

Affected file:
appserv/era_server/Dockerfile

Affected code:
```
FROM dart:3.8.0 AS build

WORKDIR /app
COPY . .

RUN dart pub get
RUN dart compile exe bin/main.dart -o bin/server

FROM alpine:latest

ENV runmode=production
ENV serverid=default
ENV logging=normal
ENV role=monolith

COPY --from=build /runtime/ /
COPY --from=build /app/bin/server server
COPY --from=build /app/confi[g]/ config/
COPY --from=build /app/we[b]/ web/
COPY --from=build /app/migration[s]/ migrations/


EXPOSE 8080
EXPOSE 8081
EXPOSE 8082

ENTRYPOINT ./server --mode=$runmode --server-id=$serverid --
logging=$logging --role=$role --apply-migrations
```

It is recommended to change the user running the application in the container to a non-root user, as this will effectively minimize the impact of a container takeover.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-008 WP4: Insufficient HMAC check in Onramper endpoints *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While reviewing the source code of the *onramper_endpoint.dart* file, it was noted that the *createCheckout* and *createSellCheckout* functions perform a HMAC check, but only on the request's wallet address. The endpoints are used to create the checkout process, and the payloads sent to these endpoints contain other vital information for the checkout - such as *amount*, *network*, *fiat*, and *crypto*, among others. Due to the fact that only the *walletAddress* parameter is part of the HMAC authentication, all other fields of the request could be altered by a sophisticated attacker.

In particular, an attacker who managed to place themselves in between the ERA Wallet mobile app and its backend could alter all fields of the *createCheckout* and *createSellCheckout* but the request's wallet address. If the victim did not double-check the ultimate checkout process, then they could sign an unintended checkout due to this manipulation.

The excerpt below demonstrates the issue. It shows the *createCheckout* function. It is evident that the HMAC check only includes the request's *walletAddress* field. The same observation applies to the *createSellCheckout* function:

Affected file:
appserv/era_server/lib/src/endpoints/onramper_endpoint.dart

Affected code:

```
Future<OnramperCheckoutResponse> createCheckout(
  Session session,
  AuthEnvelope? hmac,
  OnramperCheckoutRequest request,
) async {
  [...]
  await HmacVerifier.verify(
    session,
    hmac,
    'onramper.createCheckout',
    utf8.encode(request.walletAddress),
  );

  final pendingTx = await OnramperTransaction.db.insertRow(
    session,
    OnramperTransaction(
      [...]
      walletAddress: request.walletAddress,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        network: request.network,
        crypto: request.crypto,
        fiat: request.fiat,
        fiatAmount: request.amount,
        [...]
      ),
    );
    [...]
}
```

To mitigate this issue, Cure53 recommends always considering the entire payload of a request in its HMAC sums. This will effectively prevent unauthorized tampering with any payload field of any request.

HWL-01-012 WP4: Path traversal in Onramper and preview API endpoints *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While further reviewing the backend for path traversal vulnerabilities (as also documented in HWL-01-004), Cure53 uncovered further locations in the source code suffering from such issues. In particular, some of the endpoint handlers for the Onramper API (e.g. *getSellQuotes*) and the endpoints of the preview API uses strings such as *fiat* or *crypto*, among others, when interpolating request URLs, without sanitizing them first. This results in a path traversal vulnerability, effectively enabling an attacker to reach unintended endpoints on the outbound APIs via traversal using characters such as *../*

The request below demonstrates the issue for the *getSellQuotes* function. It should be noted that the *fiat* field contains a traversal payload. From the response, it is clear that the traversal completed successfully, as the response contains quote-related values in the respective response fields:

Request #1:

```
POST /api/onramper HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 145
host: api.era-wallet.com
Connection: keep-alive

{"hmac":null,"crypto":"usdt_tron","fiat":"usd/../../../quotes/usdt_tron/
usd","cryptoAmount":419.96,"paymentMethod":null,"method":"getSellQuotes"}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Response #1:

```
HTTP/1.1 200 OK
[...]

[{"__className__":"OnramperQuote","ramp":"alchemypay","paymentMethod":"cred
itcard","quoteId":"01KZ6XGMZ80AR8X4BQX7NYA6G0alchemypay","rate":1.0,"payout
":406.0311,"networkFee":0.0,"transactionFee":13.9289,"kycRequirements":
{"__className__":"OnramperKycRequirements","onramperLoginRequired":false,"o
nramperKycRequirement":"NO_KYC","partnerLoginRequired":true,"partnerKycRequ
irement":"KYC_REQUIRED"},"recommendations":["BestPrice","Recommended"]},
[...]]
```

The request below demonstrates the issue for the *getQuotes* function. It should be noted that it contains a traversal payload in the *crypto* field. Again, from the response, it is clear that the traversal completed successfully, as the response contains quote-related values in the respective response fields:

Request #2:

```
POST /api/onramper HTTP/1.1
user-agent: Dart/3.11 (dart:io)
content-type: application/json; charset=utf-8
Accept-Encoding: gzip, deflate, br
Content-Length: 132
host: api.era-wallet.com
Connection: keep-alive

{"hmac":null,"fiat":"usd","crypto":"usdt_ethereum/../../usd/
usdt_ethereum","amount":100.0,"paymentMethod":null,"method":"getQuotes"}
```

Response #2:

```
HTTP/1.1 200 OK
[...]

[{"__className__":"OnramperQuote","ramp":"coinify","paymentMethod":"creditc
ard","quoteId":"01KZ6XA5XRQ39599SXKVEV95AZcoinify","rate":1.057809175866911
5,"payout":94.535009037,"transactionFee":5.5,"availablePaymentMethods":
[{"__className__":"OnramperQuotePaymentMethod","paymentTypeId":"banktransfe
r","name":"Bank","icon":"https://cdn.onramper.com/icons/payments/
banktransfer.svg"},
{"__className__":"OnramperQuotePaymentMethod","paymentTypeId":"creditcard",
"name":"Credit
Card","icon":"https://cdn.onramper.com/icons/payments/creditcard.svg"}],"ky
cRequirements":
{"__className__":"OnramperKycRequirements","onramperLoginRequired":false,"o
nramperKycRequirement":"NO_KYC","partnerLoginRequired":true,"partnerKycRequ
irement":"KYC_REQUIRED"},"recommendations":["BestPrice","Recommended"]},
[...]]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The excerpt below demonstrates the issue for both the *getQuotes* and *getSellQuotes* endpoints of the Onramper API. It is evident that the service issuing the outbound request uses string interpolation directly, without sanitizing the parameters used for the interpolation. The other affected file contains similar constructs.

Affected files:

- *appserv/era_server/lib/src/services/onramper_service.dart*
- *appserv/era_server/lib/src/endpoints/preview_endpoint.dart*

Affected code (*onramper_service.dart*):

```
Future<List<OnramperQuote>> getQuotes({
  required String fiat,
  required String crypto,
  required double amount,
  String? paymentMethod,
  String? country,
}) async {
  [...]
  final uri = Uri.parse('$_baseUrl/quotes/$fiat/$crypto')
      .replace(queryParameters: qp);

  final response = await _http.get(uri, headers: _authHeaders);
  [...]
}
[...]
Future<List<OnramperQuote>> getSellQuotes({
  required String crypto,
  required String fiat,
  required double amount,
  String? paymentMethod,
  String? country,
}) async {
  [...]
  final uri = Uri.parse('$_baseUrl/quotes/$crypto/$fiat')
      .replace(queryParameters: qp);

  final response = await _http.get(uri, headers: _authHeaders);
  [...]
}
```

To mitigate this issue, Cure53 recommends sanitizing all strings from path traversal payloads before interpolating them into URLs.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

HWL-01-013 WP4: Insufficient HMAC check on MoonPay session creation *(Low)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

As a follow up to HWL-01-008, Cure53 further scrutinized the backend for insufficient HMAC checks. This revealed that also the MoonPay session creation endpoint suffers from an insufficient HMAC check issue. In particular, the HMAC check of the endpoint handler to create a new session only involves the request's wallet address, and fails to take into account all other fields - e.g. *fiatCode*, *fiatAmount*, or *paymentMethod*, among others.

This cryptographic flaw would allow an attacker who managed to place themselves between mobile app and backend to alter all fields of MoonPay session creation requests but the *walletAddress* field. The victim may therefore use tampered values in the further workflow involving MoonPay.

The excerpt below demonstrates the issue. It is evident that the *createSession* function implements a HMAC check only accounting for the *request.walletAddress* field:

Affected file:
appserv/era_server/lib/src/endpoints/moonpay_endpoint.dart

Affected code:

```
Future<MoonPayCheckoutResponse> createSession(
  Session session,
  AuthEnvelope? hmac,
  MoonPayCheckoutRequest request,
) async {
  [...]
  await HmacVerifier.verify(
    session,
    hmac,
    'moonpay.createSession',
    utf8.encode(request.walletAddress),
  );
  [...]
  await MoonPayTransaction.db.insertRow(
    session,
    MoonPayTransaction(
      [...]
      deviceId: request.deviceId,
      status: 'created',
      walletAddress: request.walletAddress,
      [...]
      fiatCode: request.fiatCode,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        fiatAmount: request.fiatAmount,
        paymentMethod: request.paymentMethod,
        createdAt: DateTime.now(),
        updatedAt: DateTime.now(),
      ),
    );
    [...]
    final widgetUrl = _service.buildSignedWidgetUrl(
      cryptoCode: currency.code,
      fiatCode: request.fiatCode,
      fiatAmount: request.fiatAmount,
      walletAddress: request.walletAddress,
      [...]
      externalCustomerId: request.customerId,
      [...]
      redirectUrl: request.redirectUrl,
    );

    return MoonPayCheckoutResponse(
      widgetUrl: widgetUrl,
      externalTransactionId: externalTransactionId,
      status: 'created',
    );
  }
```

To mitigate this issue, Cure53 recommends always considering the entire payload of a request in its HMAC sums. This will effectively prevent unauthorized tampering with any payload field of any request.

HWL-01-020 WP3: Insecure parsing of UR codes could result in exceptions *(Info)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During a source code review of the BC-UR flows, it was noted that the package to handle UR codes contains several models - each of which has its own parsing implementation. Review of the parser showed that the corresponding implementations fail to check if keys are present in the CBOR maps after decoding. Instead, the handlers directly access elements for such keys, which could lead to null values and consequent exceptions / errors being raised. If the application utilizing these parsers fails to catch the exception, then this results in a crash.

The listing below enumerates all folders that were found to contain UR parsers suffering from this issue:

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected files:

- *ERA-SW/packages/bc_ur_dart/lib/src/models/btc/**
- *ERA-SW/packages/bc_ur_dart/lib/src/models/common/**
- *ERA-SW/packages/bc_ur_dart/lib/src/models/eth/**
- *ERA-SW/packages/bc_ur_dart/lib/src/models/key/**

The excerpt below demonstrates the issue for one file (namely the *fragment.dart* file). It is evident that the *fromUR* function fails to check the existence of the accessed keys in the *data* map:

Affected code (*common/fragment.dart*):

```
static FragmentUR fromUR({required UR ur}) {
  final data = ur.decodeCBOR();
  if (data is! CborList || data.length != 5) throw Exception('Invalid
fragment ur');

  final seqNum = (data[0] as CborInt).toInt();
  final seqLength = (data[1] as CborInt).toInt();
  final messageLength = (data[2] as CborInt).toInt();
  final checksum = (data[3] as CborInt).toInt();
  final part = Uint8List.fromList((data[4] as CborBytes).bytes);
  [...]
}
```

To mitigate this issue, Cure53 recommends revisiting all code parts utilizing these UR parsers, and ensuring that all of them are guarded with a *catch* block.

HWL-01-022 WP1: ECDSA malleability due to missing length check *(Low)*

***Fix note:** This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While reviewing the use of cryptographic primitives, it was noted that ECDSA signature generation lacks a length check. The function *verifyPubKeyFromRaw* verifies an ECDSA signature in raw format (i.e. P1363 encoding). For this encoding, the size of the signature should be twice the size of a field element of the underlying field - i.e. 64 bytes for the curves *secp256r1* and *secp256k1*.

The missing check allows signature malleability. I.e., it is possible for an attacker to modify an existing signature for a given message *M* by inserting additional 0 bytes before the values *r* and *s*. Because of the missing length check, the modification is not noticed, and the modified signature is still accepted as a valid signature for *M*.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The severity of a signature malleability issue depends on how the signature verification is being used in the project. Such an issue can for example be a serious vulnerability if the raw signatures are being used to detect replay attacks. Cure53 has analyzed the calling code and concluded that an exploit of the issue is unlikely, which is the reason for its *Low* severity rating.

Affected file:
appserv/era_server/lib/src/crypto_util.dart

Affected code:

```
static bool verifyPubKeyFromRaw({
   required String publicKey,
   required String data,
   required String rawSignature,
 }) {
   final signatureBytes = HexUtils.decode(rawSignature);
   final r = signatureBytes.sublist(0, signatureBytes.length ~/ 2);
   final s = signatureBytes.sublist(signatureBytes.length ~/ 2);

   return CryptoUtils.ecVerify(
     CryptoUtils.ecPublicKeyFromPem(publicKey),
     utf8.encode(data),
     ECSignature(
       BigInt.parse(HexUtils.encode(r), radix: 16),
       BigInt.parse(HexUtils.encode(s), radix: 16),
     ),
     algorithm: "SHA-256/ECDSA",
   );
 }
```

Cure53 recommends adding a check for the length of the signature. I.e., *signatureBytes* should be exactly 64 bytes long for the curves *secp256r1* and secp256k1, respectively *2*((pub.parameters.n.bitLength + 7)~/8)* in general.

HWL-01-023 WP1: Inclusion of sensitive Keychain data in iOS backups *(Low)*

***Fix note:** This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During the review of secure data storage within the iOS platform, it was observed that multiple sensitive data elements are stored which utilize the *WhenUnlocked* accessibility level. The implications of this configuration are that these sensitive items are included within both iCloud backups and encrypted iTunes backups. An attacker who managed to gain possession of these backups could exfiltrate sensitive data from them.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Steps to reproduce:

1. Using a jailbroken device running *Frida*, connect with *Objection* and dump the Keychain content:

   Command:

```
> objection -f start
# ios keychain dump
[...]
WhenUnlocked                    None  Password  pin
flutter_secure_storage_service  pbkdf2-
v1$100000$XC0A13CYgB56DOD0CEr2sA==$3BSRh3teeIHbqYPad8Ydmnsk71nOmBcJ[.
..]
WhenUnlocked                    None  Password
hmac_install_secret_v1   flutter_secure_storage_service
P9TMleMfG2[...]
```

2. Observe that these items are stored with *WhenUnlocked*.

To mitigate this issue, it is advised that the accessibility level of such keys should be changed to *AfterFirstUnlockThisDeviceOnly*. Keychain items marked with *ThisDeviceOnly* are cryptographically bound to the device's UID, effectively preventing this type of compromise.

HWL-01-025 WP1: Use of weak cryptographic libraries *(Info)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The project was found to depend on a number of weak cryptographic libraries. This is almost certainly a consequence of the lack of reputable cryptographic libraries available for the Dart language. During the audit, potential weaknesses were taken into account. The code was examined for explicit calls to weak primitives and potential attacks stemming from their use.

In particular the code was checked for the following weaknesses:

- *basic_utils.CryptoUtils* contains a timing side channel in ECDSA, that allows recovery of the private key. This primitive is used in the signing code implemented in *appserv/era_server/lib/src/crypto_util.dart*. Fortunately, this code is not used in any production environment, and there are no plans to do so.
- *basic_utils.CryptoUtils.ecVerify* is susceptible to signature malleability. Signatures for a message *M* can be modified, so that the resulting modified signature is still accepted as a valid signature for *M*. Signature malleability can sometimes allow replay attacks. However, code review of the project did not find an exploit.
- *basic_utils.CryptoUtils.rsaVerify* accepts some invalid signatures. The weakness is not serious enough to allow signature forgeries. Signature malleability is possible, but does not lead to any issues in this project.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- *secp256k1* contains a timing side-channel during signature generation that allows recovery of the private key. *secp256k1* is used in this project, but not for generating signatures.

Affected file:
appserv/era_server/lib/src/crypto_util.dart

Affected code:

```
static String signData(String data, ECPrivateKey privateKey) {
    final dataBytes = utf8.encode(data);
    ECSignature sig = CryptoUtils.ecSign(privateKey, dataBytes,
        algorithmName: 'SHA-256/ECDSA');
    final signature = CryptoUtils.ecSignatureToBase64(sig);
    return HexUtils.encode(base64Decode(signature));
  }

  static String signDataWithRaw(String data, ECPrivateKey privateKey) {
    final dataBytes = utf8.encode(data);
    ECSignature sig = CryptoUtils.ecSign(privateKey, dataBytes,
        algorithmName: 'SHA-256/ECDSA');
    return HexUtils.encode(Uint8List.fromList([
      ...HexUtils.decode(sig.r.toRadixString(16)),
      ...HexUtils.decode(sig.s.toRadixString(16)),
    ]));
  }
```

The package *basic_utils* is based on the package *pointycastle.* The elliptic curve implementation of this package is not constant time. In particular, the time it takes to sign a message is strongly correlated with the bit-length of the nonce *k* used during the signing. This timing leak allows an attacker to select a set of signatures for which the most significant bits of *k* are all 0. As a consequence, the attack by Howgrave-Graham and Smart[4] can be used. D.Wong published a detailed analysis and PoC[5].

While Wong's work required considerable effort to remove noise from the collected timings, an attack against the *basic_utils* package is much easier. The main reason is that the timing differences of up to 200 microseconds are exceptionally large for such an attack. Noise reduction can be achieved by selecting the fastest signatures. The image below shows the time measured for an ECDSA signature depending on the bit-length of the nonce *k*. Only the fastest 1% of the signatures are shown, in order to reduce noise:

[4] N. A. Howgrave-Graham and N. P. Smart. "*Lattice attacks on digital signature schemes."* Designs, Codes and Cryptography, 23(3):283–290, 2001.

[5] D.Wong, *"Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really?",* https://eprint.iacr.org/2015/839.pdf

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

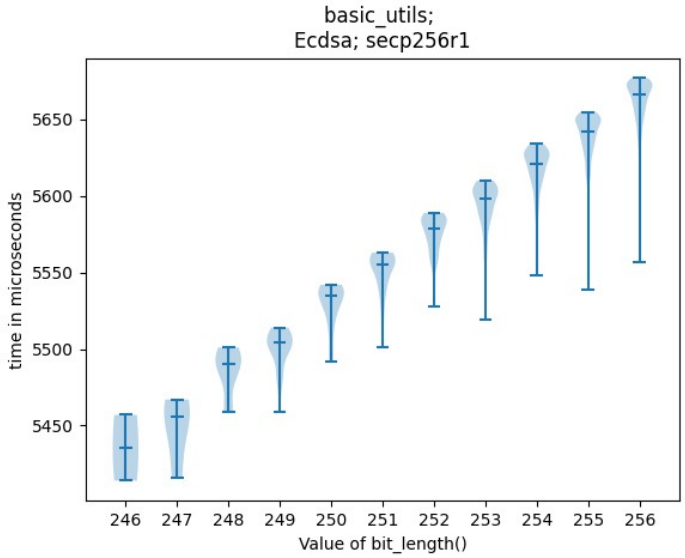

Fig.: Time it takes to generate a secp256r1 signature depending on nonce bit-length.

Another library used in the project is *secp256k1.* This library has a similar timing leak during the generation of ECDSA signatures. The time it takes to sign a message is again strongly correlated to the bit-length of the nonce *k* used during the generation of the signature. The following image shows the time for generating an ECDSA signature depending on the length of the nonce. As above, only the fastest 1% of the signatures were used, so as to remove noise. The image contains an artifact for signatures with nonces of size 256. This artifact is caused by the fact that signatures are normalized. For the attack itself, this has little impact.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

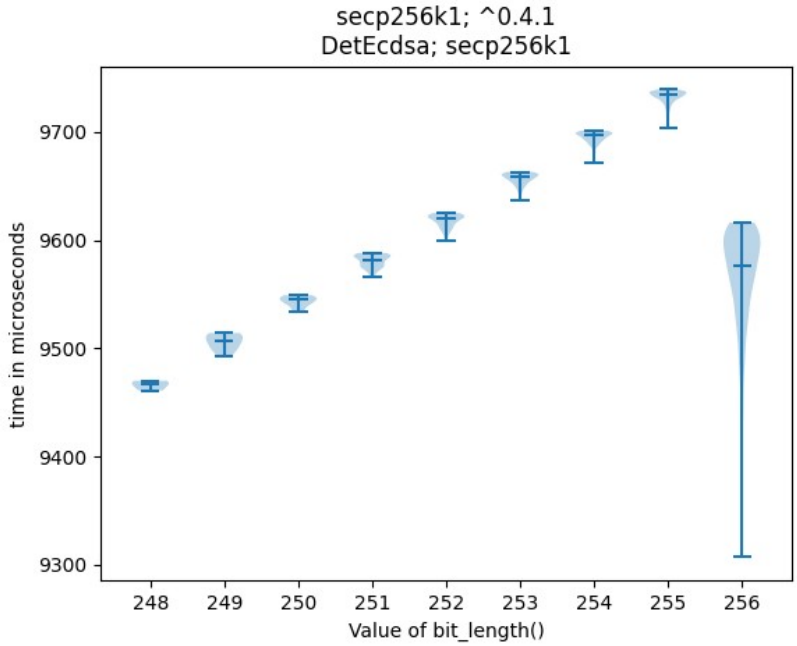

Fig.: Time it takes to generate a secp256k1 signature depending on nonce bit-length.

While the dart package *secp256k1* is used in the project, it was verified during the review that the package is not used to sign any message. However, the intention behind this finding is to raise the awareness of security issues in the underlying packages when using them for signature generation.

HWL-01-028 WP1: Biometric verification bypass in iOS *(Medium)*

Fix note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

An evaluation of the local biometric security controls within the iOS application indicated that the implementation relies solely on a Boolean validation of the iOS biometric API. There is no cryptographic binding between the application data or session and the biometric authentication result. As a result, an attacker with code injection capabilities could hook the relevant API methods and modify the returned value to artificially simulate a successful authentication, thereby completely circumventing the biometric security control.

Affected file:
ERA-SW/lib/features/pin/presentation/manager/pin_cubit.dart

Affected code:

```
Future<void> useBiometric(String reason) async {
    try {
      final auth = await _localAuthentication.authenticate(
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            localizedReason: reason,
            // Biometric ONLY — never offer the device passcode as a fallback.
                 On
            // cancel / failure / unavailable biometrics the call returns false
                 (or
            // throws) and the user falls back to the in-app PIN screen, not
                 the OS
            // passcode (SW-299).
            biometricOnly: true,
            persistAcrossBackgrounding: true,
          );
          if (auth) {
            // The OS gate vouched for the owner — clear the brute-force
                 counter
            // alongside the unlock. (While a lockout is active the biometric
            // affordances are gated off in PinPage, so this only runs
                 unlocked.)
            // Storage reset runs after the emit, same as the PIN path.
            _clearLockStopwatch();
            [...]
}
```

It is advised that the result of the biometric verification should no longer be treated as the final authentication decision. Instead, biometric authentication should be utilized to unlock a secret securely stored within the Keychain, ensuring that a forced successful response yields no usable key material and the wallet remains locked. Apple documents this specific architectural pattern in Accessing Keychain Items with Face ID or Touch ID[6].

[6] https://developer.apple.com/[...]/localauthentication/accessing-keychain-items-with-face-id-or-touch-id

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the *Introduction*, this August 2026 (CW32) penetration test and source code audit was conducted by Cure53 against the ERA wallet mobile applications and Serverpod RPC API.

From a contextual perspective, eighteen working days were allocated to reach the coverage expected for this project. The methodology used conformed to a white-box strategy, and a team of five senior testers was assigned to the project's preparation, execution, and finalization. As expected for this type of assessment, testing primarily consisted of code review, supplemented by dynamic testing.

Testing comprised a total of four work packages (WPs). WP1 assessed the security of the ERA Wallet mobile apps in general, whereas WP2 and WP3 targeted two of the apps' main flows - namely the transaction construction, as well as the QR and BC-UR flows. Lastly, WP4 assessed the security of the API utilized by the mobile apps.

The ERA Wallet and Cure53 teams shared a dedicated Slack channel to facilitate all communications. The customer demonstrated high responsiveness and addressed all inquiries promptly. While no major impediments were encountered during the testing phase, it should be noted that the absence of the appropriate hardware wallet constrained the exploration of certain specific features. Cure53 live reported all of the engagement's findings.

Prior to the start of testing, the ERA Wallet team shared all of the necessary information for Cure53 to begin the security assessment. This included access to the corresponding source code repositories, development builds for the mobile apps, URLs, and documentation. Additionally, the customer also provided builds for auxiliary applications that were necessary in order to perform dynamic tests.

The ERA Wallet mobile app acts as a proxy app between hardware wallets and the ERA Wallet API, to transfer and manage funds. The purpose of the app is to separate the process of signing transactions using private keys from the ERA Wallet mobile apps themselves, thereby shielding the private keys of wallets from Internet connectivity; the signing flows rely on QR and BC-UR code scanning instead. The mobile app is developed using Flutter, and the backend is entirely written in Dart. All source code repositories were found to be well-organized, and it was straightforward for the testing team to grasp the implemented functionality.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Mobile applications

As noted above, Cure53's assessment of the ERA Wallet mobile applications was conducted utilizing a white-box methodology, with comprehensive access granted to the source code. The mobile application leverages the Flutter SDK and shares the majority of its codebase across the iOS and Android platforms.

A thorough review of both versions was conducted. The team noted that certain additional protection mechanisms - such as TLS pinning - were not implemented in the shared application.

The application contains an HMAC app pepper used for account registration. As this secret is present on the mobile device, an attacker may be able to extract and reuse it to register multiple accounts. This does not appear to align with the intended design, which seems to use the HMAC app pepper to bind account creation to the respective device.

It is advised that additional attention should be given to the lack of TLS pinning and the application's support for older iOS versions (such as iOS 16), as well as jailbroken devices and outdated Android versions (HWL-01-001). Although the product is designed to interact with local hardware, which significantly limits the impact of these conditions, the implementation of the additional protection measures discussed is still recommended.

Following the evaluation of the in-device attack surface, the PIN lock mechanism was subjected to detailed scrutiny, which revealed multiple bypass vulnerabilities. Most notably, it was found that the time lock restriction enforced after continuous failed authentication attempts could be trivially bypassed, without technical complexity, simply by advancing the device's clock (HWL-01-027).

During an investigation into the local secure storage, hardcoded tokens and a Supabase endpoint were discovered within an *.env* file. Although no direct impact affecting these authentication tokens was identified, the Supabase instance exposed numerous files from non-production environments that should not be accessible via the Internet (see HWL-01-019). Furthermore, the team also noted that both apps persist sensitive information (such as contacts or wallet BC-UR codes) in plaintext (HWL-01-021).

The transaction flow, as well as the QR and BC-UR code flows, were investigated for correctness and potential flaws. Here the testing team gained a generally positive impression of the respective flows. However, some minor issues were also identified. For example, HWL-01-026 discusses a recommended improvement to the mobile application's transaction logic. The finding concerns the lack of validation of transaction details contained in QR code responses received from the hardware wallet. Although the issue cannot be directly exploited to steal funds, it may allow transactions that were cancelled at the last minute, or subsequently frozen, to proceed to broadcast. Further, HWL-01-024 documents

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

how ZIP bombs could crash the app on scanning codes for the TRON network. Lastly, it was found that the BC-UR decoders, and other related decoders failed to enforce strict data validations on codes, which could potentially result in unhandled exceptions (HWL-01-020).

iOS

The iOS application exhibited a limited attack surface, characterized by a near-total absence of Inter-Process Communication (IPC) mechanisms and external deep links, thereby minimizing the potential vectors for remote exploitation against the application. Consequently, local attack vectors were analyzed, revealing numerous deficiencies in the correct deployment of security protocols. It is advised that the mechanism for securing sensitive data on the device exhibited an absence of rigorous protective measures, coupled with specific configuration errors pertaining to the iOS ecosystem. An example of this is HWL-01-023, which allows the backup of sensitive data.

An additional instance of the inadequate implementation of platform-specific security mechanisms was noted within the iOS environment, involving biometric controls. Specifically, the operating system's API functions merely as a Boolean authentication mechanism, rather than a robust cryptographic asset, thereby rendering the security control susceptible to straightforward bypass techniques (HWL-01-028).

In conclusion, it is advised that the iOS application exhibits non-compliance with Apple’s security specifications in critical domains, specifically regarding security APIs and secure data storage. However, owing to the architectural nature of the hardware wallet, these vulnerabilities did not directly compromise user funds or transaction integrity, and impacted the mobile client environment exclusively.

Android

The Android app was investigated for common flaws. The team began by investigating the app's manifest, and checked all exported activities, services, and alike for flaws. Here, the ERA Wallet app was found to export only a single activity.

Next, the exported activity was fuzzed for DoS situations, revealing an attack vector that could crash the mobile app (HWL-01-002). The activity was also checked for deep links or similar that could allow phishing attempts or bypasses, but no such exposures were detected.

In terms of backups, it was found that the app opts-out from app backups - decreasing its attack surface. This is a positive sign. The signing schemes used to sign the app’s APK were investigated. No usage of the insecure v1 signing scheme was found.

Finally, the team investigated task hijacking vulnerabilities. Due to the utilized modes, it was concluded that the StrandHogg 1.0 vulnerability is not exploitable, however it remained unclear if an attacker could exploit the StrandHogg 2.0 vulnerability.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

In summary, the Android application made a comparatively good impression on the testing team. Overall for the mobile applications, it is advised that the apps are in a moderate state from a security perspective. As evident from the list of issues discovered, there is still room for improvement with regard to security and the implementation of defense-in-depth measures. Information leakage, as well as access to sensitive information and bypasses, were found to constitute major areas of concern.

Backend

Large parts of the assessed scope consisted of the backend API, where a number of severe issues were identified. It is advised that particular attention should be given to authentication and authorization, as multiple instances failed to enforce the expected security controls. HWL-01-011, for example, discusses administrative authorization checks implemented within assert statements, which are not evaluated in production environments. Similarly, some endpoints leak data to unauthenticated users (HWL-01-007). Further, HWL-01-018 demonstrates that one of the endpoints can be abused by unauthenticated attackers to use the ERA wallet's TRON relay at no cost, by misusing the established proxy. Other endpoints enable the unauthorized linkage of a device to existing accounts if the fingerprint of an account is known (which is public information) (HWL-01-009). An unauthenticated attacker may even disrupt the availability of the entire ERA Wallet backend (HWL-01-010).

Cure53 scrutinized the backend for injection flaws - such as code / command injection, and SQL injection (SQLi). Given that no sinks for commands or similar were in use, no issues could be identified in this area. Furthermore, the backend uses OR mapper functionality to access the underlying database, thereby safeguarding against SQLi attacks.

HWL-01-014 demonstrated that the mechanism used to determine users' IP addresses can be manipulated by users, and cannot therefore not be considered to be reliable. Similarly, HWL-01-015 discusses how attackers may abuse the implemented rate-limiting mechanism to deny access to publicly accessible, rate-limited endpoints.

The ERA Wallet backend also sends outbound HTTP requests to upstream services. Most of these upstream requests were seen to adopt best practices, however, some exemptions to this were noted. These exemptions suffered from path traversal vulnerabilities, allowing an attacker to traverse the outbound APIs used by the backend (HWL-01-004, HWL-01-012).

On a more minor note, Cure53 also found that the backend Docker container runs with root privileges (HWL-01-006). Further, a service that was deemed to be out-of-scope was seen to suffer from an open redirect vulnerability (HWL-01-016).

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

In summary, it is advised that the backend of the ERA Wallet is in a brittle state from a security perspective. Numerous severe flaws were identified, particularly involving authentication and authorization. These flaws were mainly caused by missing HMAC checks, which are vital in enforcing proper access control to the ERA Wallet API.

Cryptography

As the ERA Wallet is a crypto coin wallet, special attention was paid to all cryptographic primitives in use by the mobile apps and backend. Generally, it was found that cryptographic primitives were used correctly, and keys, nonces, and algorithm parameters were found to be correctly selected to give a good level of security in the majority of cases. However, an exception to this was noted involving the password hashing function used to persist a user's PIN and its parameters (HWL-01-005).

Most cryptographic primitives use underlying libraries. The primitives that were implemented in the code were reviewed, with the implementation typically being adequate. One omission is detailed in HWL-01-022, which leads to a signature malleability issue. Fortunately, in this case, there is no immediate effect, given that the protocol design is robust and does not depend on such implementation details. Signature malleability is a frequent issue with cryptographic libraries (for example the RSA signature verification in *pointycastle* suffers from such a problem). Hence, it is important to write robust surrounding code in order to avoid such issues. The code review found no places where signature malleability would lead to issues.

A major issue was noted involving the selection of underlying cryptographic libraries. It was found that none of the libraries used in the project is in a robust state, and all have significant vulnerabilities. Some of the main issues with these libraries are described in HWL-01-025. In particular, the library used to implement ECDSA has a major flaw that could leak the private key. Fortunately, the corresponding code is not used in a critical path of the project. Nevertheless, it is advised that having such code is dangerous, as it could easily lead to vulnerabilities when extending the codebase. The fault here is not with the project itself, but rather the Dart ecosystem, which does not seem to have well-written and reviewed cryptographic libraries available. ERA Wallet is aware of this issue and mentioned that there is a plan to implement critical cryptographic operations outside of the Dart ecosystem. It is advised that this is a good plan, and will help to ensure a robust project.

Cure53 also investigated important instances of the use of secure randomness. Here, it was found that the backend fails to utilize secure randomness to generate device attestation values (HWL-01-017).

Although the choice of cryptographic primitives was sound, some schemes were seen to be lacking in terms of fully-authenticating data on some occasions. In particular, HWL-01-008 and HWL-01-013 document two instances where the HMAC values fail to take into account whole request bodies. This would leave room for attack if an attacker was to successfully

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

tamper with the request bodies of the affected endpoints. In general, on most occasions, the backend was seen to use constant time comparisons of cryptographic relevant values or secrets. However, the team identified some exemptions from this secure approach (HWL-01-003).

Overall conclusions

In summary, it is advised that the ERA Wallet complex appears to be in a brittle state from a security perspective. While no *Critical* severity findings were made during this engagement, numerous *High* severity vulnerabilities were discovered. For the backend API, a recurring flaw was noted involving a lack of authentication and authorization. It is advised that this should be reworked systematically in order to close this security gap. For the mobile app, the access and handling of sensitive data in particular was seen to suffer from numerous issues.

Moving forward, Cure53 strongly recommends addressing all of the identified issues in a timely manner. Once all issues have been mitigated, it is recommended to perform a re-test against the entire target, with the same scope, to ensure that all issues have been correctly remediated.

Cure53 would like to thank Grigorii Siabruk, Dan Mutsvanga, Igor Skladchikov, and Ochir Mandzhiev from the ERA Wallet (HWLT FZE) team for their excellent project coordination, support and assistance, both before and during this assignment.