Qoinpay

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Caue Obici caue@osec.io

Naveen Kumar J naina@osec.io

Contents

01 | Executive Summary

Overview

Qoinpay engaged OtterSec to perform an assessment of the Qoinpay Application. This assessment was conducted between February 6th and February 26th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 11 findings total.

In particular, we reported issues around the email-adding feature (OS-QPY-ADV-05) and encryption techniques (OS-QPY-ADV-03), along with issues related to the web view (OS-QPY-ADV-01, OS-QPY-ADV-02).

We also made recommendations around code obfuscation (OS-QPY-SUG-00) and suggestions to improve encryption techniques (OS-QPY-SUG-02, OS-QPY-SUG-03).

02 | Scope

The source code was delivered to us in a repository at bitbucket.org/loyaltoid/audit_app/src/master/. This audit was performed against commit `2f023ba`.

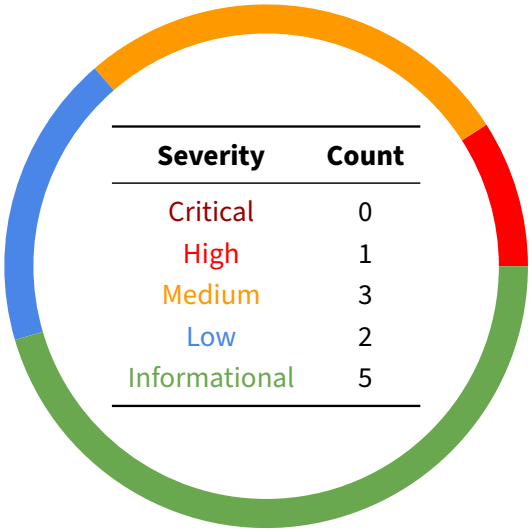
A brief description of the programs is as follows.

Name	Description
Qoinpay	A digital wallet for monetary transactions within Indonesia that offers a range of services, such as the payment of bills, the purchase of goods and services, and the transfer and request of funds.

03 | Findings

Overall, we reported 11 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	1
Medium	3
Low	2
Informational	5

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-QPY-ADV-00	High	Resolved	Encryption keys and secrets are embedded in the code.
OS-QPY-ADV-01	Medium	Resolved	An attacker can manipulate the WebView URL due to inadequate validation of QR data.
OS-QPY-ADV-02	Medium	Resolved	`onLoadStart` may cause an open redirect in the WebView when used with a malicious fragment.
OS-QPY-ADV-03	Medium	Resolved	The `IV` is generated twice while encrypting and decrypting data using `fromLength`, which is not secure.
OS-QPY-ADV-04	Low	Resolved	The mobile application lacks SSL pinning and is potentially vulnerable to MITM attacks.
OS-QPY-ADV-05	Low	Resolved	`phoneNumberConvertDev` incorrectly parses phone numbers that start with a country code.

OS-QPY-ADV-00 [high] | Hardcoded Secret Data

Description

The encryption keys and secret keys are embedded in the code. These keys must be set appropriately before the product ships so that the secrets cannot mistakenly be exposed to potential attackers. Attackers could utilize these secret keys to carry out various types of attacks.

1. **Encryption secrets**

```dart
//audit_qoin_crypto/../hive_storage.dart
Future<void> put({required String key, required dynamic value}) async {
    var encrypted = value == null ? null : Encrypt.encrypt(value, "****");
    var encryptedKey = Encrypt.encrypt(key, "****");
}

// qoin_services/../merchant_controller.dart
String decryptAESData(String source) {
if (QoinServices.environment == 'production') {
  key = en.Key.fromUtf8('****');
}

// qoin_services/../ota_controller.dart
String decryptData(String source) {
if (QoinServices.environment == 'production') {
  key = en.Key.fromUtf8('****');
}

// audit_crypto_module/../encrypt_data.dart
class EncrypData {
  static final _key = Key.fromUtf8('****');
}
```

Gaining access to the secret encryption key could have severe consequences as it allows access to sensitive information that was intended to be protected through encryption. In this case, the encryption key that is found in the `hive_storage.dart` file is used to encrypt the application's PIN, which is a crucial secret for the application. The PIN is used for authentication purposes to access the application, perform transactions, modify the user's profile, and more.

2. **API keys**

```dart
// qoin_account/../account_repo.dart
Future<BaseResp> getToken() async {
    request: {"SecretId": "****","SecretKey": "****"},
```

```
  }

// qoin_wallet/../yukk_repo.dart
const String yukkProdToken = '****';

// paprika_wallet/../paprika_repo.dart
String clientSecretProd    = "****";
String clientIdProd        = "****";
```

The exposure of an API secret key may lead to significant consequences, as these keys are generally utilized to authenticate and authorize access to an API. Such access may be used to obtain access to sensitive data, conduct actions on behalf of a user, or interact with other systems. For example, in the case of the `qoin_account`, the API secret key is employed to verify the status of an account and generate OTPs.

3. **JWT Secrets**

DART

```
// qoin_services/../qoin_services.dart
static Future<void> paymentQR({@required String dataQR}) async {
    var _decrypt = EnDecUtils.jwtVerifyUtil(encrypted, "****");
}

// qoin_wallet/../yukk_scanner_screen.dart
void woPayment(String scanData) {
    var _decrypt = EnDecUtils.jwtVerifyUtil(encrypted, "****");
}

// wallet_offline/../wo_pay_scanner_screen.dart
Future<void> _onQRViewCreated(QRViewController controller) async {
    var _decrypt = EnDecUtils.jwtVerifyUtil(encrypted, "****");
}
```

The disclosure of the secret used to sign JWTs may result in severe consequences. An attacker with access to this secret may produce their JWTs with arbitrary claims, thereby circumventing any authorization or authentication checks that rely on the validity of the tokens. For instance, in the context of an application, the exposed keys are employed to sign payloads and verify QR code data with sensitive fields, such as the amount, merchant, and others that may be compromised.

4. **APK Signing Keys**

DART

```
storePassword=****
keyPassword=****
keyAlias=****
storeFile=****
```

> If the `key.properties` file is made public, an attacker may use this information to gain access to the key and sign their own version of the app with the same key. This may allow the attacker to distribute malicious versions of the app, including malware, spyware, or other malicious code. This may potentially compromise the user's device or personal information.

An additional disadvantage is the lack of flexibility. Suppose the key is compromised or needs to be changed for any other reason, then the entire codebase may need to be modified and recompiled.

Remediation

Remove all the hardcoded secrets from the client-side codebase. It is possible to create API routes that perform the same function that the client-side is performing by using the secrets. This way, the client-side would not have access to these secrets, and they will not be leaked by reverse engineers.

Patch

Partially resolved in `d8b210a` by introducing a keychain to store the secrets. Even though, the secrets are still handled on client side, Talsec mobile app protection makes it harder to access the secrets by reverse engineers. Qoinpay acknowledges that this mitigation is likely sufficient without larger rewrites of the architecture.

env.dart DIFF

```
+         (await FlutterKeychain.get(key: "env"))!)["qoinServiceMerchantAesKey"];
    }

-   static String merchantDataKey() {
-     return const String.fromEnvironment("qoinServiceMerchantAesKey");
+   static Future<String> verificationKeyMerchant() async {
+     return jsonDecode(
+         (await FlutterKeychain.get(key: "env"))!)["qoinServiceQrJwtKey"];
    }

-   static String verificationKeyMerchant() {
-     return const String.fromEnvironment("qoinServiceQrJwtKey");
+   static Future<String> get accountSecretKey async {
+     return jsonDecode(
+         (await FlutterKeychain.get(key: "env"))!)["qoinAccountSecretKey"];
    }

-   static String accountSecretKey = const
      ↪  String.fromEnvironment("qoinAccountSecretKey");
-   static String accountSecretId = const
      ↪  String.fromEnvironment("qoinAccountSecretId");
+   static Future<String> get accountSecretId async {
+     return jsonDecode(
+         (await FlutterKeychain.get(key: "env"))!)["qoinAccountSecretId"];
+   }
```

```
+
+  static Future<String> get encryptKeyCryptoModule async {
+    return jsonDecode((await FlutterKeychain.get(key: "env"))!)[
+        "cryptoModuleEncryptKey"]; //QoinCryptoWallet
+  }

-  static String encryptKeyCryptoModule = const
     ↪  String.fromEnvironment("cryptoModuleEncryptKey"); //QoinCryptoWallet
+  static Future<String> get aesIVCryptoModule async {
+    return jsonDecode((await FlutterKeychain.get(key: "env"))!)[
+        "aesIVCryptoModule"]; //QoinCryptoWallet
+  }
```

OS-QPY-ADV-01 [med]| QR Code Scanner URL Validation

Description

Insufficient validation of the URL host permits attackers to construct payloads that can redirect users to malicious websites within the WebView page. The application validates scanned QR data by verifying the presence of the string `frodo2weborderdev.page.link`. However, this validation may be circumvented by URLs such as `https://evil.com/?frodo2weborderdev.page.link` or `https://frodo2weborderdev.page.link.evil.com`.

DART

```dart
// qoin_wallet/.../scanner_screen.dart
} else if (scanData.code.contains(QoinWallet.urlQrQoinLounge)) {
    scannerController.qoinLoungeQR(scanData.code);
return;

// qoin_wallet/.../merchant_screen.dart
case SourceType.qoinlounge:
  MerchantController.to.merchantUrl = widget.qrData;
  break;
}

// qoin_wallet/.../web_view_page.dart
initialUrlRequest: URLRequest(
  url: Uri.parse(widget.url),
  headers: {},
),
```

The consequences of an attacker obtaining control over the WebView URL may be severe, as it may potentially affect the user's security and privacy through various malicious attacks. For example, phishing and cross-site scripting may potentially result in significant harm.

Remediation

Validate a URL after parsing it using the `Uri.parse(url)` method.

Patch

Fixed in `e770abd` by parsing the URL and strictly comparing the URL host to the allowed host.

scanner_controller.dart DIFF

```
   void qoinLoungeQR(String qrData) async {
-    if (QoinWallet.qoinWalletFeatures.isQoinLoungeQR == false) {
+    if (QoinWallet.qoinWalletFeatures.isQoinLoungeQR == true) {
```

```
+      Uri resultUri = Uri.tryParse(qrData);
+      if (resultUri.host == QoinWallet.urlQrQoinLounge) {
+        Get.back();
+        SmartSystemUtil.openQoinLounge(qrData);
+      } else {
+        QWDialogUtils.showFailedPopUp(
+            title: QoinWalletLocalization.theresProblem.tr,
+            textButton: QoinWalletLocalization.close.tr,
+            description: '');
+        qrViewController.resumeCamera();
+      }
+    } else {
```

OS-QPY-ADV-02 [med]| Open Redirect In Native WebView

Description

A malicious URL containing `S.browser_fallback_url=<evil_site>` in a URL fragment may cause an open redirect vulnerability. Despite strict validation being performed on the host in the QR code, there is no validation being performed on `S.browser_fallback_url`. This URL is loaded directly into the WebView without any checks, leading to an open redirect vulnerability.

DART

```
onLoadStart: (controller, url) {
  String newUrl;
  if (Platform.isAndroid) {
    if ('$url'.startsWith('intent://${QoinServices.qrQoinLounge}')) {
      webView.stopLoading();
      try {
        newUrl = url.fragment.split(";").firstWhereOrNull((element) =>
                element.split("=")[0] == "S.browser_fallback_url").split("=")[1];
      } catch (e) {}
      if (newUrl != null) {
        newUrl = Uri.decodeFull(newUrl) +
            ↪  '&source=qoinpay&data=${QoinAccount.userData.fullName};...';
        controller.loadUrl(urlRequest: URLRequest(url: Uri.parse(newUrl)));
        return null;
      }
    }
  }
}
```

A malicious actor may launch various types of attacks, such as phishing, cross-site scripting, and others. These attacks potentially result in significant harm to both security and privacy.

Note that this attack only works on Android devices.

Remediation

Perform validation on URLs that contain the `S.browser_fallback_url` fragment to ensure that they originate from trusted sources. Specifically, URLs that come from QR code data should not be allowed directly.

Patch

Fixed in `37a0889` by parsing the `S.browser_fallback_url` and checking its host against the allowed host.

web_view_page.dart DIFF

```
+       Uri newUrlUri;
+       newUrlUri = Uri.tryParse(newUrl);
+       if (newUrlUri != null &&
+           newUrlUri.origin ==
+               QoinServices.webOrderQoinLounge) {
+         newUrl = Uri.decodeFull(newUrl) +
+                                     '&source=qoinpay&data=[...]}
+         controller.loadUrl(
+           urlRequest:
+               URLRequest(url: Uri.parse(newUrl)));
+         return null;
+                               }
```

OS-QPY-ADV-03 [med] | Unsecure And Incorrect Encryption

Description

It is important to note that `IV.fromLength` is not designed to function as a cryptographically secure pseudorandom number generator. Instead, the purpose of `fromLength` is to generate initialization vectors (IVs) that are initialized with zeros of a specified length, n. IVs are typically generated randomly and it is crucial to generate them using a proper random number generator, rather than relying on `fromLength` to ensure cryptographic security.

The `IV`s are incorrectly generated twice in `encrypt` and `decrypt`. This currently works when the function `IV.fromLength` returns the same output. However, in the case of random `IV` generators, the `IV` used for encryption should be used for decryption in order to decrypt data correctly.

RUST

```
class Encrypt {
  static String encrypt(dynamic data, String pin) {
    ...
    final iv = IV.fromLength(16);
    final encrypted = encrypter.encryptBytes(utf8.encode(json.encode(data)), iv:
    ↪  iv);
    ...
  }

  static dynamic decrypt(String text, String pin) {
    ...
    final iv = IV.fromLength(16);
    var decryptBytes = encrypter.decryptBytes(Encrypted.from64(text), iv: iv);
    ...
  }
}
```

Remediation

It is recommended to generate a secure pseudorandom `IV` in order to ensure secure encryption and decryption.

Patch

Fixed in `d8b210a` and `530c268` by generating the IV in the backend. Now, the client fetches the IV through an API and stores it in a keychain.

env.dart DIFF

```
@@ -63,6 +67,27 @@ class EnvironmentConfig {
     }
   }

+  static Future<String> eciIV() async {
+    return jsonDecode((await FlutterKeychain.get(key: "env"))!)["aesIVEci"];
+  }
+
```

qoinbase.dart DIFF

```
+  static Future<void> _setupEnv() async {
+    try {
+      var recordList = (await pb
+              ?.collection(
+                  "${packageInfo.appName.split(" ").first.toLowerCase()}_env")
+              .getList(filter: 'packageid = "${packageInfo.packageName}"'))
+          ?.items;

...

+      if (record != null) {
+        _envSucceed = true;
+        env = record.toJson();
+        envCallback?.call(record.data["data"]);
```

OS-QPY-ADV-04 [low] | Missing SSL Pinning

Description

SSL pinning is a security measure used by mobile applications to ensure secure communication with the intended server and prevent Man-in-the-Middle (MITM) attacks. However, the mobile application currently does not implement SSL pinning.

Without SSL pinning, an attacker can intercept traffic between the mobile application and the server using a proxy tool such as Burp Suite. As a result, they may modify the traffic in transit or inject their own malicious code. This could lead to several security vulnerabilities, including data theft, session hijacking, or unauthorized access to sensitive information.

Remediation

Utilize the following code snippet as the app uses Dio to implement SSL pinning.

DART

```dart
final dio = Dio();
ByteData bytes = await rootBundle.load('certificates/demo.pem');
(dio.httpClientAdapter as DefaultHttpClientAdapter).onHttpClientCreate  = (client)
    ↪ {
  SecurityContext sc = SecurityContext();
  sc.setTrustedCertificatesBytes(bytes.buffer.asUint8List());
  HttpClient httpClient = HttpClient(context: sc);
  return httpClient;
};
```

Patch

Fixed in e45646c by implementing Talsec FreeRasp rather than certificate pinning in the application. This security library hardens the app but SSL pinning is left unimplemented. Qoinpay acknowledges that this mitigation is probably sufficient without larger rewrites of the architecture.

qoinbase.dart DIFF

```
+  TalsecConfig config = TalsecConfig(
+    // For Android
+    androidConfig: AndroidConfig(
+      expectedPackageName: expectedPackageName,
+      expectedSigningCertificateHash: expectedSigningCertificateHash,
+      supportedAlternativeStores: supportedAlternativeStores,
+    ),
+
+    // For iOS
```

```
+    iosConfig: IOSconfig(
+      appBundleId: iosAppBundleId,
+      appTeamId: iosAppTeamId,
+    ),
+
+    // Common email for Alerts and Reports
+    watcherMail: watcherMail,
+  );
+
+  // Talsec callback handler
+  TalsecCallback callback = TalsecCallback(
+    // For Android
+    androidCallback: AndroidCallback(
+      onRootDetected: () => QWDialogUtils.showRootedPopup(),
```

OS-QPY-ADV-05 [low] | Incorrect Parsing Of Phone Numbers

Description

`phoneNumberConvertDev` is not correctly parsing phone numbers that start with a country code. If the true number is '91XXXXXXX' and the country code is '91', the result should be '+9191XXXXXX'. Instead, it is being interpreted as '+91XXXXXX'.

DART

```
String phoneNumberConvertDev(String? phoneNumber, String countryCode) {
  if (phoneNumber != null && phoneNumber.isNotEmpty) {
    String standardNumber =
        phoneNumber.replaceAll('-', '').replaceAll(' ', '').replaceAll('+', '');
    String finalNumber = "";
    if (standardNumber.startsWith('0')) {
      finalNumber = standardNumber.substring(1);
    } else if (standardNumber.startsWith(countryCode)) {
      finalNumber = standardNumber.substring(2);
    } else if (standardNumber.startsWith('+$countryCode')) {    // Redundant `+` is
        ↪  already removed
      finalNumber = standardNumber.substring(3);
    } else {
      finalNumber = phoneNumber;
    }

    return countryCode + finalNumber;
  } else {
    return "";
  }
}
```

This issue may prevent users from registering or logging into the application, as the OTP verification will not be possible due to an incorrect phone number.

Remediation

Ensure that the function does not trim the phone number if it starts with the country code.

Patch

Fixed in `454dbb9` by removing the phone trimming feature.

any_util.dart DIFF

```
@@ -22,9 +22,11 @@ String phoneNumberConvertDev(String? phoneNumber, String
    ↪  countryCode) {
     String finalNumber = "";
     if (standardNumber.startsWith('0')) {
       finalNumber = standardNumber.substring(1);
-    } else if (standardNumber.startsWith(countryCode)) {
-      finalNumber = standardNumber.substring(2);
-    } else if (standardNumber.startsWith('+$countryCode')) {
+    }
+    //  else if (standardNumber.startsWith(countryCode)) {
+    //   finalNumber = standardNumber.substring(2);
+    // }
+    else if (standardNumber.startsWith('+$countryCode')) {
       finalNumber = standardNumber.substring(3);
     } else {
       finalNumber = phoneNumber;
```

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-QPY-SUG-00	Code obfuscation should be used when building the app to increase the difficulty of understanding when decompiled.
OS-QPY-SUG-01	Authenticate the user when going from the background.
OS-QPY-SUG-02	The app currently uses an unencrypted hive box for data storage, which is not secure.
OS-QPY-SUG-03	The encryption algorithm currently used is considered weak and vulnerable to attacks.
OS-QPY-SUG-04	Unnecessary logging of sensitive information in the production environment should be avoided.

OS-QPY-SUG-00 | Code Obfuscation

Description

It is important to ensure that all business and security logic is not in a human-readable format when decompiled through third-party sources. The app code and compiled binaries may be reverse-engineered, revealing strings, method and class names, and API keys, which can help attackers understand the logic very quickly.

Remediation

Utilize code obfuscation to increase the difficulty of reading and understanding the code. Obfuscation can be implemented by using the following command while building the application.

BASH

```
flutter build apk --obfuscate --split-debug-info=./ProjectFolderName/debug
```

OS-QPY-SUG-01 | Lock App When In Background

Description

It has been observed that the application will not be locked if it is moved to the background, even when the phone is locked and subsequently unlocked. Authorization must be requested to access the application from the background, primarily because the application contains sensitive data.

To replicate the behavior, first open and unlock the application, then move it to the background. Even after performing some unrelated tasks outside of the application or locking and subsequently unlocking the device and reopening the application, it will not prompt for a PIN.

Anyone with access to the phone, even without knowledge of the app PIN, can gain entry into the application and potentially access sensitive information.

Remediation

Implement authorization logic when the app is in the background with the help of observers.

OS-QPY-SUG-02 | Encrypt Data Storage

Description

Currently, the app stores sensitive data in an unencrypted hive box in the local storage. Although the data is encrypted explicitly before being stored in the hive box, this method is not secure. It is recommended to use alternative options that ensure encryption.

There are two options that can be employed to ensure secure data storage:

1. Hive already supports encrypted hive boxes where values are encrypted but not keys.
2. An alternative solution is to use the `flutter_secure_storage` plugin to store data in the local storage. This plugin utilizes the Keychain for iOS and Android's AES encryption internally to ensure security.

Remediation

Utilize one of the following methods to ensure encrypted data storage.

1. If access time is a major concern, use an encrypted hive box by including the following code:

   ```dart
   await Hive.openBox("boxName", encryptionCipher: HiveAesCipher(key));
   ```

   Note that keys will remain unencrypted.
2. If security is the primary concern, then use the `flutter_secure_storage` plugin.

OS-QPY-SUG-03 | Use Strong Encryption Algorithm

Description

The current encryption algorithm, AES-ECB, encrypts each plaintext block independently, meaning that identical plaintext blocks will always be encrypted to identical ciphertext blocks. This can be exploited by attackers who compare the ciphertext blocks to reveal patterns in the plaintext, leading to a complete loss of confidentiality.

Moreover, AES-ECB is vulnerable to replay attacks, tampering attacks, chosen-plaintext attacks, and brute-force attacks.

DART

```
final encrypter = en.Encrypter(en.AES(key, mode: en.AESMode.ecb));
```

Remediation

Use other encryption modes, such as AES-GCM, that provide better security features and protect against various types of attacks.

OS-QPY-SUG-04 | Remove Unnecessary Logging

Description

In order to prevent potential security vulnerabilities, it is recommended to avoid logging sensitive information in production environments. During the audit, it was observed that certain application dependencies were logging sensitive information such as the user PIN during entry.

wallet_offline/../wo_pin_screen.dart DART

```
child: PinScreen(
    ...
    pinValue: (value) {
        debugPrint(value);
    },
    onSuccess: ...
```

Remediation

Remove unnecessary logging in production. One method of preventing logs is to override `debugPrint`.

DART

```
debugPrint = (String message, {int val}) {};
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.