F Y E O

Aureus-ox-ios October Ongoing review

Aureus-ox

October 2025
Version 0.12

Presented by:
FYEO Inc.
PO Box 147044
Lakewood CO 80214
United States

Reviewed by: Balthazar@gofyeo.com
Thomas@gofyeo.com

Security Level
Public

Table of Contents

Executive Summary

Overview

Aureus-ox engaged FYEO Inc. to perform an ongoing review of Aureus-ox-ios.

The assessment was conducted remotely by the FYEO Security Team. Testing took place on September 27 - September 30, 2025, and focused on the following objectives:

- To provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the results of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the FYEO Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following issues have been identified during the testing period. These should be prioritized for remediation to reduce the risk they pose:

- FYEO-AO-ID-01 – Insufficient Attestation Verification - Missing Cryptographic Validation
- FYEO-AO-ID-02 – Missing Array Bounds Validation in Event Decoding
- FYEO-AO-ID-03 – Sleep Used for Synchronization - Brittle Timing Logic
- FYEO-AO-ID-04 – Unhandled Error Propagation in Async Task Closures
- FYEO-AO-ID-05 – Incomplete test coverage

Based on our review process, we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

The FYEO Review Team performed a Aureus-ox-ios October Ongoing review. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://github.com/Aureus-Ox/aureus-ox-ios/ with the commit hash poc/xrpl_integration.

Files included in the code review

aureus-ox-ios/
└── OxenFlow/
  └── OxenFlow/
  └── Contracts/

Table 1: Scope

Technical Analyses and Findings

During the Aureus-ox-ios October Ongoing review, we discovered:

- 1 finding with HIGH severity rating.
- 1 finding with MEDIUM severity rating.
- 2 findings with LOW severity rating.
- 1 finding with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

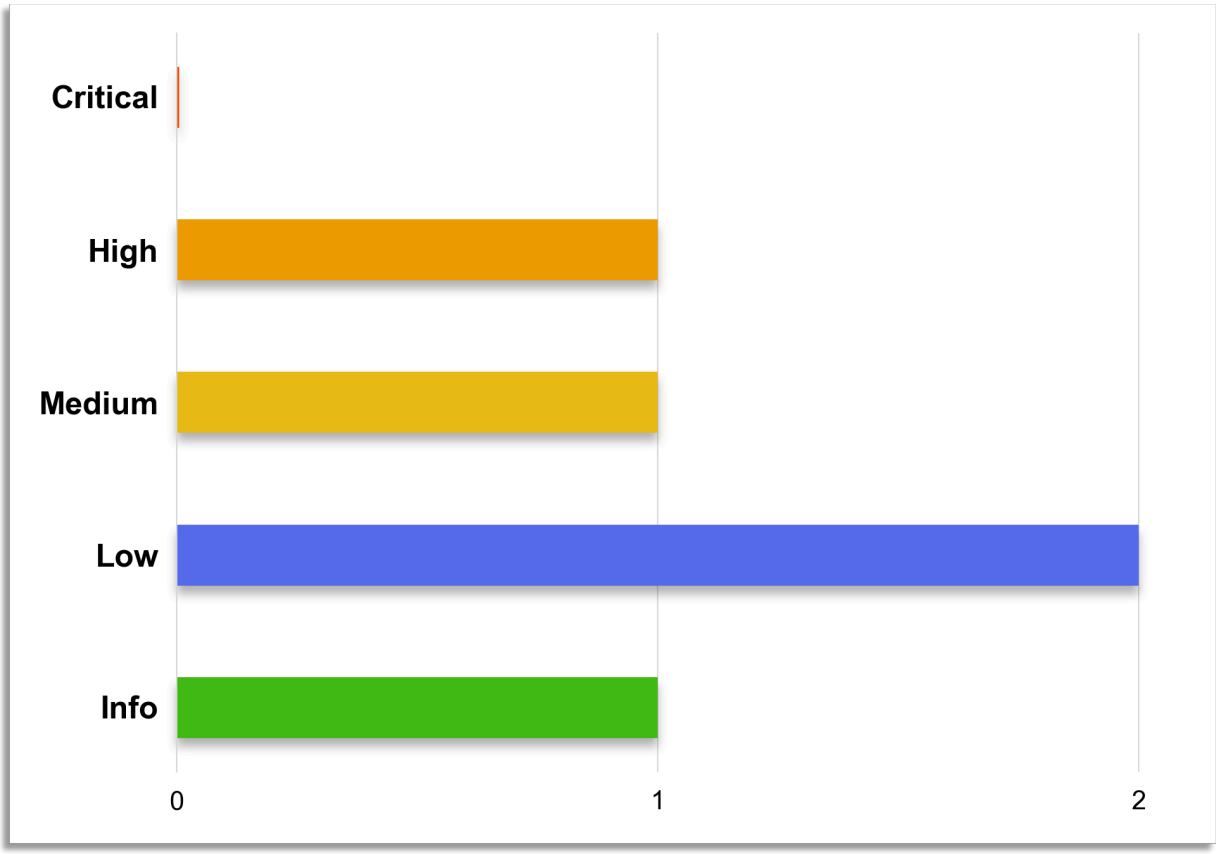

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

Finding #	Severity	Description	Status
FYEO-AO-ID-01	High	Insufficient Attestation Verification - Missing Cryptographic Validation	Remediated
FYEO-AO-ID-02	Medium	Missing Array Bounds Validation in Event Decoding	Remediated
FYEO-AO-ID-03	Low	Sleep Used for Synchronization - Brittle Timing Logic	Remediated
FYEO-AO-ID-04	Low	Unhandled Error Propagation in Async Task Closures	Remediated
FYEO-AO-ID-05	Informational	Incomplete test coverage	Remediated

Table 2: Findings Overview

The Classification of vulnerabilities

Security vulnerabilities and areas for improvement are weighted into one of several categories using, but is not limited to, the criteria listed below:

Critical – vulnerability will lead to a loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - vulnerability has potential to lead to a loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that cannot be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Txn signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - vulnerability hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-reviewed crypto functions
- Program crashes, leaves core dumps or writes sensitive data to log files

Low – vulnerability has a security impact but does not directly affect the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

Technical Analysis

The source code has been manually validated to the extent that the state of the repository allowed. The validation includes confirming that the code correctly implements the intended functionality.

Conclusion

Based on our review process, we conclude that the code implements the documented functionality to the extent of the reviewed code.

General Observations

This diff implements FXRP (Flare-wrapped XRP) minting and redemption functionality for the OxenFlow iOS wallet, allowing users to bridge XRP to/from the Flare network.

New Files Added

1. FdcManager.swift

- Manages interactions with Flare Data Connector (FDC)
- Submits attestation requests to verify XRP transactions
- Calculates voting round IDs for attestation processing
- Queues minting requests after attestation

2. MintingStrategy.swift

- **Core minting workflow**:
    1. Reserve collateral on Flare
    2. Submit XRP payment to agent's address
    3. Construct attestation proof of payment
    4. Submit attestation to FDC
    5. Queue minting request for execution
- Handles errors and state persistence for recovery
- Parses blockchain events (CollateralReserved)

3. RedemptionStrategy.swift

- **Redemption workflow** :
    1. Submit redemption request to burn FXRP and receive XRP
- Parses RedemptionRequested events
- Tracks redemption progress

Modified Files

4. OverviewTabModel.swift

- **UI/Business logic** for minting and redemption
- Fetches FXRP settings and available agents
- Executes minting/redemption strategies
- Tracks pending operations (mints and redemptions)
- Calculates available lots based on balances

Key Functionality

Minting Flow: User XRP → Reserve Collateral → Pay XRP → Prove Payment (Attestation) → Queue Mint → Receive FXRP

Redemption Flow: User FXRP → Submit Redemption → Receive XRP

State Management: - Tracks pending operations via `dependencies.pendingFAssetMints` and `dependencies.pendingRedemptionRequests` - Persists failed mint states for retry - Real-time updates via Combine publishers watching for completion events

Insufficient Attestation Verification - Missing Cryptographic Validation

Finding ID: FYEO-AO-ID-01
Severity: High
Status: Remediated

Description

The `constructAttestation` function fetches attestation data from a web API and uses it to submit attestation requests to the FDC (Flare Data Connector) without proper cryptographic verification of the response data.

Proof of Issue

File name: `OxenFlow/OxenFlow/Overview/Strategy/MintingStrategy.swift`

In the `constructAttestation(transactionHash:)` function (lines ~319-340):

```
    // Pause to allow attestation to be read by the server
    try await Task.sleep(nanoseconds: 2_000_000_000)

    // ⚠️ ISSUE: Fetches attestation from webservice without cryptographic
verification
    guard let response: Result<AbiBytesAttestationResponse> = try? await
dependencies.webservice.send(request: request) else {
        return .failure(.webserviceError, currentData)
    }

    switch response {
    case .success(let attestationEncoded):
        // ⚠️ ISSUE: Uses attestationEncoded.abiEncodedAttestationRequest directly

        guard let _ = attestationEncoded.abiEncodedAttestationRequest else {
            return .failure(.fdcError, currentData)
        }
        return .attestationConstructionSuccess(attestationEncoded)
    }
```

Then in `submitAttestation(attestationEncoded:)` function (lines ~342-353):

```
func submitAttestation(attestationEncoded: AbiBytesAttestationResponse) async throws ->
MintingStrategyResult {
    guard let userAddress = accountManager.ethAccountAddress,
          let abiEncodedRequest = attestationEncoded.abiEncodedAttestationRequest,
          // ⚠️ ISSUE: Submits unverified attestation data to FDC
          let expectedRoundId = try? await fdcManager.submitAttestationRequest(from:
userAddress, abiEncodedRequest: abiEncodedRequest) else {
        return .failure(.fdcError, currentData)
    }

    return .attestationSubmissionSuccess(expectedRoundId)
}
```

The code does not verify:

- That the response is cryptographically signed by a trusted authority
- That the signature is valid

Severity and Impact Summary

If the attestation web service is compromised or a man-in-the-middle attack occurs, an attacker could:

- Provide false attestation data leading to incorrect minting amounts
- Inject malicious attestation data to manipulate the minting process
- Tamper with payment verification data

Since attestations are used to verify on-chain XRPL payment events that trigger FXRP minting on Flare, lack of verification could lead to **direct financial loss** for users or the protocol.

Recommendation

- Verify Cryptographic Signatures
- Implement TLS Certificate Pinning
- Validate Response Fields and Add Replay Protection

Missing Array Bounds Validation in Event Decoding

Finding ID: FYEO-AO-ID-02
Severity: Medium
Status: Remediated

Description

The code accesses array elements by index without validating that the array has the expected size when decoding event topics.

Proof of Issue

File name: `OxenFlow/OxenFlow/Overview/Strategy/MintingStrategy.swift`

In the `parseCollateralReservationEvent(_:)` function (lines ~180-200):

```
func parseCollateralReservationEvent(_ txReciept: EthereumTransactionReceipt) ->
CollateralReserveEvent? {

    for log in txReciept.logs {
        let collateralReservedTopic = CollateralReservedTopic(from: log.topics, hex:
log.data)
        guard "0x" + collateralReservedTopic.topicHash == log.topics.first else {
continue }

        let results = collateralReservedTopic.decode()

        // ⚠️ ISSUE: Accessing array indices without checking results.count first
        guard let reservationId =
results[collateralReservedTopic.collateralReservationIdIndex] as? BigUInt,
              let valueUBA = results[collateralReservedTopic.valueUBAIndex] as? BigUInt,
              let feeUBA = results[collateralReservedTopic.feeUBAIndex] as? BigUInt,
              let paymentAddress = results[collateralReservedTopic.paymentAddressIndex]
as? DynamicUTF8String,
              let paymentReference =
results[collateralReservedTopic.paymentReferenceIndex] as? Data,
              let lastUnderlyingBlockIndex =
results[collateralReservedTopic.lastUnderlyingBlockIndex] as? BigUInt
        else { return nil }

        return CollateralReserveEvent(...)
    }

    return nil
}
```

The same pattern also appears in `OxenFlow/OxenFlow/Overview/Strategy/RedemptionStrategy.swift` in the `parseRedemptionEvent(_:)` function (lines ~63-75).

The code accesses multiple array indices without first checking that `results.count` is sufficient. If the decoded results array is smaller than the largest index being accessed, this will cause an array index out of bounds crash.

Severity and Impact Summary

Array index out of bounds errors will crash the application. Different event versions or malformed blockchain data could have different array sizes, making this a real risk. This is especially critical in a wallet application handling financial transactions where a crash during transaction processing could lead to loss of funds or incomplete state.

Recommendation

Add explicit bounds checking before accessing array elements.

Apply the same fix to `parseRedemptionEvent` in `RedemptionStrategy.swift`.

Sleep Used for Synchronization - Brittle Timing Logic

Finding ID: FYEO-AO-ID-03
Severity: Low
Status: Remediated

Description

The code uses a fixed 2-second sleep as a synchronization mechanism to wait for attestation processing on the server. This is a brittle approach that relies on timing assumptions rather than proper synchronization.

Proof of Issue

File name: `OxenFlow/OxenFlow/Overview/Strategy/MintingStrategy.swift`

In the `constructAttestation(transactionHash:)` function (line ~324):

```
func constructAttestation(transactionHash: String) async throws -> MintingStrategyResult
{
    let config = dependencies.blockchainConfig
    let xrplConfig = dependencies.xrpLedgerConfig

    guard let request = try? AbiBytesAttestationRequest(network: config, details:
AttestationRequestDetails(attestationType: .payment, urlSource: .xrp, source:
AttestationSource.source(for: xrplConfig), txId: transactionHash, inUtxo: 0, utxo: 0))
else {
        return .failure(.functionCreationError, currentData)
    }

    // Pause to allow attestation to be read by the server
    try await Task.sleep(nanoseconds: 2_000_000_000)  // ⚠️ ISSUE: Fixed 2-second delay

    guard let response: Result<AbiBytesAttestationResponse> = try? await
dependencies.webservice.send(request: request) else {
        return .failure(.webserviceError, currentData)
    }
    // ...
}
```

This approach has several problems:

- **Race conditions**: The server may not have processed the attestation within 2 seconds, leading to failures
- **Unnecessary delays**: If the server processes faster, users wait unnecessarily
- **Non-deterministic behavior**: Network conditions and server load can vary
- **Poor user experience**: Fixed delays feel unresponsive

Severity and Impact Summary

Using sleep for synchronization creates unreliable behavior that can lead to transaction failures or degraded user experience. The code makes assumptions about server processing time that may not hold under different network conditions or server loads.

Recommendation

Replace the fixed sleep with proper polling and preferably with exponential backoff.

- This approach responds immediately when the server is ready
- Handles slow server responses gracefully with retry logic
- Provides deterministic failure after a reasonable timeout
- Improves user experience with faster average response times

Unhandled Error Propagation in Async Task Closures

Finding ID: FYEO-AO-ID-04
Severity: **Low**
Status: **Remediated**

Description

Inside `Task` closures in multiple functions, there are `try await` calls without proper error handling. When errors are thrown, they propagate unhandled which can lead to silent failures.

Proof of Issue

File name: `OxenFlow/OxenFlow/Overview/OverviewTabModel.swift`

In the `executeMint(reserveFunction:unwrapCollateral:)` function (lines ~221-238):

```
func executeMint(reserveFunction: ReserveCollateral, unwrapCollateral: BigUInt = 0) {
    Task { [weak self] in
        guard let details = self?.fAssetDetails,
              let fdcManager = self?.fdcManager,
              let dependencies = self?.dependencies,
              let loadingModel = self?.loadingModel
        else { return }

        self?.latestReserveData = LatestReserveData(reserveFunction: reserveFunction,
unwrapCollateral: unwrapCollateral)

        await self?.dismissMintingAlertAction()
        await self?.setDisplayContextLoader(true)

        // ⚠️ ISSUE: try await without do/catch - errors propagate unhandled
        let status = try await MintingStrategy.route(...).execute(loadingModel:
loadingModel)

        guard case .success = status else {
            // ... error handling
        }
        // ...
    }
}
```

The same issue occurs in: - `executeMint(from:)` function (line ~289) - `executeRedeem(redeemFunction:)` function (line ~331) - `fetchFXRPSettings()` function (line ~424)

When `try await` throws an error inside a `Task` closure without a surrounding `do/catch` block, the error is unhandled in the task scope. This leads to:

- **Silent failures**: Errors are not caught or logged, making debugging impossible
- **Inconsistent state**: Partial execution with no rollback mechanism
- **Poor user experience**: Users don't know why operations failed
- **Potential crashes**: Unhandled errors can terminate tasks unexpectedly

Severity and Impact Summary

Unhandled errors in critical financial operations (minting, redeeming) can lead to silent failures where users lose funds or transactions fail without notification. This creates a poor user experience and makes debugging production issues extremely difficult.

Recommendation

Wrap all `try await` calls in `do/catch` blocks with proper error handling.

Apply the same pattern to all other functions with unhandled `try await` in Task closures - `executeMint(from:)` - `executeRedeem(redeemFunction:)` - `fetchFXRPSettings()`

Incomplete test coverage

Finding ID: FYEO-AO-ID-05
Severity: **Informational**
Status: **Remediated**

Description

The current test coverage is not complete and covers only basic positive scenarios.

Proof of Issue

File name: OxenFlow/OxenFlowTests/XRPL/TestXRPLPublicKeystore.swift

Tests are written correctly, basic checks are present and cover important scenarios: - positive scenario with a valid point on secp256k1. - case when the point lies on the wrong curve (secp256r1 instead of secp256k1) - case with incorrect x value (shortened value)

But a case can also be added where a point is on secp256k1 but is invalid, i.e., it fails the `y^2 ≡ x^3 + 7 (mod p)` check.

Proof of Issue

File name: OxenFlow/OxenFlowTests/XRPL/Tss/TestXRPLTssAccount.swift

The tests only cover positive scenarios, but it is worth adding negative scenarios and additional checks to the existing tests: - case where `derEncodeSignature()` on other `r` and `s` gives a different result than the hardcoded value from mock. this confirms that the encoding is dynamic, not static - a check that the `signingPubKey` and `txnSignature` fields are indeed set in `xrplTssAccount.constructSignedTransaction()` should be added to the existing `testTssAccount()` test - check that `constructSignedTransaction()` throws an error if `sign()` fails

Severity and Impact Summary

Insufficient test coverage can cause bugs in the code to go undetected, especially in critical parts such as address generation (XRPLPublicKeystore) and transaction signing (XRPLTssAccount).

Recommendation

Improve the test coverage.

Our Process

Methodology

FYEO Inc. uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

Figure 2: Methodology Flow

Kickoff

The project is kicked off as the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

FYEO Inc. delivers a draft report that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We report security issues identified, as well as informational findings for improvement, categorized by the following labels:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes within a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination.