F Y E O

Security Code Review of XRPL Delegate Authority

Ripple

April 2025
Version 1.0

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

Security Level
Public

Table of Contents

Executive Summary

Overview

Ripple engaged FYEO Inc. to perform a Security Code Review of XRPL Delegate Authority.

The assessment was conducted remotely by the FYEO Security Team. Testing took place on March 24 - March 31, 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-XRPL-01 – Shadowed variable
- FYEO-XRPL-02 – Code optimization
- FYEO-XRPL-03 – Concerns for Exchanges, Bridges, 3rd party tooling
- FYEO-XRPL-04 – The DelegateSet SLE can be created empty

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 Security Code Review of XRPL Delegate Authority. 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 public repository at https://github.com/XRPLF/rippled/ with the commit hash 31f4051d855ae2294ab202cc79b9533075366c4b.

The audit was done on a specific PR. Some files listed below were partially reviewed according to the PR:
https://github.com/XRPLF/rippled/pull/5354/commits/31f4051d855ae2294ab202cc79b9533075366c4b

Remediations were provided with the commit hash 8f48a4e707ba6e841eab9577baa820df59c9d667.

Files included in the code review

```
rippled/
├── include/
│   └── xrpl/
│     └── protocol/
│          ├── Indexes.h
│          ├── Permissions.h
│          ├── Protocol.h
│          ├── TxFlags.h
│          ├── detail/
│          │   ├── features.macro
│          │   ├── ledger_entries.macro
│          │   ├── permissions.macro
│          │   ├── sfields.macro
│          │   └── transactions.macro
│          └── jss.h
└── src/
    ├── libxrpl/
    │   └── protocol/
    │       ├── Indexes.cpp
    │       ├── InnerObjectFormats.cpp
    │       ├── Permissions.cpp
    │       ├── STInteger.cpp
    │       ├── STParsedJSON.cpp
    │       └── TxFormats.cpp
    ├── test/
    │   ├── app/
    │   │   └── Delegate_test.cpp
    │   └── jtx/
    │       ├── TestHelpers.h
    │       ├── delegate.h
    │       ├── flags.h
    │       ├── impl/
    │       │   ├── Env.cpp
    │       │   ├── delegate.cpp
    │       │   └── mpt.cpp
    │       ├── mpt.h
    │       └── rpc/
```

Files included in the code review

```
│           └── JSONRPC_test.cpp
└── xrpld/
    ├── app/
    │   ├── misc/
    │   │   ├── DelegateUtils.h
    │   │   └── detail/
    │   │           └── DelegateUtils.cpp
    │   └── tx/
    │       └── detail/
    │           ├── DelegateSet.cpp
    │           ├── DelegateSet.h
    │           ├── DeleteAccount.cpp
    │           ├── InvariantCheck.cpp
    │           ├── MPTokenIssuanceSet.cpp
    │           ├── MPTokenIssuanceSet.h
    │           ├── Payment.cpp
    │           ├── Payment.h
    │           ├── SetAccount.cpp
    │           ├── SetAccount.h
    │           ├── SetTrust.cpp
    │           ├── SetTrust.h
    │           ├── Transactor.cpp
    │           ├── Transactor.h
    │           └── applySteps.cpp
    └── rpc/
        ├── detail/
        │   └── TransactionSign.cpp
        └── handlers/
            └── LedgerEntry.cpp
```

Table 1: Scope

Technical Analyses and Findings

During the Security Code Review of XRPL Delegate Authority, we discovered:

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

The following chart displays the findings by severity.

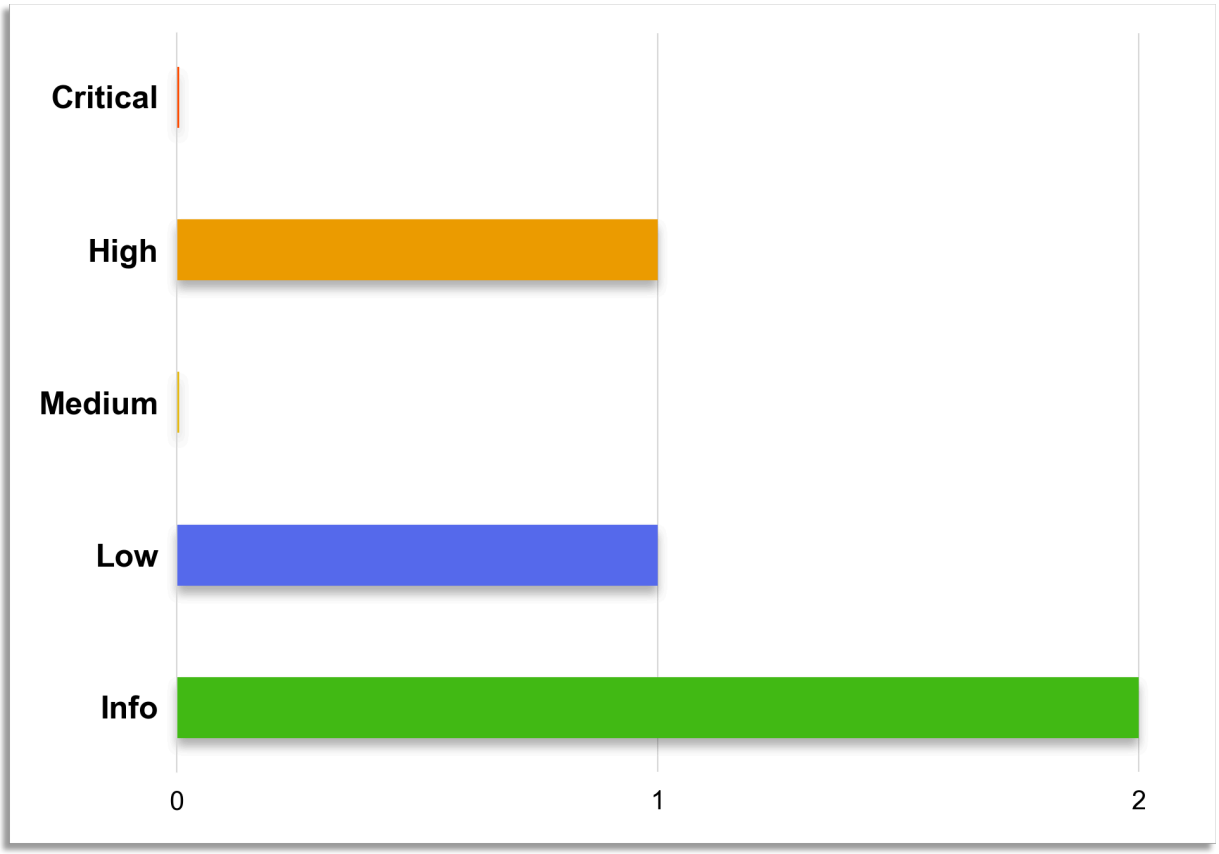

Figure 1: Findings by Severity

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
FYEO-XRPL-01	High	Shadowed variable
FYEO-XRPL-02	Low	Code optimization
FYEO-XRPL-03	Informational	Concerns for Exchanges, Bridges, 3rd party tooling
FYEO-XRPL-04	Informational	The DelegateSet SLE can be created empty

Table 2: Findings Overview

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.

Technical Findings

General Observations

The latest update to the XRPL code introduces a new delegation feature that allows an account to delegate a subset of its permissions to another account. The changes add a new ledger object type, `ltDELEGATE`, and a corresponding `keylet` function that computes the ledger index for delegate objects. A new transaction type, `DelegateSet`, is introduced along with the relevant validations and preflight checks.

The update defines a new granular permissions system where each permission is associated with a value that is offset by 1 to the corresponding transaction type. New macros and enumerations are added to support these granular permissions, and helper functions are introduced to convert between transaction types and permission values.

In several transaction processing paths such as `Payment`, `SetAccount`, `SetTrust`, and `MPTokenIssuanceSet`, the code now checks for the presence of a delegate field. If a delegate is present, the ledger is queried for the delegate object, and its permissions are used to decide whether the transaction is authorized.

In the fee processing logic, when a delegate is used, the fee is deducted from the delegate's account rather than the original sender's account. The changes also add functions to load granular permissions from a delegate ledger entry and to validate the transaction's flags against the delegated permissions.

The implications of these new features include a more flexible permission model that allows for precise control over which operations a delegated account can perform. However, this introduces complexity in ensuring that the correct account is charged for fees and that the permissions are accurately verified. This new delegation mechanism is particularly important for bridges and exchanges, as it changes the assumptions about which account's balance is affected and which permissions are used during transaction validation. Although there are no issues identical to the confusion experienced with partial payments, there is a need for careful integration to avoid misinterpretations of account balances and transaction authorizations.

Delegated Transactions and Fee Handling

The new delegate mechanism lets one account (the "authorizing" account) delegate a set of granular permissions to another ("authorized") account. In delegated transactions, the presence of the `sfDelegate` field causes the system to look up a delegate ledger object (with type `ltDELEGATE`) and, if found, use its permissions for checking whether the transaction is allowed. • In fee processing (both in fee deduction and fee payment paths), the code now checks if a delegate is present. If so, the fee is deducted from the delegate's balance rather than the sending account's.

Implications for Bridges and Exchanges

The delegate feature is used strictly for permission delegation and fee payment. • However, exchanges and bridges must be careful. For example, if they assume that the fee payer is always the original account (or that all transactions are self-signed), they might misinterpret balance changes or authorization status. • In particular, if an exchange's systems do not correctly account for fees being deducted from a delegate account, they might under- or over-credit funds or misapply fee reconciliations. As long as the exchange infrastructure is updated to handle delegated transactions, the risk is lower.

Overall, while the new feature brings greater flexibility, it also requires meticulous implementation and clear documentation to ensure that all systems, especially those interfacing with high-value operations like bridges and exchanges, correctly handle the modified logic for fee payments and permission validations.

Shadowed variable

Finding ID: FYEO-XRPL-01
Severity: **High**
Status: **Remediated**

Description

Within the function loadGranularPermission, the parameter named “type” is inadvertently shadowed by a local variable declared with the same name. As a result, the comparison intended to filter granular permissions based on the provided TxType does not work as expected. Instead of comparing the permission’s associated transaction type to the function parameter, the code compares the local variable to itself, effectively nullifying the intended filtering logic.

Proof of Issue

File name: src/xrpld/app/misc/detail/DelegateUtils.cpp
Line number: 45

```
void
loadGranularPermission(
    std::shared_ptr<SLE const> const& delegate,
    TxType const& type,
    std::unordered_set<GranularPermissionType>& granularPermissions)
{
    if (!delegate)
        return;

    auto const permissionArray = delegate->getFieldArray(sfPermissions);
    for (auto const& permission : permissionArray)
    {
        auto const permissionValue = permission[sfPermissionValue];
        auto const granularValue =
            static_cast<GranularPermissionType>(permissionValue);
        auto const& type =
            Permission::getInstance().getGranularTxType(granularValue);
        if (type && *type == type)
            granularPermissions.insert(granularValue);
    }
}
```

Severity and Impact Summary

This shadowing issue is significant because it causes the function to load granular permissions without the intended filtering. As a result, all permissions present in the delegate ledger object could be accepted, even if they do not match the expected transaction type. This may lead to incorrect behavior when processing transactions that depend on precise permission filtering, potentially allowing unauthorized or unintended operations to be performed on behalf of an account.

Recommendation

It is recommended to rename the inner variable to a distinct name, such as granularTxType, so that the comparison can correctly verify that the granular permission's transaction type matches the intended TxType provided as a parameter. This change will ensure that the function behaves as intended and only loads the permissions that are truly authorized for the given transaction type. Additionally, the revised code should be thoroughly tested to confirm that it meets the intended functionality and does not introduce any regressions.

Code optimization

Finding ID: FYEO-XRPL-02
Severity: Low
Status: Remediated

Description

A few general improvements could be made to enhance clarity and maintainability. The issues identified are not security concerns but rather areas where the code could be refactored for better performance, readability, and resilience to future changes.

Proof of Issue

File name: src/xrpld/app/misc/detail/DelegateUtils.cpp
Line number: 37

```
auto const transactionType = tx.getTxnType();
for (auto const& permission : permissionArray)
  if (permissionValue == transactionType + 1)
```

This does the same calculation in every iteration even though it could be done just once outside of the loop.

File name: src/xrpld/app/misc/DelegateUtils.h
Line number: 20

```
#ifndef RIPPLE_APP_MISC_DELEGATEUTILS_H_INLCUDED
#define RIPPLE_APP_MISC_DELEGATEUTILS_H_INLCUDED
...
#endif  // RIPPLE_APP_MISC_DELEGATEUTILS_H_INLCUDED
```

There is a typo in this identifier in the word `INLCUDED`.

File name: src/xrpld/app/misc/detail/DelegateUtils.cpp
Line number: 37

```
if (permissionValue == transactionType + 1)

if (Permission::getInstance().isProhibited(permissionValue - 1))
```

Throughout the code base there are calculations to translate ids by adding or subtracting 1. This strategy is somewhat prone to errors. It is recommended to have stricter helper functions that can better guard against potential issues. A helper function could look like this:

```
static PermissionValue fromTxType(TxType tx)
```

File name: src/libxrpl/protocol/Permissions.cpp
Line number: 109

```
bool
Permission::isProhibited(std::uint32_t const& value) const
{
    if (value == ttSIGNER_LIST_SET || value == ttREGULAR_KEY_SET ||
        value == ttACCOUNT_SET || value == ttDELEGATE_SET ||
```

```
        value == ttACCOUNT_DELETE)
        return true;

    return false;
}
```

It may be better to maintain a whitelist in this function rather than a blacklist so new types aren't assumed to be available.

Severity and Impact Summary

These issues do not pose immediate security threats; however, they affect the code's maintainability, performance, and clarity. In the long term, inefficiencies, inconsistent or error-prone arithmetic for permission values, and reliance on a blacklist could lead to subtle bugs or misconfigurations as the system evolves. Addressing these concerns would help prevent potential mistakes and reduce technical debt.

Recommendation

These suggestions improve the maintainability of the code base and prevent potential mistakes. Centralizing the conversion logic will reduce the risk of errors. Consider using a whitelist rather than a blacklist for permitted transaction types in delegation. This would ensure that any new transaction types are denied by default until they are explicitly approved for delegation, thereby strengthening the overall security posture as the protocol evolves.

Concerns for Exchanges, Bridges, 3rd party tooling

Finding ID: FYEO-XRPL-03
Severity: Informational
Status: Acknowledged

Description

The introduction of delegated transactions changes critical assumptions about fee payment sources and transaction authorization, creating risks for systems such as exchanges and bridges that rely on historical fee-deduction logic. Key concerns include:

Unexpected Fee Source: Delegated transactions deduct fees from the delegate account (authorizing account) rather than the account that is doing the 'transacting' (Account field). - Systems assuming fees are always charged to the account doing the transacting may miscalculate balances (e.g., under-credit users if delegate fees are unaccounted for, or over-credit if fees are double-counted).

Authorization Misinterpretation: Delegated transactions use permissions from the `ltDELEGATE` ledger object, the identifier of these objects is derived from the delegate and the account that is transacting. Systems that check sender-owned permissions directly (e.g., for compliance) might incorrectly approve/deny transactions.

Assumption-Driven Risks: Bridges assuming "self-signing == fee payment" could misattribute transaction costs, especially in cross-chain operations where delegate accounts are external.

Proof of Issue

This is not a code defect in the XRPL implementation itself. The risk arises from incorrect assumptions in external systems (exchanges, bridges, analytics tools) that process transactions without accounting for: - The sfDelegate field's presence. - The ltDELEGATE object's permissions. - Fee source changes when delegation is active.

Severity and Impact Summary

Financial risks due to balance miscalculations (e.g., users over-credited if fees are ignored, or liquidity shortfalls if delegate balances drain unexpectedly). Operational disruptions from misauthorized transactions (e.g., blocked valid transactions or approved invalid ones).

Recommendation

Explicit Documentation: Clearly state that delegated transactions shift fee liability to the delegate account. - Provide examples of transaction traces with sfDelegate and ltDELEGATE logic.

Integration Guidance: Advise systems to check for `sfDelegate` in transactions and query `ltDELEGATE` entries during reconciliation. - Track both sender and delegate account balances for fee-critical workflows. - Update APIs to surface delegate metadata (e.g., fee_payer field indicating the delegate account).

Monitoring Tools: Add alerts for delegate account balance thresholds to prevent liquidity issues. - Include delegate permissions in transaction audit logs for compliance checks.

Bridge/Exchange Communication: Proactively notify integrators about delegation logic changes via developer portals, newsletters, or dedicated advisories.

The DelegateSet SLE can be created empty

Finding ID: FYEO-XRPL-04
Severity: **Informational**
Status: **Remediated**

Description

In the `DelegateSet::doApply()` function, the code handles an existing delegate ledger object by checking if the permissions array is empty and, if so, deleting the ledger object. However, if no delegate ledger object exists yet and the incoming permissions array is empty, the code proceeds to create a new delegate ledger entry with an empty permissions field. This behavior may not be intended because an empty permissions set is typically used to revoke delegation. By creating a new ledger object with an empty array, the system might unnecessarily store an object that represents no active permissions, which could lead to clutter in the ledger or ambiguous state interpretation.

Proof of Issue

File name: src/xrpld/app/tx/detail/DelegateSet.cpp
Line number: 93

```
auto sle = ctx_.view().peek(delegateKey);
if (sle)
{
    auto const& permissions = ctx_.tx.getFieldArray(sfPermissions);
    if (permissions.empty())
        // if permissions array is empty, delete the ledger object.
        return deleteDelegate(view(), sle, account_, j_);

    sle->setFieldArray(sfPermissions, permissions);
    ctx_.view().update(sle);
    return tesSUCCESS;
}

STAmount const reserve{ctx_.view().fees().accountReserve(
    sleOwner->getFieldU32(sfOwnerCount) + 1)};

if (mPriorBalance < reserve)
    return tecINSUFFICIENT_RESERVE;

sle = std::make_shared<SLE>(delegateKey);
sle->setAccountID(sfAccount, account_);
sle->setAccountID(sfAuthorize, authAccount);
auto const& permissions = ctx_.tx.getFieldArray(sfPermissions);
sle->setFieldArray(sfPermissions, permissions);
```

Severity and Impact Summary

While this issue does not pose an immediate security threat, it can result in the unnecessary creation of ledger objects that serve no functional purpose. Maintaining such objects in the ledger could lead to increased storage usage and potential confusion in state interpretation.

Recommendation

It is recommended to add an additional check for empty permissions in the case where no delegate ledger object exists. If the permissions set is empty, the function should fail or return success without creating a new ledger entry. This change would align the behavior with the intended semantics of delegation revocation and prevent the ledger from being cluttered with objects that do not confer any actual permissions.

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.

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