// Security Assessment 01.06.2025 - 01.30.2025

Ripple - Smart Contract Audit - Credentials

Ripple

HALBORN

Ripple - Smart Contract Audit - Credentials - Ripple

Prepared by: HALBORN

Last Updated 06/27/2025

Date of Engagement: January 6th, 2025 - January 30th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	2	2	4

TABLE OF CONTENTS

1. Introduction

`Ripple` engaged Halborn to conduct a security assessment on XRP Ledger (XRPL) feature amendments beginning on January 6th, 2025 and ending on January 30th, 2025, focusing on modifications to the codebase between the commit 4c2e6a3 and the commit cd9b5c9. The review specifically targets **changes introduced during these commits**, assuming the validity of the pre-existing logic and excluding verification of its security, e.g., input validation of previously existing parameters.

The **Credentials** feature introduced new transaction validation mechanisms and structures that allow credential-based authorization within the Ripple network. This feature enables accounts to issue, manage, and verify credentials for transaction approval, adding an additional layer of security and control.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to assess the security of the node. The security engineer is a blockchain and smart-contract security expert in advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the new features operate as intended.
- Identify potential security issues with the new features.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly acknowledged by the Ripple team. The main ones were the following:

- Ensure that expired NFT offers and credentials are handled separately by using distinct error codes.
- Explicitly reject transactions signed with a disabled Master Key, even if it is also set as the Regular Key.
- Replace assertions with explicit error handling that operates in all build configurations.
- Include a check that confirms whether the recipient has configured Deposit Auth before invoking verifyDepositPreauth.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the node security assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual code review and walkthrough to identify any logic issue.
- Graphing out functionality and contract logic/connectivity/functions.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE

Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9

Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: rippled

(b) Assessed Commit ID: cd9b5c9

(c) Items in scope:

- src/libxrpl/protocol/ErrorCodes.cpp
- src/libxrpl/protocol/Indexes.cpp
- src/libxrpl/protocol/InnerObjectFormats.cpp
- src/libxrpl/protocol/TER.cpp
- src/xrpld/app/main/Main.cpp
- src/xrpld/app/misc/CredentialHelpers.cpp
- src/xrpld/app/tx/detail/applySteps.cpp
- src/xrpld/app/tx/detail/Credentials.cpp
- src/xrpld/app/tx/detail/DeleteAccount.cpp
- src/xrpld/app/tx/detail/DepositPreauth.cpp
- src/xrpld/app/tx/detail/Escrow.cpp
- src/xrpld/app/tx/detail/InvariantCheck.cpp
- src/xrpld/app/tx/detail/PayChan.cpp
- src/xrpld/app/tx/detail/Payment.cpp
- src/xrpld/app/tx/detail/Transactor.cpp
- src/xrpld/net/detail/RPCCall.cpp
- src/xrpld/rpc/detail/RPCHelpers.cpp
- src/xrpld/rpc/handlers/DepositAuthorized.cpp
- src/xrpld/rpc/handlers/LedgerEntry.cpp

Out-of-Scope: The review specifically targets changes introduced during the commit 4c2e6a3 and the commit cd9b5c9, assuming the validity of the pre-existing logic and excluding verification of its security, e.g., input validation of previously existing parameters.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	2	2

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT HANDLING OF EXPIRED CREDENTIALS AND NFT OFFERS	MEDIUM	RISK ACCEPTED - 02/18/2025
MASTER KEY PERSISTENCE ALLOWS TRANSACTIONS DESPITE BEING DISABLED	MEDIUM	RISK ACCEPTED - 01/28/2025
ASSERTIONS MAY LEAD TO INCONSISTENT BEHAVIOR IN PRODUCTION	LOW	RISK ACCEPTED - 02/18/2025
INCONSISTENT CREDENTIAL HANDLING WHEN DEPOSIT AUTH IS NOT CONFIGURED	LOW	RISK ACCEPTED - 02/18/2025

INCORRECT ERROR CODE FOR DUPLICATE CREDENTIAL ENTRIES	INFORMATIONAL	ACKNOWLEDGED - 03/16/2025
MISLEADING ERROR HANDLING FOR INVALID LEDGER INDEX VALUES	INFORMATIONAL	SOLVED - 03/11/2025
EXPIRATION VALIDATION IN DOAPPLY RATHER THAN PRECLAIM	INFORMATIONAL	ACKNOWLEDGED - 02/18/2025
INABILITY TO REPLACE AN EXPIRED CREDENTIAL DUE TO DUPLICATE CHECKS	INFORMATIONAL	ACKNOWLEDGED - 02/26/2025

7. FINDINGS & TECH DETAILS

7.1 INCORRECT HANDLING OF EXPIRED CREDENTIALS AND NFT OFFERS

// MEDIUM

Description

In the `Transactor::operator()` function, the transaction processing logic **treats expired NFT offers and expired credentials as the same condition** (`tecEXPIRED`). The following assignments cause this issue:

```
929 bool const doNFTokenOffers = (result == tecEXPIRED);
930 bool const doCredentials = (result == tecEXPIRED);
```

Since both flags are set based on the same transaction result (`tecEXPIRED`), this could lead to cases where a transaction processing expired credentials unintentionally affects NFT offers, or vice versa.

While the `Ripple team` has clarified that expired credentials or NFT offers are only deleted if they are explicitly referenced in the transaction, this logic still poses **a risk of unintended interactions, inconsistencies, or future regressions** if transaction handling changes. Furthermore, the `Ripple team` has acknowledged a known bug that prevents expired NFT offers from being deleted, meaning the behavior may change once the bug is fixed.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (5.0)

Recommendation

It is recommended to ensure that expired NFT offers and credentials are handled separately by using distinct error codes, e.g.: `tecEXPIRED_NFT` and `tecEXPIRED_CREDENTIAL` .

Remediation Comment

RISK ACCEPTED: The **Ripple team** accepted this risk and stated the following:

> It can't happen because candidates to deletion are added by the transaction in `doApply()` to the special list. This code will always delete only what TX ask to delete.

7.2 MASTER KEY PERSISTENCE ALLOWS TRANSACTIONS DESPITE BEING DISABLED

// MEDIUM

Description

An account that previously set its **Regular Key equal to its Master Key** before the activation of the `fixMasterKeyAsRegularKey` amendment can still sign transactions using the Master Key **even after explicitly disabling it**. This occurs because the Master Key remains valid through the Regular Key mechanism.

This issue affects all transaction types, as accounts that believe they have **disabled their Master Key** can still sign transactions unintentionally. **For example, in credential management transactions**, an account that has disabled its Master Key may still sign `CredentialCreate`, `CredentialAccept`, or `CredentialDelete` transactions, despite assuming it has removed its ability to authorize such actions.

Because `Transactor::checkSingleSign()` does not properly reject a disabled Master Key when it is also used as a Regular Key, affected accounts may unknowingly retain unauthorized signing capabilities:

```
512     if (ctx.view.rules().enabled(fixMasterKeyAsRegularKey))
513     {
514         // Signed with regular key.
521
```

```
522             // Signed with enabled mater key.
523             if (!isMasterDisabled && idAccount == idSigner)
524             {
525                 return tesSUCCESS;
526             }
527 
528             // Signed with disabled master key.
529             if (isMasterDisabled && idAccount == idSigner)
530             {
531                 return tefMASTER_DISABLED;
532             }
533 
534             // Signed with any other key.
535             return tefBAD_AUTH;
536         }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:M/Y:M (5.0)

Recommendation

It is recommended to modify `Transactor::checkSingleSign` to **explicitly reject transactions signed with a disabled Master Key**, even if it is also set as the **Regular Key**. The validation should ensure that once the **Master Key** is disabled (via `lsfDisableMaster`), it **cannot be used for signing in any way**, preventing unintended access through the **Regular Key** mechanism. However, to allow affected legacy accounts to recover, `SetRegularKey` transactions should be exempted from this restriction.

Remediation Comment

RISK ACCEPTED: The **Ripple team** accepted this risk and stated the following:

> *Only 23 accounts out of approximately 6 million are affected by this issue, so the complexity of implementing the solution, including the necessary exception, may not be justified.*

7.3 ASSERTIONS MAY LEAD TO INCONSISTENT BEHAVIOR IN PRODUCTION

// LOW

Description

The codebase extensively uses `assert()` statements, which are **disabled in production builds** when `NDEBUG` is defined. This can result in **silent failures**, **undefined behavior**, and **inconsistent ledger states** since conditions enforced during development may not be validated in production. Additionally, assertions are used in transaction processing, ledger operations, and security-sensitive functions, which can introduce **unintended side effects** when they are omitted.

For example, the following assertion ensures a pointer is not null, but in a production build, it would be skipped, potentially leading to a segmentation fault or undefined behavior:

```
368 TER
369 Transactor::consumeSeqProxy(SLE::pointer const& sleAccount)
370 {
372     SeqProxy const seqProx = ctx_.tx.getSeqProxy();
373     if (seqProx.isSeq())
374     {
375         // Note that if this transaction is a TicketCreate,
376         // then the transaction will modify the account root
377         // sfSequence yet again.
379         return tesSUCCESS;
380     }
381     return ticketDelete(
382         view(), account_, getTicketIndex(account_, seqProx), j_);
383 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (3.4)

Recommendation

Assertions should be replaced with explicit error handling that operates in all build configurations. Instead of using `assert()`, apply conditionals with proper logging and error responses. For instance, use `if (!condition) { return errorCode; }` along with appropriate logging mechanisms to ensure that failures are captured and handled correctly.

Remediation Comment

RISK ACCEPTED: The **Ripple team** accepted this risk and stated the following:

> We don't know why these asserts are left in the code, may be for some debug purpose. Their presence is useless, the code is constructed in such a way that those asserts will never fire (nor debug either release).

7.4 INCONSISTENT CREDENTIAL HANDLING WHEN DEPOSIT AUTH IS NOT CONFIGURED

// LOW

Description

When **Deposit Auth** is not configured for a recipient, transactions that include credentials behave inconsistently across multiple functions. The functions `DeleteAccount::doApply()`, `EscrowFinish::doApply()`, `PayChanClaim::doApply()`, and `Payment::doApply()` validate if `featureDepositAuth` is enabled by checking `ctx_.view().rules().enabled(featureDepositAuth)` before invoking `verifyDepositPreauth`. However, these functions **do not verify whether the recipient has actually configured Deposit Auth**.

As a result:

- If valid credentials are included, the transaction succeeds.
- If expired or invalid credentials are included, the transaction fails.

This partial validation leads to unintended behavior, where credentials influence transaction outcomes even when **Deposit Auth** is not set by the recipient:

```
if (ctx_.view().rules().enabled(featureDepositAuth)) {
    // Called without confirming if Deposit Auth
    // is configured by the recipient
    verifyDepositPreauth(...);
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:L/Y:N (2.1)

Recommendation

Update the functions `DeleteAccount::doApply()`, `EscrowFinish::doApply()`, `PayChanClaim::doApply()`, and `Payment::doApply()` to include a check that confirms whether the recipient has configured **Deposit Auth** before invoking `verifyDepositPreauth`. If **Deposit Auth** is not configured, credentials should be ignored entirely to ensure consistent transaction behavior.

Remediation Comment

RISK ACCEPTED: The **Ripple team** accepted this risk and stated the following:

> It's intentional, for a better user experience.

7.5 INCORRECT ERROR CODE FOR DUPLICATE CREDENTIAL ENTRIES

// INFORMATIONAL

Description

The function `authorized` in the **CredentialHelpers.cpp** incorrectly returns `tefINTERNAL` when a transaction includes duplicate credential entries with the same `(Issuer, CredentialType)`. This error code is meant for internal failures but is being used for a user-input validation failure, which can mislead users and make debugging difficult. The appropriate response should be `tefBAD_LEDGER`, which indicates an unexpected state in the ledger caused by invalid transaction data.

At the end of the function, the issue occurs when inserting credentials into the `std::set`, where a duplicate entry results in `tefINTERNAL`:

```
199 auto [it, ins] =
200     sorted.emplace((*sleCred)[sfIssuer],
201         (*sleCred)[sfCredentialType]);
202 if (!ins)
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

The function should return `tefBAD_LEDGER` instead of `tefINTERNAL` when detecting duplicate credential entries. This ensures that the error properly reflects a ledger state issue rather than an internal failure, improving debugging clarity and user feedback.

Remediation Comment

ACKNOWLEDGED: The **Ripple team** acknowledged this issue and stated the following:

> It returns `tefINTERNAL` because this function is only used in the apply stage, where the incoming credentials array has already been checked for duplicates. `tefINTERNAL` represents an impossible scenario that should never occur, so this behavior is expected and correct.

7.6 MISLEADING ERROR HANDLING FOR INVALID LEDGER INDEX VALUES

// INFORMATIONAL

Description

The `jvParseLedger` function in **RPCCall.cpp** fails to properly handle invalid `ledger_index` values, defaulting to `0` instead of rejecting them. When a user provides an out-of-range or malformed ledger index—such as `-1` or `4294967297` —the function attempts to convert it using `beast::lexicalCast<std::uint32_t>`. If the conversion fails, it silently assigns `0`, which does not exist in the XRP Ledger:

```
97      // TODO New routine for parsing ledger parameters,
98      // other routines should standardize on this.
99      static bool
100     jvParseLedger(Json::Value& jvRequest,
101         std::string const& strLedger)
```

```
102     {
103         if (strLedger == "current" || strLedger == "closed" ||
104             strLedger == "validated")
105         {
106             jvRequest[jss::ledger_index] = strLedger;
107         }
108         else if (strLedger.length() == 64)
109         {
110             // YYY Could confirm this is a uint256.
111             jvRequest[jss::ledger_hash] = strLedger;
112         }
113         else
114         {
115             jvRequest[jss::ledger_index] =
116                 beast::lexicalCast<std::uint32_t>(strLedger);
117         }
118
119         return true;
120     }
```

This behavior results in misleading error messages when transactions or queries reference an invalid ledger, complicating debugging and user interactions. While this does not pose a direct security concern, it can lead to confusion and inefficient troubleshooting due to inaccurate error reporting.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to explicitly validate `ledger_index` values, ensuring they are within the allowed range (starting from `1`). If an invalid value is detected, return a descriptive error instead of defaulting to `0`.

Remediation Comment

SOLVED: The **Ripple team** solved the issue in the specified PR.

7.7 EXPIRATION VALIDATION IN DOAPPLY RATHER THAN PRECLAIM

// INFORMATIONAL

Description

The code checks whether a credential is expired **only** in the `doApply()` phase of certain transactions (e.g., `CredentialCreate::doApply` and `CredentialAccept::doApply`), but does **not** verify expiration during `preclaim`. As a result, transactions involving an expired credential are allowed to pass `preclaim`, only to fail later in `doApply`. This inconsistency consumes validator and node resources unnecessarily and delays rejection of invalid transactions.

An attacker (or careless user) could submit numerous transactions referencing obviously expired credentials. Because no expiration check is done in `preclaim`, these transactions are placed in the nodes mempools. Although each transaction will eventually fail in `doApply`, it still requires processing overhead, adds queue latency, and costs the transaction fee payer extra XRP.

Code Location

- Expiration Check in `CredentialAccept::doApply`: the check happens only at the final stage.

```
365 if (checkExpired(sleCred, view().info().parentCloseTime))
366 {
367     JLOG(j_.trace()) << "Credential is expired: "
368         << sleCred->getText();
369     // delete expired credentials even if the transaction failed
370     auto const err = credentials::deleteSLE(view(),
371         sleCred, j_);
372     return isTesSuccess(err) ? tecEXPIRED : err;
373 }
```

- Expiration Check in `CredentialCreate::doApply`: the check happens only at the final stage.

```
132 if (closeTime > *optExp)
133 {
134     JLOG(j_.trace()) << "Malformed transaction: "
135                         "Expiration time is in the past.";
136     return tecEXPIRED;
137 }
```

BVSS

AO:A/AC:H/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.6)

Recommendation

It is recommended to add an expiration check in `preclaim` for any transactions that depend on non-expired credentials. If the credential is expired, return `tecEXPIRED` (or an analogous code) immediately, preventing it from entering the transaction queue in the first place. This ensures prompt rejection of invalid transactions, saving network and node resources.

Remediation Comment

ACKNOWLEDGED: The **Ripple team** acknowledged this issue and stated the following:

> It is because of deletion of the expired credentials (which is not possible in the preclaim).

7.8 INABILITY TO REPLACE AN EXPIRED CREDENTIAL DUE TO DUPLICATE CHECKS

// INFORMATIONAL

Description

When creating a credential (`CredentialCreate`), the logic checks for duplicates of the same credential type on the same subject, returning `tecDUPLICATE` if one already exists. However, it does not check whether the existing credential is expired. Consequently, an expired credential with the same type blocks the creation of a newer, valid credential. This can lead to a situation in which the subject is effectively unable to refresh or replace an expired credential without manual credential deletion.

Code Location

- In `CredentialCreate` : the function refuses to recreate a credential if it already exists, not checking if the current credential was expired.

```
104 | if (ctx.view.exists(
105 |     keylet::credential(subject, ctx.tx[sfAccount], credType)))
106 | {
107 |     JLOG(ctx.j.trace()) << "Credential already exists.";
108 |     return tecDUPLICATE;
109 | }
```

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.3)

Recommendation

During the credential creation, when detecting an preexisting credential, it is recommended to verify whether it is expired, and in that case allow the new credential to overwrite or replace the old one without requiring explicit deletion.

Remediation Comment

ACKNOWLEDGED: The **Ripple team** acknowledged this issue and stated the following:

> *It* is not recognized as bug, as it implemented as designed. However it seems like a feature that worth to discuss and we may return to it.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.