XRPL Integration Security Assessment

Axelar
Version 1.2 – January 7, 2025

Prepared By
Kevin Henry
Parnian Alimi

Prepared For
Nikolaos Kamarinakis

1 Executive Summary

Synopsis

During the December of 2024, Axelar engaged NCC Group to conduct a security assessment of the XRP ledger's integration with the Axelar Amplifier protocol. This project provides the necessary support for bridging tokens between XRP and connected chains on the Axelar network. Two consultants conducted this review in 8 person-days, over the course of two weeks.

A detailed description of the scope and the codebase was provided at the kick-off. The Axelar and Common Prefix teams answered a number of questions asked by NCC Group consultants in Slack.

NCC Group performed a follow-up retest after this initial engagement and updated the findings' ratings accordingly. NCC Group also found that the majority of the engagement notes had been appropriately addressed.

Scope

NCC Group's evaluation included the following paths from the https://github.com/commonprefix/axelar-amplifier repository on commit `cfb4395`:

- XRPL-specific ampd signing and verification off-chain components:
  - https://github.com/commonprefix/axelar-amplifier/tree/xrpl/ampd/src/xrpl
  - https://github.com/commonprefix/axelar-amplifier/blob/xrpl/ampd/src/handlers/xrpl_multisig.rs
  - https://github.com/commonprefix/axelar-amplifier/blob/xrpl/ampd/src/handlers/xrpl_verify_msg.rs
- XRPL Multisig Prover: https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-multisig-prover
- XRPL Gateway: https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-gateway
- XRPL Voting Verifier: https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-voting-verifier
- XRPL types: https://github.com/commonprefix/axelar-amplifier/tree/xrpl/packages/xrpl-types

PR #1 was used, in the follow up retesting phase, to assess the implemented recommendations and update this report.

Limitations

The NCC Group team focused their review on the scope above, paying attention to various XRP ledger's concepts (documented at https://xrpl.org/docs) and their interaction with the Amplifier protocol.

Key Findings

The NCC Group team uncovered a total of 8 findings, among which the most notable were:

- Finding "Incomplete Dust Accounting can Lead to Parallel Dust Claims" warns that multiple parallel dust claims are possible. This finding was updated during retesting to indicate that it was a false positive. The follow up discussions are included in finding "Using Conflicting Transaction Sequence Numbers".
- Finding "Inconsistent Error Handling in Dust Amount Arithmetic" notes that the dust payment amount types handle arithmetic overflow differently. This finding is considered to be fixed once PR #1 is merged.

Strategic Recommendations

After addressing the findings presented in this review, NCC Group recommends running extensive integration tests to ensure robust handling of various cross-chain payments and GMP message passings, to and from the XRP ledger. In addition, the serialization and decimal conversion implementations are great candidates for fuzz testing. Lastly, Clippy should be used to catch arithmetic operations that can potentially result in unexpected side-effects and to improve the Rust code's quality.

2 Dashboard

Target Data

Name	Axelar XRPL Integration
Type	CosmWasm Contracts, Amplifier Daemon (Ampd)
Platforms	Rust
Environment	Smart Contracts, Local Nodes

Engagement Data

Type	Security Assessment
Method	Code-assisted
Dates	2024-12-10 to 2024-12-20
Consultants	2
Level of Effort	8 person-days

Targets

XRPL-specific ampd signing and verification off-chain components	• https://github.com/commonprefix/axelar-amplifier/tree/xrpl/ampd/src/xrpl • https://github.com/commonprefix/axelar-amplifier/blob/xrpl/ampd/src/handlers/xrpl_multisig.rs • https://github.com/commonprefix/axelar-amplifier/blob/xrpl/ampd/src/handlers/xrpl_verify_msg.rs
XRPL Multisig Prover	https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-multisig-prover
XRPL Gateway	https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-gateway
XRPL Voting Verifier	https://github.com/commonprefix/axelar-amplifier/tree/xrpl/contracts/xrpl-voting-verifier
XRPL types	https://github.com/commonprefix/axelar-amplifier/tree/xrpl/packages/xrpl-types

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	0	
Low issues	4	🟨🟨🟨🟨
Informational issues	3	⬜⬜⬜
Total issues	**7**	

Category Breakdown

Data Validation	2	🟨🟨
Error Reporting	2	🟨🟨
Patching	1	⬜
Session Management	2	⬜⬜

Component Breakdown

axelar-amplifier	1	⬜
verifier-node	1	🟨
xrpl-multisig-prover	5	🟨🟨🟨⬜⬜

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Incomplete Dust Accounting can Lead to Parallel Dust Claims	False Positive	YXX	Medium
Contract Panics During Type Conversion	False Positive	K3F	Low
Inconsistent Error Handling in Dust Amount Arithmetic	Fixed	HQN	Low
Potentially Incorrect Quorum Computation	Fixed	TUH	Low
Improved Error Handling During Serialization	Fixed	2HB	Low
Corrupt Message will Crash the Verifier Node	Fixed	6PR	Low
Dependencies with Known RustSec Advisories	Reported	C44	Info
Fragile Transaction Status Reporting	Updated	3YV	Info
Using Conflicting Transaction Sequence Numbers	New	A9X	Info

4 Finding Details

Medium

Incomplete Dust Accounting can Lead to Parallel Dust Claims

Overall Risk	Medium	**Finding ID**	NCC-E010021-XRPL-YXX
Impact	Medium	**Component**	xrpl-multisig-prover
Exploitability	Medium	**Category**	Session Management
		Status	False Positive

Impact

Failure to properly manage dust accounting can lead to multiple parallel dust claims.

Description

The `ClaimDust` instruction, which can only be invoked by the admin or governance accounts, is used to transfer dust of a given token type, to a destination address on the XRP ledger. This feature is implemented in the `construct_dust_claim_payment_proof()` function. It first loads the dust from storage, converts it to an XRPL amount, issues a payment to the destination address, and finally starts a signing session:

```
pub fn construct_dust_claim_payment_proof(
    storage: &mut dyn Storage,
    querier: &Querier,
    self_address: Addr,
    destination_address: XRPLAccountId,
    token_id: TokenId,
    chain: ChainNameRaw,
) -> Result<Response, ContractError> {
    let config = state::CONFIG.load(storage).map_err(ContractError::from)?;

    let current_dust = DUST.load(storage, &(token_id.clone(), chain.clone()))
        .map_err(|_| ContractError::DustNotFound)?;

    if current_dust.is_zero() {
        return Err(ContractError::DustAmountTooSmall {
            dust: current_dust,
            token_id,
            chain,
        });
    }

    let (claimable_dust, updated_dust) = match current_dust.clone() {
        DustAmount::Local(current_local_dust) => {
            assert!(chain == config.chain_name, "local dust stored under invalid chain name");
            (current_local_dust.clone(), DustAmount::Local(current_local_dust.zeroize()))
        }
        DustAmount::Remote(current_remote_dust) => {
            let (claimable_dust, new_dust) = compute_xrpl_amount(
                querier,
                token_id.clone(),
                chain.clone(),
                current_remote_dust.clone(),
            )?;

            (claimable_dust, DustAmount::Remote(current_remote_dust - new_dust))
        }
    };

    if claimable_dust.is_zero() {
        return Err(ContractError::DustAmountTooSmall {
            dust: current_dust,
            token_id,
            chain,
        });
    }

    let unsigned_tx_hash = xrpl_multisig::issue_payment(
        storage,
        &config,
        destination_address,
        &claimable_dust,
        None,
        None,
    )?;

    state::UNSIGNED_TX_HASH_TO_DUST_INFO.save(storage, &unsigned_tx_hash, &DustInfo {
        token_id,
        chain,
        dust_amount: updated_dust,
    })?;

    Ok(Response::new().add_submessage(start_signing_session(storage, &config,
    ↪ unsigned_tx_hash, self_address, None)?))
}
```

Note that the Multisig Prover contract does not update the `DUST` storage, and therefore until the payment is finalized in `confirm_tx_status()`, depicted below, the admin or the governance accounts can create another claim for this dust:

```
XRPLUnsignedTx::Payment(_) => {
    // Do nothing further if TX is not dust claim.
    if tx_info.original_cc_id.is_some() {
        return Ok(None);
    }

    let dust_info = UNSIGNED_TX_HASH_TO_DUST_INFO.load(storage, &unsigned_tx_hash)?;
    DUST.update(
        storage,
        &(dust_info.token_id, dust_info.chain),
        |current_dust| -> Result<DustAmount, ContractError> {
            match current_dust {
                Some(current_dust) => current_dust.sub(dust_info.dust_amount),
                None => panic!("dust amount must be set"),
            }
        }
    )?;
    None
}
```

Depending on the MultiSig Prover contract's token balances, multiple parallel dust claims for the same token may or may not succeed. If it does succeed the contract will be spending funds that it did not accumulate from dusts; as such, the severity of this finding is set to medium.

Recommendation

Implement a mechanism to track ongoing dust claims and prevent multiple concurrent dust claims for the same token.

Location

https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/contracts/xrpl-multisig-prover/src/contract/execute.rs#L238-L300

Retest Results

2025-01-06 – New

The client pointed out that while there is a pending dust claim, subsequent dust claims will use the same sequence number (see `next_sequence_number()`'s logic). Since only one of these transactions will be validated and recorded on the ledger, double-spending is not possible.

See finding "Using Conflicting Transaction Sequence Numbers" for follow up discussions regarding reusing sequence numbers.

Client Response

Client indicated that “[t]he multisig’s sequence number will only be incremented once the dust claim TX is confirmed. So if you try to issue another dust claim before the 1st one was confirmed, it will be assigned the same sequence number”.

Low

Contract Panics During Type Conversion

Overall Risk	Low	**Finding ID**	NCC-E010021-XRPL-K3F
Impact	Low	**Component**	xrpl-types
Exploitability	Low	**Category**	Data Validation
		Status	False Positive

Impact

Converting an `AxelarSigner` type to a `Participant` panics if the signer's weight is 0.

Description

The `AxelarSigner` to `Participant` type conversion is implemented below:

```
impl From<AxelarSigner> for Participant {
    fn from(signer: AxelarSigner) -> Self {
        let weight = nonempty::Uint128::try_from(Uint128::from(u128::from(signer.weight))).unwr
        ↳ ap();
        Self {
            address: signer.address,
            weight,
        }
    }
}
```

The `nonempty::Uint128` type expects a non-zero value, as such the highlighted `unwrap()` will cause a panic if the Axelar signer's weight is 0. The list of Axelar signers is queried from the service registry in the `active_verifiers()` API:

```
pub fn active_verifiers(
    querier: &Querier,
    signing_threshold: MajorityThreshold,
    block_height: u64,
) -> Result<VerifierSet, ContractError> {
    let verifiers: Vec<WeightedVerifier> = querier.active_verifiers()?;

    let verifiers_with_pubkeys = verifiers
        .into_iter()
        .filter_map(|verifier| {
            let address = verifier.verifier_info.address.clone();
            querier.public_key(address.to_string())
            .ok()
            .map(|pk| (verifier, pk))
        })
        .collect::<Vec<_>>();

    let num_of_verifiers = verifiers_with_pubkeys.len();
    if num_of_verifiers > MAX_NUM_XRPL_MULTISIG_SIGNERS {
        return Err(ContractError::TooManyVerifiers);
    }

    let service = querier.service()?;
    if num_of_verifiers < service.min_num_verifiers.try_into().expect("minimum number of
    ↳ verifiers is too large") {
        return Err(ContractError::NotEnoughVerifiers);
    }

    let weights = convert_or_scale_weights(
        verifiers_with_pubkeys
            .clone()
            .iter()
            .map(|(verifier, _)| Uint128::from(verifier.weight))
            .collect::<Vec<Uint128>>()
            .as_slice(),
    )?;

    let mut signers: Vec<AxelarSigner> = vec![];
    for (i, (verifier, pub_key)) in verifiers_with_pubkeys.iter().enumerate() {
        signers.push(AxelarSigner {
            address: verifier.verifier_info.address.clone(),
            weight: weights[i],
            pub_key: pub_key.clone(),
        });
    }

    let sum_of_weights: u32 = weights.iter().map(|w| u32::from(*w)).sum();

    let quorum = u32::try_from(
        u64::from(sum_of_weights)
            .checked_mul(signing_threshold.numerator().into())
            .unwrap()
            .checked_div(signing_threshold.denominator().into())
            .unwrap(),
    )
    .unwrap();

    let verifier_set = VerifierSet {
        signers: BTreeSet::from_iter(signers),
        quorum,
        created_at: block_height,
    };

    Ok(verifier_set)
}
```

Figure 1: Excerpt from `contracts/xrpl-multisig-prover/src/axelar_verifiers.rs`

The `active_verifiers()` API should reject a list that contains verifiers with zero weights. Since service registry is a trusted entity, and should not send corrupt input during normal operation, this finding is marked as low severity.

Recommendation

Update `active_verifiers()` to return an error when any of the signers has a 0 weight.

Carefully assess all the possible panic scenarios and ensure that a user's input cannot trigger them.

Location

https://github.com/commonprefix/axelar-amplifier/blob/392f49a88e63b829ae0d6af595524 2e9c762d4bd/packages/xrpl-types/src/types.rs#L63

Retest Results

2025-01-06 – New

The query to get the list of active verifiers from the service registry (line 88 in the second snippet above) returns a vector of the type `WeightedVerifier`, which stores `weight` as a `nonempty::Uint128` type. As such, if any of the weights are 0, the assignment on line 88 will return an error, and the `unwrap()` call in the first snippet above, will not be triggered.

It is worth noting that these verifiers' weights are scaled on line 110 in the `active_verifiers()` function, which does not produce zeros (see `convert_or_scale_weights()` for details).

Although as the code stands, it is not possible to trigger the panic in the `AxelarSigner` to `Participant` conversion, the verifiers' weights' type is converted from `nonempty::Uint128` to `u16` to `u128`, and back, to `nonempty::Uint128`. This could have been avoided by using the `nonempty::Uint128` type everywhere.

Low

Inconsistent Error Handling in Dust Amount Arithmetic

Overall Risk	Low	**Finding ID**	NCC-E010021-XRPL-HQN
Impact	Medium	**Component**	xrpl-multisig-prover
Exploitability	Low	**Category**	Error Reporting
		Status	Fixed

Impact

Inconsistent error handling leads to failure codes in one case and panics in another.

Description

In the following code snippet from the `DustAmount` implementation, 2 types of amounts are handled:

1. A local dust amount in `XRPLPaymentAmount`, and
2. a remote dust amount in CosmWasm's `Uint256`.

```rust
pub fn sub(self, other: Self) -> Result<Self, ContractError> {
    match (self, other) {
        (DustAmount::Local(a), DustAmount::Local(b)) => Ok(DustAmount::Local(a.sub(b)?)),
        (DustAmount::Remote(a), DustAmount::Remote(b)) => Ok(DustAmount::Remote(a - b)),
        _ => panic!("cannot subtract local and remote dust amounts"),
    }
}
```

The `XRPLPaymentAmount` implements checked arithmetic[1], whereas the `Uint256`'s implementation uses strict subtraction which will panic[2] if an overflow is detected.

This note also applies to the `DustAmount`'s `add()` implementation.

Recommendation

Consider using checked arithmetic with the `Uint256` type.

Location

https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/contracts/xrpl-multisig-prover/src/state.rs#L89-L103

Retest Results

2025-01-02 – Fixed

As part of PR #1, commit `cc2d13e`, the `add()` and `sub()` functions were updated to use checked arithmetic, as recommended. Once the linked PR is merged, this finding is considered "Fixed".

1. See details at *packages/xrpl-types/src/types.rs*.
2. See implementation at *cosmwasm_std/math/uint256.rs.html*.

Low

Potentially Incorrect Quorum Computation

Overall Risk	Low	**Finding ID**	NCC-E010021-XRPL-TUH
Impact	Medium	**Component**	xrpl-multisig-prover
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

The XRPL multisig-prover may operate with a signer quorum different from that used by the Axelar multisig-prover, potentially leading to diverging definitions of message validity.

Description

The function `active_verifiers()` builds a `VerifierSet` based on a signing threshold. In order to build this set, the quorum of total weights required must be computed from the set of signers, their respective weights, and the specified `MajorityThreshold`:

```
    let sum_of_weights: u32 = weights.iter().map(|w| u32::from(*w)).sum();

    let quorum = u32::try_from(
        u64::from(sum_of_weights)
            .checked_mul(signing_threshold.numerator().into())
            .unwrap()
            .checked_div(signing_threshold.denominator().into())
            .unwrap(),
    )
    .unwrap();
```

Figure 2: contracts/xrpl-multisig-prover/src/axelar_verifiers.rs

In short, the quorum is computed by taking the total weight multiplied by the numerator and then divided by the denominator.

In the current use cases, all signers have a weight of 1 and the threshold is effectively given as the number of required signers over the total number of signers. In this situation, the above calculation will result in a correct integer result for the quorum. However, in the general case, where signers have different weights and the threshold represents a proportion of weights to reach quorum, the above calculation is incorrect and diverges from similar calculations in the codebase, e.g.:

```
let quorum = nonempty::Uint128::try_from(total_weight.mul_ceil(quorum_threshold))
```

Figure 3: packages/axelar-wasm-std/src/snapshot.rs

Notably, the above computes *the ceiling* of the product with the fractional threshold, rather than the integer division result as in `active_verifiers()`. This ensures that rounding occurs in the correct direction to enforce a majority quorum for all potential majority thresholds.

Consider a case where the set of signers contains 6 members and the threshold is set to `(3,5)`. Then `active_verifiers()` will compute the quorum as $\frac{6*3}{5} = 3$ due to integer division instead of $\lceil 6 * \frac{3}{5} \rceil = 4$.

While the existing implementation may produce correct results for all currently intended inputs, it is nevertheless recommended to ensure it is consistent with the same calculation elsewhere in the codebase to ensure the behaviors do not diverge under future use cases.

Recommendation

- Revise the implementation to correctly use the ceiling of the product as is performed elsewhere in the codebase.
- Consider adding additional tests to ensure corner cases are handled correctly.

Location

contracts/xrpl-multisig-prover/src/axelar_verifiers.rs

Retest Results

2025-01-02 – Fixed

As part of PR #1, commit `8028f07`, the quorum computation was updated to use `mul_ceil()`:

```
    let sum_of_weights = weights.iter().map(|w| u64::from(*w)).sum();
    let numerator = u64::from(signing_threshold.numerator());
    let denominator = u64::from(signing_threshold.denominator());
    let quorum = u32::try_from(mul_ceil(sum_of_weights, numerator, denominator)).unwrap();
```

This change aligns the quorum computation to match other instances in the codebase, as recommended. Once the linked PR is merged, this finding is considered "Fixed".

Low

Improved Error Handling During Serialization

Overall Risk	Low	**Finding ID**	NCC-E010021-XRPL-2HB
Impact	Low	**Component**	xrpl-multisig-prover
Exploitability	Low	**Category**	Error Reporting
		Status	Fixed

Impact

Inconsistent or uninformative error handling could increase the difficulty of debugging, or in the worst case, present a vector for denial-of-service attacks.

Description

Potential improvements to error handling during serialization were identified during the review. This finding is related to finding "Inconsistent Error Handling in Dust Amount Arithmetic", which also identified potential improvements with respect to error handling within the multisig prover.

The following code implements serialization for `XRPLPathSet`:

```rust
impl XRPLSerialize for XRPLPathSet {
    const TYPE_CODE: u8 = 18;

    fn xrpl_serialize(&self) -> Result<Vec<u8>, ContractError> {
        assert!(self.paths.len() > 0);
        assert!(self.paths.len() <= 6);
        let mut result: Vec<u8> = Vec::new();
        for (i, path) in self.paths.iter().enumerate() {
            assert!(path.steps.len() > 0);
            assert!(path.steps.len() <= 8);
```

Figure 4: contracts/xrpl-multisig-prover/src/xrpl_serialize.rs

However, despite the fact that `xrpl_serialize()` returns a `Result` with a potential `ContractError`, the function uses assertions to force a panic if the input path set is not well formed. If none of the existing `ContractError` results are suitable in this case, it is recommended to add an additional type such that a more user-friendly descriptive error message can be returned. Such an approach would be better aligned with the rest of the serialization functions, as well as with the Secure Rust Guidelines:

> Explicit error handling (Result) should always be preferred instead of calling panic. The cause of the error should be available, and generic errors should be avoided.

Recommendation

Consider leveraging the `Result` and `ContractError` types to return an informative error instead of panicking during serialization.

Location

contracts/xrpl-multisig-prover/src/xrpl_serialize.rs

Retest Results

2025-01-02 – Fixed

As part of PR #1, commit `bfac90d`, the `ContractError` enum was updated to include errors for `TooManyPaths`, `TooManySteps`, `ZeroPaths`, and `ZeroPathSteps`. The identified assertions/panics have been replaced with these errors, as recommended.

Once the linked PR is merged, this finding is considered “Fixed”.

Low

Corrupt Message will Crash the Verifier Node

Overall Risk	Low	**Finding ID**	NCC-E010021-XRPL-6PR
Impact	Medium	**Component**	verifier-node
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

Failure to gracefully reject invalid messages will result in denial-of-service.

Description

The `verify_memos()` helper function parses a message's memo and validates it. If the memo's type is not encoded as a proper hexadecimal string, the call to `hex::decode()` will return an error and the `unwrap()` will cause a crash:

```
pub fn verify_memos(memos: Vec<Memo>, message: &XRPLUserMessage) -> bool {
    let memo_kv: HashMap<String, String> = memos
        .into_iter()
        .filter(|m| m.memo_type.is_some() && m.memo_data.is_some())
        .map(|m| (String::from_utf8(hex::decode(m.memo_type.unwrap()).expect("Memo value
        ↪ should be hex")).ok(), m.memo_data))
        .filter_map(|(k, v)| {
            match (k, v) {
                (Some(k), Some(v)) => Some((k, v)),
                _ => None,
            }
        })
        .collect();

    || -> Option<bool> {
        Some(
            memo_kv.get("destination_address")? ==
            ↪ &remove_0x_prefix(message.destination_address.to_string()).to_uppercase()
            && memo_kv.get("destination_chain")? ==
            ↪ &hex::encode_upper(message.destination_chain.to_string())
            && hex::decode(&remove_0x_prefix(memo_kv.get("payload_hash")?.clone())).ok()? ==
            ↪ message.payload_hash
        )
    }().unwrap_or(false)
}
```

Since the verifier node is presumably reading the ledger from a server, it is possible that a malicious server would send it a corrupt message to cause it to go offline and cause a denial-of-service.

Recommendation

If the `memo_type` is not of the expected format, `verify_memos()` should gracefully return `false`. Ensure that the verifier node will not crash due to a malformed input.

Location

https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/ampd/src/xrpl/verifier.rs#L95

Retest Results

2025-01-02 – Fixed

As part of PR #1, commit `bc537e8`, the `verify_memos()` function was refactored to avoid the usage of `unwrap()` and to correctly return `false` when decoding fails. Once the linked PR is merged, this finding is considered “Fixed”.

Info

Dependencies with Known RustSec Advisories

Overall Risk	Informational	**Finding ID**	NCC-E010021-XRPL-C44
Impact	Low	**Component**	axelar-amplifier
Exploitability	Low	**Category**	Patching
		Status	Reported

Impact

Vulnerabilities in third-party dependencies may be inherited by the application. Even if known vulnerabilities do not apply, the presence of vulnerable dependencies may still affect the perceived security posture of the application and its maintainers.

Description

The Rust ecosystem provides several tools for managing dependencies, such as `cargo audit` for identifying known security issues, `cargo outdated` for identifying stale dependencies, and `cargo deny` for blocking or allowing various vulnerable or outdated dependencies.

The parent `axelar-amplifier` repository contains the following *vulnerable* crates according to `cargo audit`:

- `derivative 2.2.0`
- `webpki 0.21.4`
- `rustls 0.19.1`, `0.20.9`
- `rsa 0.8.2`
- `idna 0.5.0`
- `curve25519dalek 3.2.0`

It is therefore expected that the reviewed fork would be similarly affected. In addition to the above, the reviewed code contains two additional vulnerable crates:

- `hashbrown 0.15.0`: RUSTSEC-2024-0399 - *"The borsh serialization of the HashMap did not follow the borsh specification. It potentially produced non-canonical encodings dependent on insertion order. It also did not perform canonicty [sic] checks on decoding. This can result in consensus splits and cause equivalent objects to be considered distinct."*
- `rustls 0.23.16`: RUSTSEC-2024-0402 - *"A bug introduced in rustls 0.23.13 leads to a panic if the received TLS ClientHello is fragmented. Only servers that use `rustls::server::Acceptor::accept()` are affected."*

In general, it is recommended to actively address or update any applicable RustSec advisories that affect the application. The `cargo deny` tool can be used to automatically fail builds if a vulnerable crate is detected, and also supports a list of exceptions so that the inclusion of such a package can be explicitly reviewed and annotated.

Based on past engagements, it is understood that Axelar maintains a Slack bot to notify about any affecting RustSec advisories, and prior engagements have demonstrated evidence that this channel is monitored and affecting advisories are actively addressed. As such, this finding is considered informational.

Recommendation

Ensure that all crates with published RustSec advisories are reviewed and patched as needed prior to any major or public releases or development milestones.

Info

Fragile Transaction Status Reporting

Overall Risk	Informational	**Finding ID**	NCC-E010021-XRPL-3YV
Impact	None	**Component**	xrpl-multisig-prover
Exploitability	None	**Category**	Session Management
		Status	Updated

Impact

Failure to return the correct transaction status hinders caller's flow.

Description

The `confirm_tx_status()` API (which handles the `ExecuteMsg::ConfirmTxStatus` message), partially validates the inputted signers' public keys (`signer_public_keys`) against the session's verifier set and then fetches the transaction's status from the voting verifier contract:

```rust
pub fn confirm_tx_status(
    storage: &mut dyn Storage,
    querier: &Querier,
    config: &Config,
    multisig_session_id: &Uint64,
    signer_public_keys: &[PublicKey],
    tx_id: TxHash,
) -> Result<Response, ContractError> {
    let num_signer_public_keys = signer_public_keys.len();
    if num_signer_public_keys == 0 {
        return Err(ContractError::EmptySignerPublicKeys);
    }

    let unsigned_tx_hash =
        state::MULTISIG_SESSION_ID_TO_UNSIGNED_TX_HASH.load(storage,
        ↪multisig_session_id.u64())?;
    let mut tx_info = state::UNSIGNED_TX_HASH_TO_TX_INFO.load(storage, &unsigned_tx_hash)?;
    let multisig_session = querier.multisig(multisig_session_id)?;

    let xrpl_signers = multisig_session
        .verifier_set
        .signers
        .into_iter()
        .filter(|(_, signer)| signer_public_keys.contains(&signer.pub_key))
        .filter_map(|(signer_address, signer)|
        ↪multisig_session.signatures.get(&signer_address).cloned().zip(Some(signer)))
        .map(XRPLSigner::try_from)
        .collect::<Result<Vec<XRPLSigner>, XRPLError>>()?;

    if xrpl_signers.len() != num_signer_public_keys {
        return Err(ContractError::InvalidSignerPublicKeys);
    }

    let signed_tx = XRPLSignedTx::new(tx_info.unsigned_tx.clone(), xrpl_signers);
    let signed_tx_hash = xrpl_types::types::hash_signed_tx(
        signed_tx.xrpl_serialize()?.as_slice(),
    )?;

```

```
98      // Sanity check.
99      if tx_id != signed_tx_hash {
100         return Err(ContractError::TxIdMismatch(tx_id));
101     }
102
103     let message = XRPLMessage::ProverMessage(signed_tx_hash);
104     let status = querier.message_status(message)?;
105
106     match status {
107         VerificationStatus::Unknown |
108         VerificationStatus::FailedToVerify => {
109            return Err(ContractError::TxStatusUnknown);
110         },
111         VerificationStatus::InProgress => {
112            return Err(ContractError::TxStatusVerificationInProgress);
113         },
114         VerificationStatus::SucceededOnSourceChain
115         | VerificationStatus::FailedOnSourceChain
116         | VerificationStatus::NotFoundOnSourceChain => {}
117     }
```

Figure 5: Excerpt from contracts/xrpl-multisig-prover/src/contract/execute.rs

On line 80, the inputted signers list is filtered to only include signers that were part of the session's verifiers set. If the resulting signers list shrinks, i.e., it contains public keys that are not part of the verifiers set, the check on line 89 will return an error. Then on line 94, the signed transaction hash is calculated from the signatures that correspond with the `signer_public_keys`, and not necessarily the entire verifiers set; as such, if the `signer_public_keys` does not contain all the verifiers, the calculated hash will not match the transaction id, in which case the check on line 99 will fail and the function will return.

This sanity check will prevent the logic from fetching the transaction's status from the voting verifier contract. Therefore, if verification is in progress and not all signatures have been received, the `VerificationStatus::InProgress` verification error case on line 111 will be unreachable.

The upshot is that the returned error (`Err(ContractError::TxIdMismatch(tx_id))`) may not convey the transaction's correct status and could lead to confusion for the caller.

Recommendation

Assess whether the sanity check on line 99 is necessary.

Add unit tests for the `confirm_tx_status()` function that cover all possible transaction statuses and error scenarios. Ensure that an honest caller is able to query the transaction status.

Location

https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/contracts/xrpl-multisig-prover/src/contract/execute.rs#L62-L117

Retest Results

2025-01-06 – New

The Common Prefix team provided more context around this finding, and indicated that this API is called by the relayers to confirm the status of the transaction on XRPL after the signing session has ended. As such, the `signer_public_keys` list should contain the exact group that participated in the signing session, and the sanity check on line 99 will signal a corrupted state to the relayer.

The severity of this finding is updated to informational. NCC Group still recommends that more integration tests are included to cover failure scenarios in this API.

Info

Using Conflicting Transaction Sequence Numbers

Overall Risk	Informational	**Finding ID**	NCC-E010021-XRPL-A9X
Impact	Undetermined	**Component**	xrpl-multisig-prover
Exploitability	Undetermined	**Category**	Session Management
		Status	New

Impact

Failure to use valid sequence numbers will result in failed transactions.

Description

In the XRP Ledger design, an account maintains a "Sequence" field which is used to order its transactions. This sequence number is a 32-bit unsigned integer which starts at 1, and is incremented by 1 by the transaction's sender after every transaction submission. An account's sequence number ensures that each transaction is executed exactly once, regardless of whether it fails (with a tec-class error code) or succeeds. Alternatively, accounts can use tickets to submit transactions out of order.

The XRPL MultiSig prover contract supports both the sequence number and the ticketing mechanisms to submit transfers to the XRP Ledger:

```rust
pub fn issue_payment(
    storage: &mut dyn Storage,
    config: &Config,
    destination: XRPLAccountId,
    amount: &XRPLPaymentAmount,
    cc_id: Option<&CrossChainId>,
    cross_currency: Option<&XRPLCrossCurrencyOptions>,
) -> Result<TxHash, ContractError> {
    let sequence = match cc_id {
        Some(cc_id) => XRPLSequence::Ticket(assign_ticket_number(storage, cc_id)?),
        None => XRPLSequence::Plain(next_sequence_number(storage)?),
    };

    let tx = XRPLPaymentTx {
        account: config.xrpl_multisig.clone(),
        fee: config.xrpl_fee,
        sequence,
        amount: amount.clone(),
        destination,
        cross_currency: cross_currency.cloned(),
    };

    issue_tx(storage, XRPLUnsignedTx::Payment(tx), cc_id)
}
```

Figure 6: Excerpt from contracts/xrpl-multisig-prover/src/xrpl_multisig.rs#L48-L71

However, when it submits transactions with the MultiSig account's sequence number (the `next_sequence_number()` call on line 58), it reuses the account's current sequence number when there is a pending transaction:

```
272 fn next_sequence_number(storage: &dyn Storage) -> Result<u32, ContractError> {
273     match load_latest_sequential_tx_info(storage)? {
274         Some(latest_sequential_tx_info)
275             if latest_sequential_tx_info.status == XRPLTxStatus::Pending =>
276             // this might still be pending but another tx with same sequence number may be confirmed!!!
277         {
278             Ok(latest_sequential_tx_info
279                 .unsigned_tx
280                 .sequence()
281                 .into())
282         }
283         _ => NEXT_SEQUENCE_NUMBER.load(storage).map_err(|e| e.into()),
284     }
285 }
```

Figure 7: Excerpt from contracts/xrpl-multisig-prover/src/xrpl_multisig.rs#L272-L285

While, as the XRPL documentation suggests, submitting multiple unconfirmed transactions with the same sender and sequence number is possible, these transactions are mutually exclusive and at most one of them will succeed. Given that the `issue_payment()` function is used to issue payments from the MultiSig prover's account on the XRP Ledger during interchain transfers, this means that multiple parallel transfers will be assigned the same sequence number and only one of them will succeed or fail. This behavior will result in failed transfers during high transaction volume.

The MultiSig prover contract should instead either assign consecutive sequence numbers to transactions[3] or maintain a queue of interchain transfers and submit them sequentially as they get included in the XRP Ledger.

Recommendation

Consider testing multiple parallel interchain transfers on the testnet to ensure the desirable outcome.

Document the described behavior to avoid confusion as the project is maintained.

Location

https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/contracts/xrpl-multisig-prover/src/xrpl_multisig.rs#L272-L285

3. This solution is also recommended by the XRPL's guide to reliably submitting transactions: https://xrpl.org/docs/concepts/transactions/reliable-transaction-submission#determine-the-account-sequence.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Implementation Review Notes

This informational section highlights a number of observations that the NCC Group team gathered during the engagement and that do not warrant security-related findings on their own.

Unaddressed ToDos should be tracked

There are a number of untracked ToDos in the codebase. It is strongly recommended to create issues for these, and include a reference in the ToDo in order to ensure they are addressed in a timely manner. Some of the notable ToDos are listed below:

- Block height is used in place of the actual unique cross chain id. The correct transaction hash should be fetched from the Axelar nexus: https://github.com/commonprefix/axelar-amplifier/blob/392f49a88e63b829ae0d6af5955242e9c762d4bd/contracts/xrpl-gateway/src/contract/execute.rs#L460-L461
- Implementation should ensure that simple cross-chain payments do not include `data`. Cross currency payments are not handled at the moment: https://github.com/commonprefix/axelar-amplifier/blob/cfb43955b910b7da0074886e5968c0e229ea1c39/contracts/xrpl-multisig-prover/src/contract/execute.rs#L419-L427

Multiplication with -1 could be avoided

In the following code snippet from `XRPLTokenAmount`'s `from_str()` implementation, the `exponent * -1` multiplication could be avoided by swapping the `checked_sub()`'s operands on line 858:

```
858 let exponent = match decimal_offset.checked_sub(exponent_value) {
859     Some(exponent) => exponent,
860     None => {
861         return Err(XRPLError::InvalidAmount { reason: "overflow".to_string() });
862     }
863 };
864
865 if digits.starts_with('-') {
866     return Err(XRPLError::InvalidAmount { reason: "negative amount".to_string() });
867 }
868
869 if digits.starts_with('+') {
870     digits = digits[1..].to_string();
871 }
872
873 let mantissa = Uint256::from_str(digits.as_str())
874     .map_err(|e| XRPLError::InvalidAmount { reason: e.to_string() })?;
875
876 let (mantissa, exponent) = canonicalize_mantissa(mantissa, exponent * -1)?;
```

Figure 8: Excerpt from packages/xrpl-types/src/types.rs

Retest Results: The Common Prefix team implemented the recommended optimization as part of PR #1.

Unnecessary save to storage

In the following snippet from the MultiSig prover contract, if the `NEXT_VERIFIER_SET` exists and is identical to `new_verifier_set`, `different_set_in_progress()` function will return false and there will be an unnecessary save on line 153:

```
145 fn save_next_verifier_set(
146     storage: &mut dyn Storage,
147     new_verifier_set: &axelar_verifiers::VerifierSet,
148 ) -> Result<(), ContractError> {
    if different_set_in_progress(storage, new_verifier_set) {
        return Err(ContractError::VerifierSetConfirmationInProgress);
    }

    state::NEXT_VERIFIER_SET.save(storage, new_verifier_set)?;
    Ok(())
}

// Returns true if there is a different verifier set pending for confirmation, false if
↪ there is no
// verifier set pending or if the pending set is the same
fn different_set_in_progress(storage: &dyn Storage, new_verifier_set:
↪ &axelar_verifiers::VerifierSet) -> bool {
    if let Ok(Some(next_verifier_set)) = state::NEXT_VERIFIER_SET.may_load(storage) {
        return next_verifier_set != *new_verifier_set;
    }

    false
}
```

Figure 9: Excerpt from contracts/xrpl-multisig-prover/src/contract/execute.rs

Retest Results: The Common Prefix team removed the unnecessary save to storage in commit `5566061` as part of PR #1.

Stale Link in Serialize Code

The length encoding used during serialization is a port of reference code, as linked in the code comments:

```
// see https://github.com/XRPLF/xrpl-dev-portal/blob/master/content/_code-samples/tx-
↪ serialization/py/serialize.py#L92
// may error if length too big
fn encode_length(mut length: usize) -> Result<Vec<u8>, ContractError> {
```

Figure 10: Excerpt from contracts/xrpl-multisig-prover/src/xrpl_serialize.rs

However, the linked content appears to have shifted locations and is no longer present under the *content/* subdirectory. The updated link appears to be https://github.com/XRPLF/xrpl-dev-portal/blob/master/_code-samples/tx-serialization/py/serialize.py#L92.

Retest Results: The Common Prefix team updated the comment according to the recommendation in commit `41d970c` as part of PR #1.

Code Comment Should be Expanded

The following code comment in the `xrpl-voting-verifier` contract's `InstantiateMsg` structure definition needs to be expanded on. The XRPL gateway and Multisig Prover are distinct CosmWasm contracts; however, only the Multisig Prover has an account on the XRP ledger. This division should more clearly be documented:

```
    /// Axelar's gateway contract address on the source chain (i.e., the XRPL multisig
    ↪ address)
    #[serde(with = "xrpl_account_id_string")]
    #[schemars(with = "String")] // necessary attribute in conjunction with
    ↪ #[serde(with ...)]
    pub source_gateway_address: XRPLAccountId,
```

Figure 11: Excerpt from contracts/xrpl-voting-verifier/src/msg.rs

Retest Results: The Common Prefix team added a sentence to the comment to clarify that the XRPL MultiSig account is controlled by the MultiSig prover contract. See commit `04194ba` for details.

Weight Scaling Behavior should be Documented

The `convert_or_scale_weights()` helper function scales an array of weights such that the maximum weight maps to the `u16::MAX` value. It is unclear whether this conversion is necessary. It is recommended to include a link to the corresponding logic in Axelar, in the comments:

```rust
// Converts a Vec<Uint256> to Vec<u16>, scaling down with precision loss, if necessary.
// We make sure that XRPL multisig and Axelar multisig both use the same scaled down
↪ numbers and have the same precision loss
fn convert_or_scale_weights(weights: &[Uint128]) -> Result<Vec<u16>, ContractError> {
    let max_weight: Option<&Uint128> = weights.iter().max();
    match max_weight {
        Some(max_weight) => {
            let max_u16_as_uint128 = Uint128::from(u16::MAX);
            let mut result = Vec::with_capacity(weights.len());
            for &weight in weights.iter() {
                let scaled = weight.multiply_ratio(max_u16_as_uint128, *max_weight);
                result.push(convert_uint128_to_u16(scaled)?);
            }

            Ok(result)
        }
        None => Ok(vec![]),
    }
}
```

Figure 12: Excerpt from contracts/xrpl-multisig-prover/src/axelar_verifiers.rs

Retest Results: The Common Prefix team added a sentence to the comment to indicate that the XRPL MultiSig accounts require `u16` weights. See commit `b792c72` for details.

7 Contact Info

The team from NCC Group has the following primary members:

- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Kevin Henry – Consultant
  kevin.henry@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Axelar has the following primary member:

- Nikolaos Kamarinakis
  nikolas@commonprefix.com