Hypernova Eth Verifier Move
Smart Contract Assessment Report

Date: July 18, 2025
Version 1.0 [Final]

Contents

Executive Summary

Supra Labs team engaged with *RektProof Security* to perform an assessment of *Hypernova Eth Verifier Move*. This assessment was conducted between *Jun 24th* and *Jul 14th, 2025*.

Assessment overview

This report outlines the results of the security audit conducted on the Hypernova Eth Verifier Move.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

Contact Information

Name	Title	Contact Information
Supra Labs		
Arun Doraiswamy	VP of Engineering (Blockchain)	a.doraiswamy@supra.com
RektProof Security		
Naveen Kumar J	Smart Contract Auditor	naina.navs@gmail.com
Kalai Mohan	Smart Contract Auditor	kalaimohan328@gmail.com
Raj Kumar	Smart Contract Auditor	0xrajkumar1337@gmail.com

Finding Severity Classification

The following table defines a severity matrix that determines the severity of the bug by evaluating two main factors: the impact of the issue and its likelihood. This approach ensures an accurate assignment of severity.

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Scope

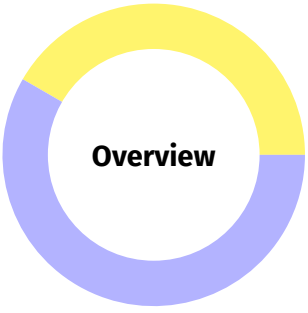
The source code was delivered to us in a git repository at `https://github.com/Entropy-Foundation/supra-interoperability-solutions/blob/hn-eth-verifier-move_v3`. This audit was performed against commit `b45f44346ba5feef90b3d219a3cdd80ce86f0b46`. Reported issues were fixed in the commit `4c8ce6849ec21e84d2efa55e7e3431337e29df21`.

A brief description of the programs is as follows:

Name	Description
Hypernova Eth Verifier Move	On-chain component responsible for verifying all the proofs coming from the Relayer.

Findings

Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	3
LOW	0
INFO	4



0.1 Moderate Risk

0.1.1 Inverted Logic in Bridge Pause State Validation Causing DoS

Severity: Moderate

Description:

The *assert_token_bridge_paused* function contains a logic error that prevents essential administrative operations from functioning as intended.

```
/// Asserts that the token bridge is currently paused.
inline fun assert_token_bridge_paused(paused: bool) {
    assert!(!paused, ETOKEN_BRIDGE_PAUSED); <--
}
```

Both *register_source_token_bridge* and *unregister_source_token_bridge* functions are designed to work when the bridge is paused. However, the internal validation function *assert_token_bridge_paused* uses inverted logic, which requires the bridge to be unpaused.

```
/// # Aborts - If the bridge is not paused
entry fun register_source_token_bridge(
    admin: &signer, source_token_bridge_addr: vector<u8>, source_chain_id: u64
) acquires TokenBridgeState {
    // Ensure the bridge is paused before making configuration changes
    assert_token_bridge_paused(bridge_state.is_paused);
    ...
}

entry fun unregister_source_token_bridge(admin: &signer) acquires TokenBridgeState {
    // Ensure the bridge is paused before unregistering
    assert_token_bridge_paused(bridge_state.is_paused);
    ...
}
```

Impact:

This bug creates a denial-of-service condition for critical administrative functions. When the bridge is paused for maintenance or emergency situations, administrators cannot register new source token bridges or unregister existing ones. This issue force administrators to unpause the bridge just to make configuration changes, thereby exposing the system to risks during what should be safe maintenance windows.

Recommendation:

Correct the logic in the *assert_token_bridge_paused* function. Consider adding another function *assert_token_bridge_not_paused* for purposes that only require the bridge not to be paused.

Status: *Resolved* - Removed all assertions of pause state from administrative functions, ensuring that they remain accessible regardless of the pause state.

0.1.2 Missing Pause State Validation in Token Creation Function

Severity: Moderate

Description:

The *create_wrapped_token* function does not validate the pause state of the wrapped token deployer before allowing token creation, while other administrative functions like *delist_wrapped_token* properly check the pause state using *ensure_wrapped_tokens_deployer_not_paused*(*state.is_paused*).

The documentation for *set_wrapped_token_deployer_pause_state* explicitly states that *"When paused, certain wrapped token deployer operations (like minting, registration) are restricted,"* indicating that token creation should be blocked when the system is paused.

However, the *create_wrapped_token* function only validates admin permissions and proceeds with token initialization without checking if the deployer is paused.

```
    /// # Aborts
    /// * If the caller is not the wrapped token deployer admin
    /// * If the wrapped token deployer is paused <---
    /// * If the source token address length is invalid

    entry fun create_wrapped_token(
        admin: &signer,
        decimals: u8,
        name: String,
        symbol: String,
        icon_uri: String,
        project_uri: String,
        token_origin_chain_id: u64,
        token_origin_addr: vector<u8>
    ) acquires WrappedTokensState {

        ensure_wrapped_token_deployer_admin(
            get_wrapped_tokens_state().admin_addr,
            signer::address_of(admin)
        );

        //Initialize the wrapped token
    }
```

Impact:

This oversight allows new wrapped tokens to be created even when the system is intentionally paused for maintenance, security incidents, or other operational reasons. This undermines the pause mechanism's effectiveness as a safety control and could lead to inconsistent system behavior where some operations are blocked while others continue.

Recommendation:

```
entry fun create_wrapped_token(
    ...
) acquires WrappedTokensState {
    ensure_wrapped_token_deployer_admin(
        get_wrapped_tokens_state().admin_addr,
        signer::address_of(admin)
    );

    ensure_wrapped_tokens_deployer_not_paused(
        get_wrapped_tokens_state().is_paused
    );

    //Initialize the wrapped token
}
```

Status: Resolved

Resolution: We have updated the function documentation to reflect the actual behavior — clarifying that the function can be invoked regardless of pause state.

0.1.3 Inconsistent Participation Threshold Validation Using Strict Inequality

Severity: Moderate

Description:

The *hypernova_core::process_light_client_optimistic_update* function uses strict inequality (*participation_count > get_sync_committee_threshold()*) to validate sync committee participation, which creates an off-by-one error in threshold enforcement.

If the *sync_committee_threshold* is configured to represent the minimum required participation count, then the current validation will incorrectly reject updates with exactly that threshold amount.

```
/// # Validation Steps:
/// 1. Sync Committee Participation
///    - Verifies sufficient committee member participation
...
public(friend) fun process_light_client_optimistic_update(
    update: &mut LightClientUpdate
) acquires LightClientState {

    // 1. Verify sync committee participation
    assert!(
        participation_count > get_sync_committee_threshold(), <---
        EINSUFFICIENT_PARTICIPATION
    );
```

Impact:

Updates with sufficient committee participation according to the configured threshold will be incorrectly rejected, potentially causing synchronization failures and preventing the light client from processing legitimate state updates.

Recommendation:

```
public(friend) fun process_light_client_optimistic_update(
    update: &mut LightClientUpdate
) acquires LightClientState {
    ...
    assert!(
        participation_count >= get_sync_committee_threshold(),
        EINSUFFICIENT_PARTICIPATION
    );
}
```

Status:

Resolved

0.2 Informational

0.2.1 Using Deprecated Aptos Standard Library Modules

Description:

Hypernova uses two deprecated Aptos modules: *aptos_std::smart_table* and *aptos_std::simple_map*. These modules have been officially deprecated due to poor performance and inefficient implementations. The Aptos team recommends replacing them with *big_ordered_map* and *ordered_map* respectively.

```
/// DEPRECATED: since it's implementation is inneficient, it
/// has been deprecated in favor of `big_ordered_map.move`.
module aptos_std::smart_table {..}

/// DEPRECATED: since it's implementation is inneficient, it
/// has been deprecated in favor of `ordered_map.move`.
module aptos_std::simple_map { ..}
```

Impact:

Using these deprecated modules consumes more gas and processes transactions slower than their replacements.

Recommendation:

Migrate *smart_table* to *big_ordered_map* and *simple_map* to *ordered_map*

Status: Acknowledged - While these modules are deprecated in the official Aptos standard library due to inefficiencies, the Supra environment does not currently provide support for the recommended replacements (*big_ordered_map* and *ordered_map*).

0.2.2 Missing Admin Change Functionality

Description:

The *Token Bridge Service* contract defines an *AdminChanged* event that tracks when the admin of the token bridge changes, including both the current and new admin addresses. However, there is no corresponding implementation to actually change or transfer admin privileges. The event structure exists, but no function calls or emits this event, creating a disconnect between the intended functionality and the actual implementation.

Impact:

This indicates incomplete functionality where admin management was planned but never implemented, leaving the system potentially without proper administrative controls.

Recommendation: Implement the missing admin change functionality or remove event.

Status: Resolved - The AdminChanged event was part of an earlier design consideration for single-admin control. However, in the current implementation, admin access is handled via a multisig account, and explicit admin transfer functionality is no longer required.

0.2.3 Redundant Duplicate Check in *update_fork_versions*

Description:

In the *hypernova_core::update_fork_versions* function, there is a redundant check for duplicate epochs. The function first checks if the new epoch already exists in the current forks list using vector::any, and then ensures that new_epoch > last_epoch. Since the fork list is maintained in strictly increasing order, the latter condition alone is sufficient to guarantee uniqueness. Thus, the duplicate check using vector::any is logically unnecessary.

```
let exists = vector::any(
    current_forks,
    |existing_fork| get_epoch(existing_fork) == new_epoch
);
```

Impact:

The redundant check introduces unnecessary gas costs by performing a linear search (O(n)) through the forks list, which scales with the number of forks.

Recommendation:

Remove the vector::any duplicate check. The new_epoch > last_epoch condition is sufficient to enforce both order and uniqueness, improving gas efficiency and simplifying the logic.

Status:

Acknowledged - Since this is an admin-only function, the frequency of execution is low.

0.2.4 Parameter Order Mismatch in Merkle Branch Validation

Description:

There is a parameter order mismatch in the *eth_types::is_valid_merkle_branch* function calls. The function signature expects ***(root, leaf, branch, depth, index)***.

```
public(friend) fun is_valid_merkle_branch(
    root: vector<u8>,
    leaf: vector<u8>,
    branch: vector<vector<u8>>,
    depth: u64,
    index: u64
): bool {
    ...
}
```

However, all calling functions are passing ***(root, leaf, branch, INDEX, DEPTH)***.

```
fun is_current_committee_proof_valid(
    ...
): bool {
    is_valid_merkle_branch(
        *get_state_root(attested_block_header),
        hash_tree_root_current_committee,
        current_committee_branch,
        CURRENT_COMMITTEE_INDEX,
        CURRENT_COMMITTEE_DEPTH
    )
}

fun is_next_committee_proof_valid(
    ...
): bool {
    is_valid_merkle_branch(
        *get_state_root(attested_block_header),
        leaf_hash_tree_root_next_committee,
        next_committee_branch,
        NEXT_COMMITTEE_INDEX, // Next committee index
        NEXT_COMMITTEE_DEPTH // Next committee depth
    )
}

fun is_finality_proof_valid(
    ...
): bool {
```

```
        is_valid_merkle_branch(
            *get_state_root(attested_block_header),
            hash_tree_root_beacon_block_header(finality_header),
            finality_merkle_proof,
            FINALITY_INDEX, // Finality index
            FINALITY_DEPTH // Finality depth
        )
    }
```

Impact:

Invalid Proof Verification: All Merkle proofs will be verified incorrectly, potentially accepting invalid proofs or rejecting valid ones.

Recommendation:

Pass the arguments in the correct order.

Status: Resolved - There is no functional bug in proof validation. The verification logic operates correctly, and Merkle branches are validated as expected. The issue is limited to naming conventions and does not affect correctness or security.