Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Hot Bridge

Veridise Inc.

September 1, 2025

► **Prepared For:**

HOT Labs
https://hot-labs.org/

► **Prepared By:**

Aayushman Magar
Evgeniy Shishkin
Kostas Ferles
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 1, 2025	V2
Aug. 8, 2025	V1
Aug. 5, 2025	Initial Draft

Contents

1 Executive Summary

From Jul. 16, 2025 to Jul. 30, 2025, HOT Labs engaged Veridise to conduct a security assessment of the Hot Bridge - a bridging protocol on the NEAR blockchain, along with a validation SDK and a Stellar-side component for interacting with the bridge. Veridise conducted the assessment over 8 person-weeks, with 4 security analysts reviewing the project over 2 weeks on commits `356f6eb`, `8729129` and `65520e7`. The review strategy involved a thorough review of the program source code, as well as penetration testing, performed by Veridise security analysts.

Project Summary. The Hot Bridge protocol enables token transfers across multiple blockchains. In contrast to similar solutions, the protocol focuses on decentralizing the off-chain validation layer. Specifically, it employs a network of validator nodes that act as a decentralized root of trust, responsible for attesting the correctness of transfers. These nodes utilize NEAR's Multi-Party Computation (MPC) technology, where no single node has sufficient authority to validate a transfer independently. A valid digital signature is generated only when a predefined threshold of nodes collectively approve a transaction.

Additionally, token swapping, which is a necessary step to convert one type of bridged token into another before a transfer can be finalized - is delegated to an external protocol called NEAR Intents. This separation of concerns keeps the bridge protocol modular and avoids embedding swap logic directly into the bridge mechanism.

The protocol architecture consists of the following components:

- **NEAR Bridge Contract (OmniBalance Contract)**. A smart contract deployed on the NEAR blockchain responsible for maintaining user balances of bridged tokens. It tracks all token amounts associated with a user, available for withdrawal or further transfer.
- **Lockers**. A set of smart contracts deployed on each supported blockchain. These contracts are responsible for locking tokens during outbound transfers and releasing them upon successful inbound transfers.
- **MPC Nodes**. A distributed set of validator nodes tasked with attesting deposit and withdrawal operations across chains. Upon reaching quorum, the nodes compute an aggregated digital signature, which the user presents on the destination chain to finalize the transfer. The validation logic executed by these nodes is implemented within the Hot Bridge SDK.

A typical user interaction flow involving the transfer of USDC tokens from Stellar to Ethereum using the HotDAO Bridge protocol consists of the following steps*:

1. **Deposit on Stellar Locker**. The user initiates the process by depositing a specified amount of USDC into the Stellar Locker Contract via the `deposit()` method. The user provides: Token identifier being deposited, Recipient address on NEAR bridge and Deposit amount. The USDC tokens are transferred to the Locker's custody, and the operation is associated with a *unique nonce* chosen by the user.

* Ethereum Locker contract was not a part of the scope, so this description is conceptual.

2. **Deposit attestation by MPC Nodes**. The user, through an off-chain service, submits a request to the MPC validator nodes to attest the deposit operation associated with the given nonce. Each MPC node independently verifies the existence and correctness of the deposit on Stellar. Once a quorum (threshold) of nodes validates the deposit, they collectively generate an aggregated digital signature (attestation), which is returned to the user.
3. **Finalizing deposit on NEAR Bridge**. The user calls the `deposit()` method on the NEAR Bridge Contract, supplying: the aggregated signature from the MPC nodes, deposit parameters (including the nonce and deposit details). Upon successful verification of the signature, the Bridge contract mints an equivalent amount of *Stellar.USDC* tokens to the user's balance on NEAR.
4. **Token swap via NEAR Intents**. To convert the minted *Stellar.USDC* tokens into *Ethereum.USDC* tokens, the user performs a swap operation via the NEAR Intents protocol. This swap applies only to NEAR Bridge token representations, not native USDC and involves a conversion fee deducted during the swap[†].
5. **Withdrawal request on NEAR Bridge**. The user requests a withdrawal of *Ethereum.USDC* tokens by invoking the `withdraw()` method on the NEAR Bridge Contract. The withdrawal request is enqueued and assigned a unique nonce identifier for tracking and attestation, the bridge tokens belonging to the user are burned.
6. **Withdrawal attestation by MPC Nodes**. Through the off-chain service, the user requests MPC nodes to attest the withdrawal request corresponding to the specified nonce. Each MPC node verifies the legitimacy of the withdrawal operation on NEAR. Once a quorum of nodes approves the operation, an aggregated digital signature is computed and returned to the user.
7. **Finalizing withdrawal on Ethereum Locker**. The user finalizes the withdrawal by calling the `withdraw()` method on the Ethereum Locker Contract, providing the aggregated signature and corresponding withdrawal parameters (including the nonce). After validating the signature and associated data, the Ethereum Locker releases the specified amount of *Ethereum.USDC* tokens to the user's balance on Ethereum.
8. **Delete used nonce value.** After the token transfer is finalized, the corresponding nonce must be deleted from the NEAR Bridge contract. Failing to remove the used nonce will cause subsequent withdrawal requests to be rejected by the MPC nodes for the same bridge user, in accordance with the protocol's validation rules.

Code Assessment. The Hot Bridge developers provided the source code of the Hot Bridge contracts for the code review. The source code appears to be mostly original code written by the Hot Bridge developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Hot Bridge developers also shared additional protocol documentation which gave a high-level overview of the Hot Bridge.

The source code contained a test suite, which the Veridise security analysts noted were lacking in integration and negative tests.

[†] This step was out of scope of this security review.

Summary of Issues Detected. The security assessment uncovered 21 issues, 6 of which are assessed to be of high or critical severity by the Veridise analysts. For example, analysts discovered the lack of access control in a callback function on the bridge, which enables arbitrary token transfers (V-HOTB-VUL-001); deposited tokens can be stolen due to hash collisions (V-HOTB-VUL-002); and the validation SDK is missing functionality to generate nonce cleaning signatures, which can prevent some nonces from being cleaned up (V-HOTB-VUL-001). The Veridise analysts additionally identified 4 medium-severity issues. For instance, analysts discovered that the lack of domain separation between deposit and withdrawal messages allows invalidating arbitrary nonces (V-HOTB-VUL-001); inaccurate criteria for signature generation can result in loss of funds due to improper consensus (V-HOTB-VUL-002). Additionally, the Veridise analysts also identified 4 low-severity, 6 warnings, and 1 informational findings.

The Hot Bridge developers have indicated an intent to fix these issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Hot Bridge.

Extend Test Suite. Both on-chain components - the Stellar Locker and NEAR Bridge - currently include a set of test cases. The Locker contract demonstrates limited test coverage, with only a small number of tests implemented, one of which contains incorrect logic. Although the NEAR Bridge contract has a comparatively larger test suite, the existing tests predominantly cover positive execution paths. Negative scenarios, such as access control validation and failure handling, remain insufficiently tested.

Expanding the test suites for both components is recommended. For the Locker contract, additional positive-path tests should be developed to ensure comprehensive validation of expected behaviors. The NEAR Bridge contract requires an extended set of negative-path tests.

Multiple issues identified during the security assessment could likely have been detected and mitigated through comprehensive negative testing, particularly those targeting access control logic.

Expand Threat Model. At the beginning of the security review a threat model document was provided, outlining a high-level overview of the system architecture along with identified risks and their corresponding mitigation strategies. This document was valuable for establishing an initial understanding of the protocol design. However, upon gaining a deeper comprehension of the system's workflows and mechanisms, it was observed that the threat model does not fully cover the primary operational flows and, as a result, omits several important risk areas.

One such example is nonce management, which is absent from the provided threat model. Notably, multiple high-severity issues identified during the assessment are directly related to this unaddressed aspect.

It is recommended that the threat model be extended to include a detailed analysis of all key workflows and their associated risks. A more comprehensive threat model will serve as a beneficial resource for both the development team and future security review processes.

Clarify Protocol Boundaries. The protocol documentation indicates support for multiple networks, specifically Solana, TON, NEAR, Stellar, and Ethereum. However, it was observed that support

for some of these networks is not fully implemented in either the NEAR Bridge source code or the Hot Bridge SDK.

It is recommended to explicitly define the current operational scope of the protocol, clearly specifying which networks are fully supported and which are planned for future integration.

Extend Protocol Monitoring Mechanisms. The NEAR Bridge contract currently emits events for critical operations such as token minting and burning. However, other significant protocol actions—including ownership transfers, MPC public key updates, and the addition of new supported chains—are not accompanied by corresponding event emissions. Additionally, the Locker smart contract does not emit any events, despite its critical role in asset custody and transfer workflows.

It is recommended to expand event coverage across both NEAR Bridge and Locker contracts to include all security-relevant state changes and protocol operations. Enhanced event logging will improve monitoring capabilities, facilitate off-chain auditability, and enable more effective detection of anomalous or unauthorized activities.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
NEAR Bridge	`356f6eb`	Rust	NEAR
Validation SDK	`8729129`	Rust	NEAR MPC plugin
Stellar Locker	`65520e7`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 16–Jul. 30, 2025	Manual & Tools	4	8 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	5	5	3
Medium-Severity Issues	4	4	3
Low-Severity Issues	4	4	4
Warning-Severity Issues	6	6	6
Informational-Severity Issues	1	1	1
TOTAL	21	21	18

Table 2.4: Category Breakdown.

Name	Number
NEAR Bridge	
Access Control	3
Data Validation	3
Logic Error	2
Cryptographic Vulnerability	1
Missing functionality	1
Usability Issue	1
Maintainability	1
Stellar Locker	
Data Validation	2
Missing functionality	1
Logic Error	1
Maintainability	1
SDK	
Cryptographic Vulnerability	1
Access Control	1
Maintainability	1
Logic Error	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Methodology

The Security Assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the codebase for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

During the security assessment, the Veridise security analysts regularly met with the Hot Bridge developers to ask questions about the code, discuss progress, and communicate identified issues. The Veridise security analysts also perused the shared documentation for the HOT OmniBalance documentation.

3.2 Identified Security Risks

After the initial phase of the security assessment was completed, a list of potential security risks was generated. Security analysts used this list during the code review as a starting point to identify potential attack vectors. A few of these risks, when expressed as questions, include the following:

- Are all high-privilege operations properly protected by access control mechanisms?
- Are there potential vectors for malicious actors to prevent legitimate users from executing deposit or withdrawal operations?
- Are there scenarios where bridge tokens could be minted without a corresponding deposit in the Locker contract?
- Are there mechanisms by which the protocol could be forced into a denial-of-service state, halting operations until an upgrade is performed?
- Are there any ways to withdraw funds from the Bridge without a corresponding deposit?
- Are griefing attacks feasible in the current protocol implementation?
- Are there methods by which MPC validators could be manipulated into signing invalid operations?
- Is the NEP-245 standard implemented with full adherence to its security considerations?
- Are there conditions under which NEAR Bridge tokens could be double-spent?
- Are there front-running risks associated with NEAR Bridge operations?
- Are there any Stellar-specific or NEAR-specific vulnerabilities present in the code?

3.3 Scope

The scope of this security assessment is limited to a specific set of source files in each repository, as agreed upon with the Hot Bridge developers:

NEAR bridge on commit `356f6eb`:

- `/contract/src/deposit.rs`
- `/contract/src/hot_protocol.rs`
- `/contract/src/lib.rs`
- `/contract/src/owner_methods.rs`
- `/contract/src/state_migration.rs`
- `/contract/src/storage_keys.rs`
- `/contract/src/views.rs`
- `/contract/src/standards/mod.rs`
- `/contract/src/standards/nep245/core.rs`
- `/contract/src/standards/nep245/metadata.rs`
- `/contract/src/standards/nep245/mod.rs`
- `/contract/src/standards/nep245/resolver.rs`
- `/contract/src/standards/storage_management/ext.rs`
- `/contract/src/standards/storage_management/mod.rs`
- `/contract/src/types/chain_nonce_tracker.rs`
- `/contract/src/types/mod.rs`
- `/contract/src/types/token_balance.rs`
- `/contract/src/types/withdraw_engine.rs`
- `/contract/src/types/proof_verifier/deposit.rs`
- `/contract/src/types/proof_verifier/mod.rs`
- `/contract/src/types/proof_verifier/withdraw_removal.rs`
- `/contract/src/withdrawals/clear.rs`
- `/contract/src/withdrawals/mod.rs`
- `/contract/src/withdrawals/withdraw.rs`

Validation SDK on commit `8729129`:

- `/src/evm.rs`
- `/src/internals.rs`
- `/src/lib.rs`
- `/src/near.rs`
- `/src/stellar.rs`

Stellar locker on commit `65520e7`:

- `src/lib.rs`

3.4 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Security Review Assumptions

4.1 Operational Assumptions

In addition to assuming that any external libraries behaved correctly and are free of security vulnerabilities, the Veridise analysts made the following assumptions for their security assessment:

- **MPC Network**: The NEAR MPC network is used by the Hot Bridge in order to generate signatures that authenticate token transfers. The Veridise analysts assume this network to be robust. That is, only valid withdrawals/deposits will be signed by the network's private key. From the point of view of the MPC network, the validity of a withdrawal or deposit is left entirely to the validation SDK.
- **Trusted configuration**: Both the NEAR bridge and Stellar locker can be configured with an owner address on construction. It is assumed that these contracts will be configured in such a way that the owner will be a trusted party, such as a DAO.
- **Deposits from other chains**: The Hot Bridge protocol also supports transfers from other sources such as different EVM chains. For these transfers, it is assumed that the `hot_verify` function used by the validation SDK will only return `true` if the corresponding tokens were in fact deposited to the corresponding EVM locker. It is also assumed that the ABI used by the validation SDK matches the one from the invoked EVM contract.
- **Bridge storage costs**: NEAR contracts stake native NEAR tokens in order to store contract data. In order to create new entries in a map or similar data structures, the NEAR bridge will need to have enough balance to cover these costs. It is assumed by the Veridise analysts that HOT Labs or some other party will maintain the bridge's balance in order for it to be able to continue to operate.
- **Swapping correctness**: If a user decides to bridge tokens from a source chain A to a different destination chain B, through the Hot Bridge they will need to swap their wrapped tokens for the appropriate type (from A-wrapped tokens to B-wrapped tokens). It is assumed by the Veridise analysts that swaps will only be performed when the locker on the destination chain (B) is guaranteed to have enough liquidity to cover a potential withdrawal.

4.2 Privileged Users

During the review, Veridise analysts assumed that the privileged users perform their responsibilities as intended. Protocol exploits relying on the below users acting outside of their privileged scope or intentionally abstaining from doing their duties are considered to be outside of scope of this security review.

- **Stellar Locker Owner**. This role has the authority to assign the public key associated with the MPC service, which is required for validating and signing withdrawal requests.

Additionally, the Stellar Locker Owner possesses the capability to withdraw tokens directly from the Locker contract without any protocol-enforced restrictions or checks.

- **NEAR Bridge Owner**. This role has the authority to add, remove, or modify the list of supported blockchains. Furthermore, the NEAR Bridge Owner can assign or update the public key used by the MPC service to validate and sign deposit requests.
- **MPC Service Configuration Administrators**. The MPC service retrieves its configuration data from an external smart contract (not included within the scope of this assessment). This configuration includes sensitive data, such as the addresses of smart contracts that are queried to validate protocol operations. The administrators managing this configuration hold significant influence over the protocol's security posture.

Operational Recommendations Highly privileged operations should be executed by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly privileged operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified.

Table 5.1: Summary of Discovered Bridge Vulnerabilities.

ID	Description	Severity	Status
V-HOTB-VUL-001	Lack of access control in resolve function . . .	Critical	Fixed
V-HOTB-VUL-002	Bridge tokens can be minted to attacker . . .	High	Fixed
V-HOTB-VUL-003	Double spend attack due to missing nonce . . .	High	Fixed
V-HOTB-VUL-004	Griefing attack on user withdrawals due . . .	High	Fixed
V-HOTB-VUL-005	Withdrawal receiver is unable to clean . . .	High	Confirming Fix
V-HOTB-VUL-006	Million Small Deposits attack through . . .	Medium	Fixed
V-HOTB-VUL-007	Empty withdrawal requests can be used to . . .	Medium	Fixed
V-HOTB-VUL-008	Bridge account vulnerable to unauthorized . . .	Low	Fixed
V-HOTB-VUL-009	Non-compliance with NEP-145	Low	Fixed
V-HOTB-VUL-010	Unbounded token batch transfer may lead . . .	Low	Fixed
V-HOTB-VUL-011	Bridge init call front-running risk	Warning	Fixed
V-HOTB-VUL-012	Minor Code Quality Issues	Info	Fixed

Table 5.2: Summary of Discovered Validation Vulnerabilities.

ID	Description	Severity	Status
V-HOTB-VUL-001	SDK is missing nonce cleaning signature . . .	High	Confirming Fix
V-HOTB-VUL-002	Inaccurate criteria for signature generation . . .	Medium	Fixed
V-HOTB-VUL-003	Servers collection allows for duplicates	Low	Fixed
V-HOTB-VUL-004	Maintainability issues	Warning	Fixed
V-HOTB-VUL-005	EVM verifier config is taken from Stellar . . .	Warning	Fixed

Table 5.3: Summary of Discovered Locker Vulnerabilities.

ID	Description	Severity	Status
V-HOTB-VUL-001	Lack of domain separation between . . .	Medium	Acknowledged
V-HOTB-VUL-002	Change of owner does not use two step . . .	Warning	Fixed
V-HOTB-VUL-003	Maintainability issues	Warning	Fixed
V-HOTB-VUL-004	Incorrect order of arguments when . . .	Warning	Fixed

5.1 Detailed Description of Issues in Bridge

5.1.1 V-HOTB-VUL-001: Lack of access control in resolve function enables arbitrary token transfers

Severity	Critical	**Commit**	356f6eb
Type	Access Control	**Status**	Fixed
File(s)	`contract/src/standards/nep245/resolver.rs`		
Location(s)	contract/src/standards/nep245/resolver.rs:13		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/14`, 06c055e		

Description NEP-245 defines a cross-contract token transfer interface that includes `mt_transfer_call()`, a method which enables users to attach tokens to an external function call.

```
// The 'mt_transfer_call' process:
//
// 1. Sender calls 'mt_transfer_call' on MT contract
// 2. MT contract transfers token from sender to receiver
// 3. MT contract calls 'mt_on_transfer' on receiver contract
// 4+. [receiver contract may make other cross-contract calls]
// N. MT contract resolves promise chain with 'mt_resolve_transfer', and may
//    transfer token back to sender
```

Snippet 5.1: Snippet from the NEP-245 specification

Critically, NEP-245 mandates that `mt_resolve_transfer()` *must* only be callable by the MT contract itself. This restriction ensures that only legitimate transfer chains can trigger the resolution logic. This resolution logic involves refunding any unused tokens back to the original sender, or refunding all the tokens transferred in (2) if the call to `mt_on_transfer()` was unsuccessful.

However, in this implementation, `mt_resolve_transfer()` is not marked as private. While direct calls by users may fail due to runtime checks like reading nonexistent promise results, those checks are insufficient to prevent abuse by other contracts.

```
#[payable]
fn mt_resolve_transfer(
    &mut self,
    previous_owner_ids: Vec<AccountId>,
    receiver_id: AccountId,
    token_ids: Vec<defuse_nep245::TokenId>,
    #[allow(unused_mut)] mut amounts: Vec<U128>,
    approvals: Option<Vec<Option<Vec<ClearedApproval>>>>,
) -> Vec<U128> {
  // ...
}
```

Snippet 5.2: | macro.]Snippet from `resolver.rs`. Note the absence of the `#[private]` macro.

An attacker can exploit this by registering `mt_resolve_transfer()` as a callback on a fake promise from their own contract:

1. The attacker's contract makes an async call to any contract, such as itself, setting up a fake promise chain.
2. It registers the vulnerable contract's `mt_resolve_transfer()` as the callback target.
3. Upon resolution, the NEAR runtime executes `mt_resolve_transfer()`, with parameters crafted by the attacker.
4. Since access is unrestricted, the OmniBridge contract processes the call, "refunding" tokens back to the attacker as if a failed transfer had occurred.

Impact An attacker can use `mt_resolve_transfer()` to arbitrarly transfer tokens from any user's account to themselves.

Recommendation Mark `mt_resolve_transfer()` as `#[private]`.

Developers Response The developers have fixed the issue by marking `mt_resolve_transfer()` as `#[private]` instead of `#[payable]`.

5.1.2 V-HOTB-VUL-002: Bridge tokens can be minted to attacker due to mt_deposit_call() hash collision

Severity	High	**Commit**	356f6eb
Type	Cryptographic Vulnerabilit	**Status**	Fixed
File(s)	`contract/src/deposit.rs`		
Location(s)	contract/src/deposit.rs:87-93		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/15`		

Description The bridge offers an `mt_deposit_call()` function, which allows a user to make an external contract call with newly deposited tokens using the NEP-245 `mt_transfer_call()` process.

```
#[payable]
pub fn mt_deposit_call(
    &mut self,
    deposit_call_args: DepositCallArgs,
    chain_id: u64,
    contract_id: String,
    amount: U128,
    nonce: U128,
    signature: SignatureBs58,
) -> PromiseOrValue<Vec<U128>> {
    // ...
    {
        let receiver_id = env::sha256_array(
            &[
                deposit_call_args.account_id.as_bytes(),
                deposit_call_args.msg.as_bytes(),
            ]
            .concat(),
        );

        self.deposit_verifier
            .verify_against_mpc_public_key(&receiver_id, &token_id, amount, nonce, &signature)
            .unwrap_or_panic_display();
    }
    // ...
    self.token_balance
        .deposit(&PREDECESSOR_ACCOUNT_ID.clone(), &token_id, amount)
        .unwrap_or_panic_display();
    // ...
    self.mt_transfer_call(
        deposit_call_args.account_id,
        token_id.to_string(),
        U128(amount),
        None,
        None,
        deposit_call_args.msg,
    )
```

```
}
```

Snippet 5.3: Snippet from `mt_deposit_call()`

To validate the deposit, the bridge checks that the provided `signature` attests to the nonce, token ID, amount, and a computed `receiver_id`. This `receiver_id` is derived by concatenating two variable-length, user-controlled fields-`deposit_call_args.account_id` and `deposit_call_args.msg`-and hashing the result:

This construction is susceptible to hash collisions. For example, the following inputs produce the same `receiver_id`:

```
// User's valid inputs
{ "account_id": "alice.near", "msg": "message" }
// Attacker input that collides
{ "account_id": "alice",      "msg": ".nearmessage" }
```

To reuse the existing `mt_transfer_call()` function, the bridge first transfers the newly minted tokens to the sender of `mt_deposit_call()`. The `mt_transfer_call()` is then invoked to forward the tokens to the `deposit_call_args.account_id` account along with a call to that account's `mt_on_transfer()` function.

An attacker can exploit this by crafting a `receiver_id` collision and ensuring that the `mt_on_transfer()` call to the forged `deposit_call_args.account_id` fails. When it does, the `mt_resolve_transfer()` callback will revert the transfer and refund the tokens as per the NEP-245 specification. Because the funds were initially sent to the sender of `mt_deposit_call()` (i.e. the attacker), this refund will allow the attacker to steal the deposited funds.

The attacker can also use this attack vector by using a signature that is meant for the `deposit()` function, and instead passing it to `mt_deposit_call()` with an empty `deposit_call_args.msg` field.

```
#[payable]
pub fn deposit(
    &mut self,
    receiver_id: AccountId,
    chain_id: u64,
    contract_id: String,
    amount: U128,
    nonce: U128,
    signature: SignatureBs58,
) {
  // ...
    {
        let receiver_id = env::sha256_array(receiver_id.as_bytes());

        self.deposit_verifier
            .verify_against_mpc_public_key(&receiver_id, &token_id, amount, nonce, &
    signature)
            .unwrap_or_panic_display();
    }
  // ...
}
```

Snippet 5.4: Snippet from `deposit()`

Impact If an attacker is able to obtain a valid signature and submit their call to `mt_deposit_call()` before the user, they can steal their deposited funds.

Recommendation Hash the `deposit_call_args.account_id` and `deposit_call_args.msg` fields separately, preventing attackers from crafting colliding values.

Developers Response The developers have provided the following fixes:

- The `receiver_id` field is now computed using bytes of the JSON representation of the `deposit_call_args`. This separates the `account_id` and `msg` fields, and prevents collisions.
- The funds in `mt_deposit_call()` are now first deposited to the bridge's owner address instead of the caller of `mt_deposit_call()` as the intermediary.

5.1.3 V-HOTB-VUL-003: Double spend attack due to missing nonce freshness check

Severity	High	**Commit**	356f6eb
Type	Data Validation	**Status**	Fixed
File(s)	`contract/src/deposit.rs`		
Location(s)	contract/src/deposit.rs:46-48		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/21`		

Description The **HotDao Bridge** contract defines an administrative function, `owner_clear_used_nonces()`, intended to remove previously used deposit nonces from on-chain storage. This mechanism serves to mitigate unbounded growth of the nonce cache and preserve storage efficiency over the long term. The function is restricted to clearing nonces deemed expired, as determined by comparing their timestamps to the `MAX_CACHE_DELAY_SECONDS` constant, which is currently configured to 18 days.

It is important to note, however, that the `deposit()` function - responsible for executing pending deposits - does not enforce any validation of nonce freshness.

Impact Since the `deposit()` function does not verify the freshness of the nonce, it introduces a potential attack vector: a malicious actor could wait until their previously used nonce has been cleared from storage, and then re-submit a deposit using the same nonce.

Recommendation It is required to ensure nonce freshness in the `deposit()` function and decline expired nonce values.

Developers Response The developers have updated the internal data structure that tracks deposit nonces so that:

- A nonce cannot be added if it is too old.
- A nonce cannot be removed if it isn't old enough.

5.1.4 V-HOTB-VUL-004: Griefing attack on user withdrawals due to missing access control checks

Severity	High	**Commit**	356f6eb
Type	Access Control	**Status**	Fixed
File(s)	`contract/src/owner_methods.rs`		
Location(s)	contract/src/owner_methods.rs:129-143		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/18`		

Description The **HotDAO Bridge** enforces **sequential processing of withdrawal requests**. Specifically, if a withdrawal request `W1` is registered prior to a request `W2`, the protocol mandates that `W1` must be processed before `W2` can be executed. To enforce this ordering, the protocol requires that the most recently used nonce - i.e., the smallest among all assigned nonces - must be removed before any subsequent withdrawal request can be processed, whether by users directly or by an external entity.

To support external execution of nonce removal, the protocol exposes two functions: `set_public_key_for_nonce_removal()` and `remove_public_key_for_nonce_removal()`. These functions assign or revoke the public key of an external party authorized to perform withdrawal nonce removal operations. Although these are clearly administrative functions, the current implementation lacks appropriate access control, thereby introducing a potential security risk.

Impact A malicious actor could perform a **griefing attack** that, while not yielding direct profit, can inflict significant damage on legitimate users.

Consider the following scenario:

1. A user initiates a withdrawal request targeting chain `C`, which becomes associated with a withdrawal nonce `n1`.
2. The attacker calls `remove_public_key_for_nonce_removal()` to remove the currently assigned public key responsible for nonce removal on chain `C`.
3. The attacker then calls `set_public_key_for_nonce_removal()` to register their own public key for chain `C`.
4. Before the user contacts the MPC service to obtain a withdrawal signature, the attacker invokes `clear_completed_withdrawal(n1)`, providing a valid signature corresponding to their registered key.

As a result, nonce `n1` is prematurely deleted from the `withdraw_engine` collection. This deletion prevents the MPC nodes from verifying and signing the user's withdrawal request. Critically, the user's bridge tokens will already have been burned at this stage, causing an **irrecoverable loss of funds**.

Recommendation Both functions have to be protected to be callable only by the protocol owner.

Developers Response The developers have added checks to both functions so that they are only callable by the bridge's owner account ID.

5.1.5 V-HOTB-VUL-005: Withdrawal receiver is unable to clean nonces for themselves

Severity	High	**Commit**	356f6eb
Type	Missing functionality	**Status**	Confirming Fix
File(s)	`contract/src/types/proof_verifier/withdraw_removal.rs`		
Location(s)	contract/src/types/proof_verifier/withdraw_removal.rs:83-85		
Confirmed Fix At	N/A		

Description After initiating a withdrawal on the NEAR side, the receiver can claim their funds by first requesting an attestation from the MPC service for the pending withdrawal. They then submit the resulting signature to the Locker contract on the destination chain to release the funds.

To process a subsequent withdrawal, the receiver must clear the used withdrawal nonce on the NEAR side - a requirement dictated by the current bridging protocol design. This can be done in several different ways:

- `clear_completed_withdrawal()`, which allows clearing the nonce with a signature from the MPC service
- `clear_withdrawal_by_receiver()`, which clears the nonce with a signature from the receiver of the withdrawal
- `clear_withdrawal_by_sender()`, which clears the nonce if called by the sender of the withdrawal

However, `clear_withdrawal_by_receiver()` is currently not implemented for the Stellar and TON blockchains and `clear_completed_withdrawal()` is currently not implemented on the MPC service.

As a result, the only available workaround for Stellar- or TON-bound withdrawals is for the sender of the original withdrawal to perform the cleanup by calling `clear_withdrawal_by_sender()`. This is acceptable if the sender and receiver are the same user, but if they are different parties, coordination is required - which may not always be feasible.

Impact In the worst case, the receiver could be blocked from processing pending withdrawal requests addressed to them.

Recommendation Support for Stellar network signatures must be implemented within the `verify_against_user_public_key()` function.

Developers Response Developers confirmed the issue and agreed to provide a fix.

5.1.6 V-HOTB-VUL-006: Million Small Deposits attack through storage_deposit() function

Severity	Medium	**Commit**	356f6eb
Type	Logic Error	**Status**	Fixed
File(s)	`contract/src/standards/storage_management/ext.rs`		
Location(s)	contract/src/standards/storage_management/ext.rs:12		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/22`		

Description The NEAR bridge implements the NEP-145 `StorageManagement` interface, which offers a way to pass NEAR storage costs to users. In the case of the bridge contract, this interface is being used to charge protocol fees when creating withdrawal requests. The implementation works by creating account balances for users of the protocol. They can then affect their balance by depositing NEAR through `storage_deposit()` and withdrawing them back through `storage_withdraw()`. Users can also deposit NEAR to other users' accounts by passing in a different account ID as argument.

However, creating a new balance for a user requires the contract to stake NEAR tokens as well, and this cost is not charged to the user. This means that users can create very small balances (i.e. 1 yoctoNEAR) and have the contract cover the difference in cost. This attack vector is also known as the Million Small Deposits in the NEAR documentation.

Impact Repeated calls to `storage_deposit()` with new account IDs can allow an attacker to drain the balance of the bridge contract. The attacker will effectively only be charged gas costs to perform this which, based on testing, are less than the amount staked by the contract to create a new balance.

If the bridge's balance becomes too low, a lot of its functionality will also stop working. This includes any operation that requires the creation of new state, such as processing a deposit to a new user, transferring tokens to a new user or creating withdrawals.

Recommendation Calculate the storage cost of creating a `StorageManagement` account and charge this cost to the user when they create a new account, as is recommended by NEP-145.

Developer Response The developers have fixed the issue as follows:

- The `storage_deposit()` function now requires a minimum deposit that covers the cost of 200 bytes of storage, which is the size of a single entry in the stored balances.
- The `storage_deposit()` function now makes use of the `registration_only` flag, in the way that is outlined by the NEP-145 spec.

5.1.7 V-HOTB-VUL-007: Empty withdrawal requests can be used to block specific users

Severity	Medium	**Commit**	356f6eb
Type	Data Validation	**Status**	Fixed
File(s)	`contract/src/withdrawals/withdraw.rs`		
Location(s)	contract/src/withdrawals/withdraw.rs:30		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/24`		

Description The `withdraw()` function, which creates and registers withdrawal requests, does not validate that the withdrawal amount is non-zero.

Impact This creates a potential attack vector where a malicious actor can generate hundreds of zero-value withdrawal requests targeting a specific legitimate user. The protocol only allows users to obtain a signature for a single pending withdrawal at a time. Therefore, in order to process their own legitimate withdrawal, the targeted user would first need to clear all these pending empty requests by calling either `clear_completed_withdrawal()` or `clear_withdrawal_by_receiver()`. Importantly, these functions do not support batch operations, meaning each nonce must be cleared individually.

This imposes a significant operational burden on the user, both in time and effort, effectively preventing them from completing their withdrawal in a timely manner. Meanwhile, the attacker incurs minimal cost: they can register a large number of withdrawal requests within a single transaction, and each call costs approximately 0.004 NEAR in protocol fees-roughly one cent at current prices.

Recommendation Introducing a minimum withdrawal amount is recommended to prevent the described griefing attack scenario.

Developers A set of batch nonce-cleaning functions was introduced, enabling more efficient nonce management. Furthermore, an additional mechanism now adjusts the withdrawal fee based on the pending queue length.

5.1.8 V-HOTB-VUL-008: Bridge account vulnerable to unauthorized fund withdrawals via NEP-245 transfer call

Severity	Low	**Commit**	356f6eb
Type	Logic Error	**Status**	Fixed
File(s)	`contract/src/standards/nep245/mod.rs`		
Location(s)	contract/src/standards/nep245/mod.rs:86-92		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/25`		

Description As part of NEP-245, users can perform a *transfer call* with a single or a batch of tokens. This process allows them to:

1. Transfer tokens to a specified `receiver_id`.
2. Call an `mt_on_transfer()` function on the `receiver_id`.
3. Finalize the transaction in the `mt_resolve_transfer()` callback function. This function should refund any unused tokens, or the entire amount transferred in (1) in case (2) fails.

In the NEAR bridge contract, (1) is performed by calling `internal_mt_batch_transfer()`. In most cases, this function updates the internal state of the sender and receiver accounts in order to perform the transfer. However, in the case that `receiver_id` is the bridge account itself, `internal_mt_batch_transfer()` will instead create a new withdrawal for the transferred tokens.

```
fn internal_mt_batch_transfer(
    &mut self,
    sender_id: AccountId,
    receiver_id: AccountId,
    token_ids: Vec<defuse_nep245::TokenId>,
    amounts: Vec<U128>,
    memo: Option<&str>,
) -> Result<()> {
    // ...
    if receiver_id == CURRENT_ACCOUNT_ID.clone() {
        // This logic is needed in places, where only NEP245 API is available,
        //  but we still want to make OmniBridge withdrawals, i.e. call 'withdraw
    (...)' method.
        for (token_id, amount) in token_and_amount_to_send {
            let receiver_id = StringBase58::new(memo.unwrap().to_string());
            self.withdraw(token_id.to_string(), receiver_id, U128(amount));
        }
    } else {
        for (token_id, amount) in token_and_amount_to_send {
            self.token_balance
                .transfer(&sender_id, &receiver_id, &token_id, amount)?
        }
    }
    // ...
}
```

Snippet 5.5: Snippet from `internal_mt_batch_transfer()`

Additionally, the bridge does not implement an `mt_on_transfer()` function, so any attempted call to it will fail. Therefore, if a user calls `mt_transfer_call()` or `mt_batch_transfer_call()` with the bridge as the `receiver_id`, they will initiate a withdrawal and the bridge will attempt to refund them after the call to `mt_on_transfer()` fails.

Impact If the bridge account has a balance for a given token, an attacker can use `mt_transfer_call()` in order to create a withdrawal for the same token type and be refunded by the bridge's account, effectively stealing funds.

Recommendation Either prevent users from calling `mt_transfer_call()` and `mt_batch_transfer_call()` with the bridge as the `receiver_id` or implement an `mt_on_transfer()` function that ensures that no refunds will be made.

Developers Response The developers have added a check to `internal_mt_batch_transfer()` that prevents the `receiver_id` from being the bridge's ID.

5.1.9 V-HOTB-VUL-009: Non-compliance with NEP-145

Severity	Low	**Commit**	356f6eb
Type	Usability Issue	**Status**	Fixed
File(s)	`contract/src/standards/storage_management/ext.rs`		
Location(s)	contract/src/standards/storage_management/ext.rs		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/22`		

Description The NEAR bridge contract implements the NEP-145 `StorageManagement` interface in `contract/src/standards/storage_management/ext.rs`. However, the implementation here differs in a few areas from the NEP-145 specification. In particular:

1. The `storage_deposit()` function takes a `registration_only` flag as an argument. According to the specification, when set the caller should be refunded any amount exceeding the minimum balance or the full deposit if they already had an account registered. However, this flag is currently unused by the bridge contract.
2. In the specification, the `storage_unregister()` function allows the caller to unregister themselves and withdraw their stored NEAR. This is intended to allow users to reclaim the tokens that they staked to store their balance. However, in the bridge contract this function is currently unimplemented.
3. As mentioned in Million Small Deposits attack through storage_deposit() function, the bridge contract's implementation does not charge users for the storage cost of storing their account balance. This can lead to discrepancies in the actual amount that they are able to withdraw (as it may be less than the amount returned by `storage_balance_of()`)

Impact Differences between the standard specification of NEP-145 and its implementation can lead to unexpected behavior for users of the bridge.

Recommendation If the aim is to support NEP-145, then the reported issues should be fixed to match the specification of it.

Developer Response The developers have yet to acknowledge the issue.

5.1.10 V-HOTB-VUL-010: Unbounded token batch transfer may lead to token loss in some scenarios

Severity	Low	**Commit**	356f6eb
Type	Data Validation	**Status**	Fixed
File(s)	`contract/src/standards/nep245/mod.rs`		
Location(s)	contract/src/standards/nep245/mod.rs:106-107		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/27`		

Description The NEP-245 function `mt_batch_transfer_call()` enables batch token transfers while notifying the receiver via a callback to `mt_on_transfer()`. If this callback fails or indicates that some tokens are not accepted, the contract is required to invoke `mt_resolve_transfer()` to perform a full or partial refund to the sender.

The current implementation does not impose a limit on the number of token transfers a user can include in a single batch request. Additionally, token identifiers are provided as Base58-encoded strings, which are decoded into TokenID structures during execution. This decoding process can be computationally intensive when handling large batches.

Impact In case of a large number of token transfers while attaching an insufficient amount of gas, if the receiver rejects some of the tokens or if `mt_on_transfer()` fails, the token must invoke `mt_resolve_transfer()` to refund the sender. However, due to the unbounded batch size and the gas-intensive Base58 decoding, `mt_resolve_transfer()` could exhaust the prepaid gas and abort, resulting in token loss for the sender.

Our experiments indicate that a batch size of approximately 130 tokens can approach the 7 TeraGas (7TGas) gas limit allocated to the resolver. While typical usage scenarios are unlikely to involve such large batch transfers, if they do occur - for example, if a token identifier is repeated multiple times - the resolver may run out of gas, resulting in partial or complete token loss for the sender. Note that the resolver function can receive more than 7 TGas for execution, depending on the gas originally attached to the batch transfer call. However, if the attached gas is in that range, the described scenario will still apply.

Recommendation It is recommended to enforce a reasonable upper limit on the number of tokens that can be processed in a single batch transfer to ensure reliable execution.

Developer Response The developers have added a limit of 100 tokens to the `mt_batch_transfer_call()` method.

5.1.11 V-HOTB-VUL-011: Bridge init call front-running risk

Severity	Warning	**Commit**	356f6eb
Type	Access Control	**Status**	Fixed
File(s)	`contract/src/lib.rs`		
Location(s)	contract/src/lib.rs:53		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/23`		

Description The constructor of the NEAR bridge (`Contract::new()` in `lib.rs`) is not marked to be private (`#[private]`). This means that if the contract were to be deployed without using this function as an init function, then any arbitrary user could initialize the contract themselves.

Impact If an attacker is able to front-run the initialization of the bridge, they could set themselves as the owner of the contract and gain access to priviledged methods such as `change_owner()`, `owner_clear_withdraws()`, `owner_clear_used_nonces()`, `add_chain()`, `rotate_public_key()`, and `set_locker_address()`.

Recommendation Mark any initialization function to be private.

Developer Response The developers have marked the `new()` function to be `#[private]`.

5.1.12 V-HOTB-VUL-012: Minor Code Quality Issues

Severity	Info	**Commit**	356f6eb
Type	Maintainability	**Status**	Fixed
File(s)	`contract/src/deposit.rs`, `contract/src/hot_protocol.rs`, `contract/src/types/proof_verifier/deposit.rs`, `contract/src/storage_keys.rs`, `contract/src/owner_methods.rs`, `contract/src/lib.rs`, `contract/src/types/withdraw_engine.rs`, `contract/src/withdrawals/withdraw.rs`, `contract/src/standards/nep245/mod.rs`		
Location(s)	contract/src/deposit.rs:20, contract/src/hot_protocol.rs:10, contract/src/types/proof_verifier/deposit.rs:12, contract/src/storage_keys.rs, contract/src/owner_methods.rs:129, contract/src/lib.rs:48, contract/src/types/withdraw_engine.rs:163, contract/src/withdrawals/withdraw.rs:25, contract/src/standards/nep245/mod.rs:50		
Confirmed Fix At	`https://github.com/hot-dao/hot-bridge/pull/29`		

Description During the manual code review of the NEAR bridge, several maintainability issues were discovered.

1. In `contract/src/deposit.rs:20` the docstring incorrectly refers to withdrawals instead of deposits.
2. In `contract/src/hot_protocol.rs:10`, the docstring refers to a `wallet_id` arg that is not present in the function's signature.
3. In `contract/src/types/proof_verifier/deposit.rs:12`, the docstring incorrectly refers to withdrawals instead of deposits.
4. In `contract/src/storage_keys.rs`, the `WithdrawalsByChainAndReceiver`, `_DeprecatedMultiTokenMetadataStorageMetadataByTokenId` and `_DeprecatedMultiTokenMetadataStorageTokenIdByMetadataId` enum variants are never used in the codebase.
5. In `contract/src/owner_methods.rs:129`, the name of the function is a bit unclear, as it implies it can be used to change a given public key. In reality this function will panic if a value is already set for a key, so `add_public_key_for_nonce_removal()` would be a better name.
6. In `contract/src/lib.rs:48`, the `blacklist` field is never used by the rest of the codebase.
7. In `contract/src/types/withdraw_engine.rs:163`, the variable name is a bit confusing since the values being stored are nonces and not actual withdrawal records.
8. In `contract/src/withdrawals/withdraw.rs`, the `withdraw()` manually checks that 1 yoctoNEAR is attached instead of using `assert_one_yocto()` like the rest of the codebase does.
9. In `contract/src/standards/nep245/mod.rs:50`, the function unwraps the `memo` argument without checking if it has a value. This will cause the program to panic instead of returning a helpful error in case the user forgets to set this field.

Impact Although the above issues are maintainability issues with no security impact, leaving them unfixed will increase the likelihood of future bugs and increase the difficulty of extending

the code base.

Recommendation Address the minor issues reported above.

Developers Response Developers have not acknowledged the issue yet.

5.2 Detailed Description of Issues in Validation

5.2.1 V-HOTB-VUL-001: SDK is missing nonce cleaning signature generation function

Severity	High	**Commit**	d0d45d9
Type	Missing functionality	**Status**	Confirming Fix
File(s)	`src/internals.rs`		
Location(s)	src/internals.rs		
Confirmed Fix At	N/A		

Description The current protocol workflow for fund withdrawals requires that nonces be cleaned up after they have been used. This can be done using one of the following functions on the NEAR bridge: `clear_completed_withdrawal()`, `clear_withdrawal_by_receiver()`, or `clear_withdrawal_by_sender()`. In the first case, the caller must provide a valid signature from the MPC service, generated specifically for the nonce cleanup request. However, this functionality is currently not implemented.

Impact It is currently not possible to generate nonce-cleaning signatures compatible with the `clear_completed_withdrawal()` function. Even more concerning, the `clear_withdrawal_by_receiver()` function is not fully implemented for Stellar-targeting withdrawals, leaving `clear_withdrawal_by_sender()` as the only viable option. However, if the sender and receiver are not the same party, coordinating this action may prove infeasible in practice.

Recommendation It is required to implement the corresponding logic within the SDK.

Developers Response Developers confirmed the issue and agreed to provide a fix.

5.2.2 V-HOTB-VUL-002: Inaccurate criteria for signature generation can result in loss of funds

Severity	Medium	**Commit**	d0d45d9
Type	Logic Error	**Status**	Fixed
File(s)	`src/internals.rs`		
Location(s)	src/internals.rs:181-183		
Confirmed Fix At	`https://github.com/hot-dao/hot-validation-sdk/pull/4`		

Description To make an informed decision about a deposit or withdrawal request, the MPC service queries multiple nodes to independently verify the request's validity. However, in some cases, the current implementation can be exploited to produce a signature even when the majority of nodes do not agree on the operation's validity. The MPC service reaches a consensus on decision when it receives `>= treshold` of the same answers.

Impact Consider a scenario where the service is configured with a `threshold` of 2 and has 5 servers in total. Suppose two servers - potentially malicious - return `Some(true)`, while the remaining three honest servers return `Some(false)`. Although the honest majority believes the request should not be signed, the current implementation may produce different outcomes depending on the order in which responses were received. For example, it might return `Some(true)` if the confirming responses arrive before the others.

To execute a successful attack, an adversary would need to control or bribe at least the `threshold` number of servers. While this may be challenging or infeasible depending on the context, obtaining even one bogus signature could have catastrophic consequences for the protocol, including the potential complete draining of funds from the Locker.

Recommendation The `threshold` value must be set to at least `len(servers) / 2 + 1` to ensure a majority voting scheme. It is also recommended to have at least 5 independent servers.

Developers Response The developers have added validation logic when deserializing configurations that checks that the threshold is strictly more than half of the number of servers.

5.2.3 V-HOTB-VUL-003: Servers collection allows for duplicates

Severity	Low	**Commit**	d0d45d9
Type	Data Validation	**Status**	Fixed
File(s)	`src/stellar.rs`		
Location(s)	src/stellar.rs:102		
Confirmed Fix At	`https://github.com/hot-dao/hot-validation-sdk/pull/4`		

Description The collection of `servers`, which stores the addresses of nodes responsible for validating requests, is currently implemented as a Vector type that allows duplicates. If duplicate entries are present, the validation process becomes unfair and vulnerable to manipulation.

Impact If multiple duplicates exist in the collection, vote manipulation becomes feasible at low threshold values.

Recommendation Enforce the `servers` collection to not contain duplicate entries.

Developers Response The developers have added validation logic when deserializing configurations that checks that the provided servers are unique.

5.2.4 V-HOTB-VUL-004: Maintainability issues

Severity	Warning	**Commit**	d0d45d9
Type	Maintainability	**Status**	Fixed
File(s)	`src/internals.rs`, `src/evm.rs`		
Location(s)	src/internals.rs, src/evm.rs		
Confirmed Fix At	06ba9ce		

Description During the manual code review of the validation SDK, several maintainability issues were discovered.

1. In `src/internals.rs:131-135`, the `wallet_id` and `metadata` are said to only be used in NEAR. However, they are never validated to be empty in the EVM and Stellar verifiers.
2. In `src/internals.rs:134`, the `metadata` field is said to only be used in NEAR. However, the NEAR bridge will ignore this field as it is not part of its function parameters.
3. In `src/internals.rs:132`, the docstring for the `user_payload` field mentions that it represents "something else" on NEAR, but looking at the code on the NEAR bridge it appears to also be a nonce.
4. In `src/evm.rs`, the tests defined in the module may fail due to rate limitations on the used RPC URL.
5. In `src/lib.rs`, the chain code for NEAR is set to `0`, while on the NEAR side another code value is used `1010`

Impact Although the above issues are maintainability issues with no security impact, leaving them unfixed will increase the likelihood of future bugs and increase the difficulty of extending the code base.

Recommendation Address the minor issues reported above.

Developers Response Developers have addressed most of the observations, except for item 4.

5.2.5 V-HOTB-VUL-005: EVM verifier config is taken from Stellar entries by mistake

Severity	Warning	**Commit**	d0d45d9
Type	Data Validation	**Status**	Fixed
File(s)	`src/lib.rs`		
Location(s)	src/lib.rs:108		
Confirmed Fix At	`https://github.com/hot-dao/hot-validation-sdk/pull/3`		

Description When initializing the validation SDK, the constructor of the `Validation` struct takes a set of configuration objects for different chain IDs. It then constructs the appropriate verifiers for NEAR, Stellar and EVM chains. When constructing the verifier for EVM chains, it filters out the configuration object for the NEAR chain, before constructing a verifier for the remaining chains. The issue is that it does not filter out the configuration object for the Stellar chain.

```
pub fn new(configs: HashMap<ChainId, ChainValidationConfig>) -> Result<Self> {
    // ...
    let evm_validation = configs
        .into_iter()
        .filter(|(id, _)| *id != ChainId::Near) // [VERIDISE]: the same filtering
    should be applied to ChainId::Stellar
        .map(|(id, config)| {
            let threshold_verifier = {
                let verifier = ThresholdVerifier::new_evm(config, client.clone());
                Arc::new(verifier)
            };
            (id, threshold_verifier)
        })
        .collect();
    // ...
}
```

Snippet 5.6: Snippet from `Validation::new()`

Impact As a result, the SDK will construct an EVM verifier using Stellar RPC endpoints. Currently, this erroneous verifier will not be used but future changes could lead to unexpected behavior.

Recommendation Filter out Stellar configs when building EVM verifiers, as is done for NEAR configs.

Developer Response The developers have changed the filter to check that the IDs are EVM IDs (instead of *not-NEAR* IDs).

5.3 Detailed Description of Issues in Locker

5.3.1 V-HOTB-VUL-001: Lack of domain separation between deposit and withdrawal messages allows invalidating arbitrary nonces

Severity	Medium	**Commit**	65520e7
Type	Cryptographic Vulnerabilit	**Status**	Acknowledged
File(s)	`src/lib.rs`, `src/lib.rs`		
Location(s)	src/lib.rs:230, src/lib.rs:156		
Confirmed Fix At	N/A		

Description When transferring tokens between the Stellar locker contract and the NEAR bridge contract, the protocol requires a payload to be signed by the MPC service.

- **For deposits**: The Stellar locker contract generates this payload and stores it in its contract storage until the MPC service signs it.
- **For withdrawals**: The Stellar locker contract reconstructs the expected payload and verifies it against a provided MPC signature.

However, there's an issue: the payload structure lacks domain separation, meaning there's no explicit indicator within the payload to distinguish whether it pertains to a **deposit** or a **withdrawal**.

Additionally, both `deposit()` and `withdraw()` functions use nonce values to prevent replay attacks. Withdrawal nonces are generated and assigned by the NEAR bridge.

Deposit nonce values (called `client_timestamp` in the `deposit()` function) are arbitrarily provided by the user. This design decision creates a potential **griefing attack** that could result in legitimate users losing access to their funds.

Impact Here's how an attacker could exploit this vulnerability:

1. **Legitimate Withdrawal Initiated:**
   - User **Alice** initiates a withdrawal on the NEAR bridge by calling `withdraw()`.
   - This burns her wrapped NEAR tokens and generates a unique **withdrawal nonce** `n`.
2. **Attacker Front-Runs with a Deposit:**
   - Attacker **Mallory** calls `deposit()` on the Stellar locker.
   - She sets her own Stellar address as the receiver: this is essential needed to carry out the attack w/o loosing tokens.
   - Importantly, she **sets** `n` **(Alice's nonce) as the** `client_timestamp` in her deposit request.
3. **Attacker Obtains a Signature:**
   - Mallory gets an MPC signature `s` for her deposit payload.
4. **Attacker Withdraws Using Reused Payload:**
   - Mallory then calls `withdraw()` on the Stellar locker.

- Because the payload lacks domain separation, the Stellar locker accepts `s` as a valid withdrawal signature, even though it was generated for a deposit.
- Mallory retrieves her deposited tokens.
- The nonce `n` is now marked as used.

5. **Legitimate User's Withdrawal Fails:**
   - Later, Alice obtains an MPC signature for her legitimate withdrawal using nonce `n`.
   - When she calls `withdraw()` on the Stellar locker, the contract rejects the transaction because nonce `n` has already been marked as used.
   - As a result, **Alice's withdrawal fails**, and she loses her tokens.

Recommendation Add some kind of domain separation to the payloads corresponding to withdrawals and deposits.

Developers Response The issue was recognized; however, a code-level fix was not implemented. Instead, the approach relies on the assumption that distinct public keys will be used for signing deposit and withdrawal payloads.

5.3.2 V-HOTB-VUL-002: Change of owner does not use two step pattern

Severity	Warning	**Commit**	65520e7
Type	Access Control	**Status**	Fixed
File(s)	`src/lib.rs`		
Location(s)	src/lib.rs:62-66		
Confirmed Fix At	`https://github.com/hot-dao/stellar-hot-bridge/pull/2`		

Description The `set_owner()` function allows the current contract owner to transfer ownership to a new address by simply updating the `Owner` storage key. The new owner address is accepted and stored immediately without requiring confirmation or acknowledgment from the recipient. Making such critical changes in a single step can be error prone and lead to irrecoverable mistakes.

Impact If an incorrect address is accidentally set as the owner, all privileged functionality will become inaccessible. As a result, critical methods such as `owner_withdraw()`, `increase_instant_ttl()`, `clear_deposit()`, `set_public_key()`, and `set_owner()` can no longer be invoked.

Recommendation It is recommended to implement a two-step ownership transfer process, where the new owner must confirm the acceptance of ownership before ownership from the previous owner is revoked.

Developers Response The developers have implemented a two-step ownership transfer proccess.

5.3.3 V-HOTB-VUL-003: Maintainability issues

Severity	Warning	**Commit**	65520e7
Type	Maintainability	**Status**	Fixed
File(s)	`src/lib.rs`		
Location(s)	src/lib.rs		
Confirmed Fix At	`https://github.com/hot-dao/stellar-hot-bridge/pull/1`		

Description During the manual code review of the Stellar locker contract, several maintainability issues were discovered in the `src/lib.rs` file:

1. On line 80, there is a typo in the function name; "instant" instead of "instance".
2. When creating withdrawal and deposit payloads, the `withdraw()` and `deposit()` functions manually append a chain ID to the payload (lines 150 and 225). It would be better to store this chain ID as a named constant.
3. Line 240 seems to be redundant with the operation performed in lines 215-218.
4. On line 193, the `client_timestamp` parameter for the `deposit()` function is an unclear name. In reality, this parameter is the actual nonce of the deposit and not a timestamp.
5. On line 316, the `wallet_id` parameter of the `hot_verify()` function is not used.
6. The `clear_deposit()` function is redundant and should be removed, as the TTL mechanism already handles the expiration of nonce entries. Moreover, since the function does not verify the freshness of a nonce, it poses a risk of inadvertently locking funds if the owner mistakenly removes a still-valid nonce.

Impact Although the above issues are maintainability issues with no security impact, leaving them unfixed will increase the likelihood of future bugs and increase the difficulty of extending the code base.

Recommendation Address the minor issues reported above.

Developers Response The developers have not acknowledged the issue yet.

5.3.4 V-HOTB-VUL-004: Incorrect order of arguments when extending instance TTL

Severity	Warning	**Commit**	65520e7
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib.rs`		
Location(s)	src/lib.rs:85		
Confirmed Fix At	`https://github.com/hot-dao/stellar-hot-bridge/pull/1`		

Description The `increase_instant_ttl()` function allows the owner address to increase the TTL of the contract instance storage. However, currently the order of the arguments is incorrect.

```
pub fn increase_instant_ttl(env: Env) {
    // ...
    env.storage()
        .instance()
        .extend_ttl(INSTANCE_BUMP_AMOUNT, INSTANCE_LIFETIME_THRESHOLD);
}
```

Snippet 5.7: Snippet from `increase_instant_ttl()`

```
pub fn extend_ttl(&self, threshold: u32, extend_to: u32)
```

Snippet 5.8: Signature of `extend_ttl()` from `soroban_sdk::storage::Instance`

Impact As a result, calling `increase_instant_ttl()` will increase the instance TTL to `INSTANCE_LIFETIME_THRESHOLD` (100 days) instead of `INSTANCE_BUMP_AMOUNT` (120 days).

Recommendation Change the order of the arguments.

Developer Response The developers have changed the order of the arguments.

6 Penetration Testing Methodology & Scope

6.1 Methodology

As part of this audit, Veridise security analysts conducted penetration testing of the Hot Bridge with a particular focus on the NEAR and Stellar components of the protocol. The goal of this dynamic analysis was to uncover vulnerabilities and behaviors inconsistent with the provided system specification. While the protocol relies on a distributed MPC network to authorize cross-chain operations, this component was out of scope for this engagement. To accurately simulate the protocol's full behavior in a controlled setting, our team developed a custom mock MPC signer to emulate the threshold ECDSA signatures required by both bridge endpoints.

Setting up a representative local testing environment required substantial effort. At the time of the engagement, the Hot Bridge codebase did not offer scripts for local network deployment. As a result, the Veridise pentesting team manually deployed the bridge contracts on a local NEAR node and the Locker contract on a local Stellar network. The lack of such scripts increased the setup overhead but was necessary to ensure the accuracy of the penetration efforts performed by the Veridise security analysts.

Due to the limited time frame of the engagement, penetration testing efforts were focused on the most security-critical aspects of the bridge—namely, those that move user funds (such as the deposit and withdraw flows), functions responsible for minting and burning assets, and privileged administrative operations like key rotations. API endpoints and view methods not directly involved in these security-sensitive pathways were de-prioritized. Therefore, full coverage of the protocol's API was not achieved, and test coverage was intentionally focused on high-risk functionality.

The pentesting team operated in close coordination with the manual auditing team throughout the engagement. Regular syncs between the two groups allowed for dynamic re-prioritization of testing targets based on in-progress findings, discovery of complementary attack vectors, and the development of proof-of-concept scripts for reproducing complex or subtle issues. This collaborative workflow ensured that pentesting efforts were always focused where they could deliver the most security insight.

A significant portion of the testing effort was dedicated to negative test cases, due to the absence of such tests in the Hot Bridge codebase. Many of the bridge's critical functions had no automated checks for rejecting malformed inputs, replayed transactions, out-of-bounds values, or unauthorized callers. The pentesting suite therefore emphasized validation of such conditions—confirming that deposit and withdrawal paths fail safely when fed with incorrect signatures, reused nonces, invalid asset identifiers, or stale timestamps. Where appropriate, the team also validated "happy path" flows.

Finally, we recommend that the Hot Bridge development team incorporate the pentest suite created during this engagement directly into their internal repository. Doing so would provide multiple benefits: first, it would allow the bridge team to run these tests continuously during development via their CI, reducing the risk of regressions; second, it would furnish future

contributors with reliable deployment and testing tools for local environments; and third, it would extend the current test suite to include a breadth of security-driven validations currently missing from the codebase.

6.2 Performed tests

Table 6.1 provides a summary of the tests performed by the Veridise security analysts. For each row, the table contains a short description as well as the expected and actual behavior of the system. Specifically, when the expected behavior of the system is marked as "PASSES," it means that the protocol is expected to allow the scenario as described in the first column of the table. Conversely, rows marked as "FAILS" mean that the protocol is expected to disallow the described scenario. Two of the rows are marked "UNKNOWN" because it was not clear what was the expected behavior of the system and the tests did not break any core functionality of the protocol, but they might have some security implications. Finally, note that each of the entries of the table may contain multiple sub-tests and exercise multiple functions of the Hot Bridge.

Analysis of Results. As table 6.1 demonstrates the protocol matched the expectation of the Veridise analysts for 12 of the scenarios they attempted. For the tests whose expected behavior is marked as "UNKNOWN," the Veridise analysts noted that cross-chain withdrawals and deposits allowed to be submitted with their value amount set to zero. As demonstrated by V-HOTB-VUL-007, this exposes an attack surface where malicious actors can hinder usage of the protocol with minimal cost. The rest of tests did deviate from the expectations of the Veridise analysts and all these cases have been reported to the Hot Bridge development team as issues in this report.

Table 6.1: Penetration Tests Summary. If an actual behavior contains a "(partially)" qualifier, this means that the protocol allowed or disallowed only a sub-set of the performed tests.

Test Description	Expected Behavior	Actual Behavior
Performs a cross-chain deposit from Stellar to NEAR	PASSES	PASSES
Performs a cross-chain deposit from Stellar to NEAR with amount set to zero	UNKNOWN	PASSES
Performs a cross-chain withdrawal to a Stellar asset that is not the native currency	PASSES	PASSES
Same as cross-chain-withdraw, but the receiver is the issuer of the asset on stellar	PASSES	PASSES
Performs a cross-chain withdrawal to a Stellar asset that is not the native currency with amount set to zero	UNKNOWN	PASSES
Initiate a deposit on NEAR to a chain that hasn't been added yet	FAILS	FAILS
Performs a deposit with invalid key pair	FAILS	FAILS
Initiating a deposit without enough balance	FAILS	FAILS
Calls all owner-only function on the NEAR bridge and signs the TX with another user	FAILS	PASSES (partially)
Initiates a normal deposit on Stellar (i.e., without any message) and then forwards it to the "mt_deposit_call" function on NEAR	FAILS	PASSES
Attack scenario where a user steals funds from the bridge, in case the bridge happens to have some balance in one of its tokens	FAILS	PASSES
Nonce provided for withdraw is 31 days in the future	FAILS	FAILS
Nonce provided for withdraw is older than allowed	FAILS	FAILS
Nonce provided for withdraw has already been used	FAILS	FAILS
Calls all owner-only function on the Stellar locker and signs the TX with another user	FAILS	FAILS
Transfer tokens when sender does not have enough balance	FAILS	FAILS
Performs a batch transfer of tokens where the sender does not have enough token balance	FAILS	FAILS

A Appendix

A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-HOTB-APP-VUL-001: Lack of full access authentication in privileged functions

Severity	Low	**Commit**	356f6eb
Type	Authorization	**Status**	Intended Behavior
File(s)	`contract/src/owner_methods.rs`, `contract/src/standards/nep245/metadata.rs`, `contract/src/withdrawals/clear.rs`		
Location(s)	contract/src/owner_methods.rs,contract/src/standards/nep245/metadata.rs,contract/src/withdrawals/clear.rs:81		

Description In NEAR smart contracts, it is standard security practice to include an `assert_one_yocto()` check in privileged functions to ensure that only full-access keys (not function-call access keys) can invoke them. This requirement is especially important for functions gated by `env::predecessor_account_id() == self.owner_id`. More information on this topic can be found in the NEAR documentation.

In the audited codebase, multiple owner-restricted functions lack this check. These privileged functions all check that the predecessor matches a particular address but do not call `assert_one_yocto()`:

1. In `contract/src/owner_methods.rs`:
   - `owner_clear_withdraws()`
   - `owner_clear_used_nonces()`
   - `add_chain()`
   - `rotate_public_key()`
   - `set_locker_address()`
2. In `contract/src/standards/nep245/metadata.rs`:
   - `set_token_metadata()`
   - `remove_token_metadata()`
3. In `contract/src/withdrawals/clear.rs`:
   - `clear_withdrawal_by_sender()`

Impact Without asserting that some amount of yoctoNEAR is attached, the contract cannot distinguish whether the call was made with a full-access key (intended) or a function-call access key (potentially unintended).

Recommendation Add `assert_one_yocto()` at the beginning of all owner-only or privileged functions.

Developer Response Developers clarified that the described behavior is by design.

A.1.2 V-HOTB-APP-VUL-002: Change of owner does not use two step pattern

Severity	Warning	**Commit**	356f6eb
Type	Access Control	**Status**	Intended Behavior
File(s)	`contract/src/owner_methods.rs`		
Location(s)	contract/src/owner_methods.rs:17-25		

Description The `change_owner()` function allows the current contract owner to transfer ownership to a new account. The new owner address is accepted and updated immediately without requiring confirmation or acknowledgment from the recipient. Making such critical changes in a single step can be error prone and lead to irrecoverable mistakes.

Impact If an incorrect address is accidentally set as the owner, all privileged functionality will become inaccessible. As a result, methods such as `owner_clear_withdraws()`, `owner_clear_used_nonces()`, `add_chain()`, `rotate_public_key()`, and `set_locker_address()` can no longer be invoked.

Recommendation It is recommended to implement a two-step ownership transfer process, where the new owner must confirm the acceptance of ownership before ownership from the previous owner is revoked.

Developers Response Developers clarified that the described behavior is by design.

A.1.3 V-HOTB-APP-VUL-003: Discrepancy between NEAR and Stellar freshness intervals

Severity	Warning	**Commit**	356f6eb
Type	Usability Issue	**Status**	Intended Behavior
File(s)	`contract/src/owner_methods.rs`		
Location(s)	contract/src/owner_methods.rs:38		

Description The NEAR bridge and Stellar lockers make use of nonces for withdrawals and deposits in order to prevent replay attacks. Additionally, nonces are required to be used to finalize a transaction (withdrawal or deposit) within a given time interval. After this interval, the nonces are deemed to be stale and the user will not receive any tokens on NEAR or Stellar for their operation.

However, this interval is not the same on both sides of the protocol. In particular, nonces on the NEAR bridge are valid for 18 days and nonces on the Stellar locker are valid for 25 days.

Impact This discrepancy can lead to unexpected behavior for users of the protocol, in particular:

- Withdrawals that have been removed from the bridge due to being stale can still be redeemed on the Stellar locker.
- Once Double spend attack due to missing nonce freshness check is fixed by checking for nonce staleness in `deposit()`, a user could obtain an MPC signature for a deposit that is still live on the Stellar locker but cannot be redeemed on the NEAR bridge.

Recommendation Change the intervals so that they are the same on both ends of the protocol or clearly document this discrepancy in the Stellar locker contract.

Developer Response The developers clarified that this time gap is intentional, and they will include a corresponding warning in the documentation.