AUDITONE

Audit Report

AURORA

Omni Bridge

05.11.2024

Table of Contents

CONTENTS

INTRODUCTION

Smart Contract Security Analysis Report

Note: This report may contain sensitive information on potential vulnerabilities and exploitation methods. This must be referred internally and should be only made available to the public after issues are resolved (to be confirmed prior by the client and AuditOne).

INTRODUCTION

Minhquanym, Ubermensch3dot0, Mario Poneder and Zzzuhaibmohd, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Omni Bridge. The audit has been performed using manual analysis. This report presents all the findings regarding the audit performed on the customer's smart contracts. The report outlines how potential security risks are evaluated. Recommendations on quality assurance and security standards are provided in the report.

01-PROJECT DESCRIPTION

OmniBridge is a multi-chain asset bridging platform designed to facilitate secure and efficient asset transfers across blockchain networks. Using advanced technologies such as NEAR Multi-Party Computation (MPC) and Chain Signatures, OmniBridge ensures trustless and decentralized cross-chain transactions. The platform bridges the gap between EVM-compatible and non-EVM blockchains, enabling seamless interoperability in a decentralized ecosystem.

This audit focuses on the integrity and security of the bridging contracts.

02-Project and Audit Information

Term	Description
Auditor	Minhquanym, Ubermensch3dot0, Mario Poneder and Zzzuhaibmohd
Reviewed by	Luis Buendia and Gracious Igwe
Type	Cross-chain bridging
Language	Solidity & Rust
Ecosystem	Near
Methods	Manual Review
Repository	https://github.com/Near-One/omni-bridge
Commit hash (at audit start)	ab0b632b8f986cb028ff5080eb40db0222ecda25
Commit hash (after resolution)	288da917fbe24fa2e4a8ce61c2bf90db8139f60a
Documentation	N/A
Unit Testing	N/A
Website	https://aurora.dev/
Submission date	15/10/24
Finishing date	05/11/24

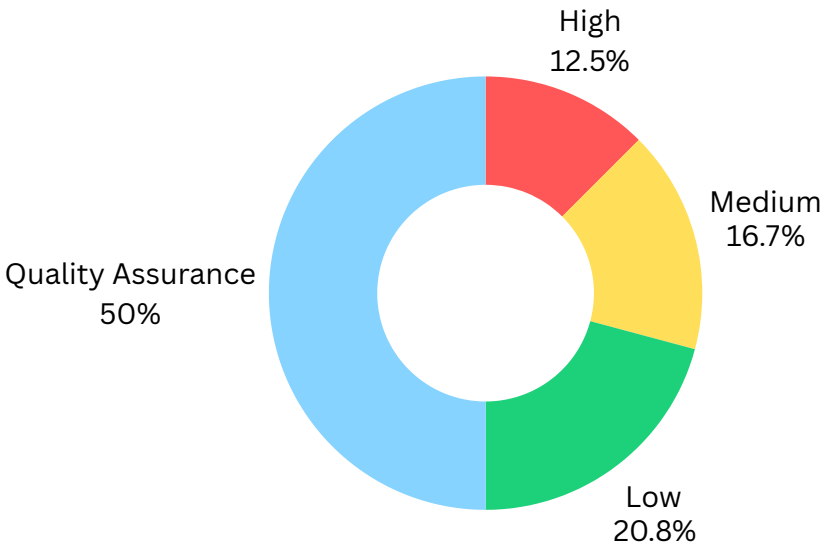
03-Contracts in Scope

Contract in scope:

- contracts/BridgeTokenFactoryWormhole.sol
- contracts/BridgeTokenFactory.sol
- contracts/Borsh.sol
- contracts/BridgeToken.sol
- contracts/SelectivePausableUpgradable.sol
- omni-types/src/lib.rs
- omni-types/src/locker_args.rs
- omni-types/src/evm/events.rs
- omni-types/src/evm/header.rs
- omni-types/src/evm/mod.rs
- omni-types/src/evm/receipt.rs
- omni-types/src/evm/utils.rs
- omni-types/src/mpc_types.rs
- omni-types/src/near_events.rs
- omni-types/src/prover_result.rs
- omni-types/src/prover_args.rs
- nep141-locker/src/lib.rs
- nep141-locker/src/errors.rs
- nep141-locker/src/storage.rs
- omni-prover/evm-prover/src/lib.rs
- wormhole-omni-prover-proxy/src/lib.rs
- wormhole-omni-prover-proxy/src/parsed_vaa.rs
- wormhole-omni-prover-proxy/src/byte_utils.rs
- omni-prover/src/lib.rs

04-Executive summary

Magicsea Booster smart contracts were audited between 07-08-2024 and 23-08-2024 by Minhquanym. Manual analysis was carried out on the code base provided by the client. The following findings were reported to the client. For more details, refer to the findings section of the report.



Issue Category	Issues Found	Resolved	Acknowledged
High	3	2	1
Medium	4	4	0
Low	5	2	3
Quality Assurance	8	2	6

05-Severity Definitions

Risk factor matrix	Low	Medium	High
Occasional	L	M	H
Probable	L	M	H
Frequent	M	H	H

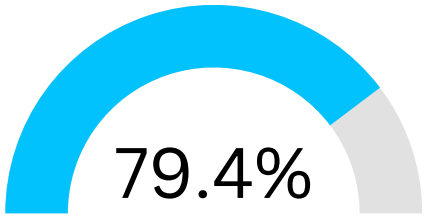
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

Medium: The impact of medium issues is less critical than high, but still probable with considerable damage. The protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions.

Low: Low issues impose a small risk on the project. Although the impact is not estimated to be significant, we recommend fixing them on a long-term horizon. Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

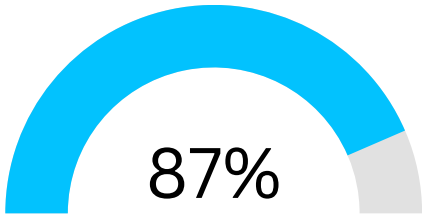
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



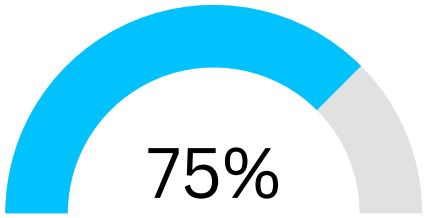
Security score

Security score is a numerical value generated based on the vulnerabilities in smart contracts. The score indicates the contract's security level and a higher score implies a lower risk of vulnerability.



Code quality

Code quality refers to adherence to standard practices, guidelines, and conventions when writing computer code. A high-quality codebase is easy to understand, maintain, and extend, while a low-quality codebase is hard to read and modify.



Documentation quality

Documentation quality refers to the accuracy, completeness, and clarity of the documentation accompanying the code. High-quality documentation helps auditors to understand business logic in code well, while low-quality documentation can lead to confusion and mistakes.

07-Findings

Finding: #1

Issue: Inverted **require** condition leading to transfer DoS and potentially stuck funds

Severity: High

Where: lib.rs#L778

Impact: No new transfer messages can be added leading to the following consequences:

- On NEAR -> ETH transfer (**ft_on_transfer** method calls **add_transfer_message**): DoS with "**ERR_KEY_EXIST**" on every transfer.
- On ETH -> ETH transfer via NEAR (**fin_transfer_callback** calls **add_transfer_message**): DoS with "**ERR_KEY_EXIST**" on every transfer, therefore potentially stuck funds (on ETH side) due to impossible processing of transfer (on NEAR side).

Description: The internal **add_transfer_message** method of the **nep141-locker** intends to add a transfer message with a given nonce to a pending message queue using the underlying **insert_raw_transfer** method.
Thereby, the **insert_raw_transfer** method returns the previous serialized message if one already existed at the given nonce.
Furthermore, the **add_transfer_message** method intends to check if the nonce is indeed unique, i.e. there should not be a previous serialized message.
However, this check was accidentally inverted, so the **add_transfer_message** method only allows to overwrite messages with existing an existing **nonce** and does not allow to add new messages with a unique **nonce**.

Recommendation: Invert the condition in the **require** macro:

```
    require!(
        self.insert_raw_transfer(nonce, transfer_message, message_owner)
-           .is_some(),
+           .is_none(),
        "ERR_KEY_EXIST"
    );
```

Status: Resolved.

Finding: #2

Issue: Failure to Handle **ft_on_transfer** Failure May Result in Loss of Funds

Severity: High

Where: lib.rs#L497-L500

Impact: If the recipient's **ft_on_transfer** function fails during a **ft_transfer_call**, the tokens are refunded to the **nep-141-locker** contract instead of the user. Since the transfer is marked as finalized and the nonce is used, the user loses access to their funds, leading to loss of funds without the possibility of recovery.

Description: In the **fin_transfer_callback** function, when transferring tokens to a NEAR account with an attached message, the contract uses **ft_transfer_call** as per the NEP-141 Fungible Token standard. This method allows the recipient to execute additional logic in their **ft_on_transfer** hook. If this hook fails—due to reasons like slippage checks, insufficient liquidity, or whitelisting—the NEP-141 standard dictates that the tokens are refunded to the sender.
In this context, the sender is the **nep-141-locker** contract itself. However, the current implementation does not handle this refund scenario. The transfer message is already marked as finalized, and the nonce is consumed. As a result, the user's funds remain locked in the **nep-141-locker** contract, and the user cannot retrieve or reinitiate the transfer, effectively leading to a loss of funds.

Recommendation: Introduce a callback function to process the result of the **ft_transfer_call**. If the transfer fails, update the contract state accordingly to prevent loss of funds, this can be done by removing the **transfer_id** from the **finalised_transfers** mapping or redirecting the refunded amount to the user.

Status: Acknowledged.

Finding: #3

Issue: deploy_token.emitter_address Allows Unauthorized Token Binding.

Severity: High

Where: lib.rs#L660-L674

Impact: Without verifying the **deploy_token.emitter_address** against the known factory address for each chain, anyone can manipulate the **tokens_to_address_mapping**. Attackers could bind arbitrary tokens by deploying contracts on any chain, emitting **DeployToken** events, and submitting proofs. This allows unauthorized tokens to be bound to NEAR tokens, potentially leading to security breaches, token spoofing, and loss of user funds.

Description: The **bind_token_callback** function is responsible for processing the result of the **verify_proof** call, which should only accept proofs from trusted factories. However, the current implementation lacks a critical check to verify that the **deploy_token.emitter_address** matches the expected factory address for the chain of the token address.
Here is the problematic code:

```rust
#[private]
pub fn bind_token_callback(
    &mut self,
    #[callback_result]
    #[serializer(borsh)]
    call_result: Result<ProverResult, PromiseError>,
) {
    let Ok(ProverResult::DeployToken(deploy_token)) = call_result else {
        env::panic_str("Invalid proof message")
    }; // @audit-issue why don't we verify deploy_token.emitter_address

    self.tokens_to_address_mapping.insert(
        &(deploy_token.token_address.get_chain(), deploy_token.token),
        &deploy_token.token_address,
    );
}
```

The lack of verification for **deploy_token.emitter_address** means that the contract does not confirm whether the event originated from a trusted source. An attacker can exploit this by:

1. Deploying a malicious contract on a target chain.
2. Emitting a **DeployToken** event from this contract.
3. Submitting a proof of this event to the **bind_token** function.

As a result, the attacker can manipulate the **tokens_to_address_mapping** by binding their arbitrary token address to a NEAR token. This can lead to:

- **Unauthorized Token Mapping**s: Attackers can bind fake tokens, causing users to interact with malicious tokens believing they are legitimate.
- **User Funds at Risk**: Users may unknowingly bridge tokens to or from malicious contracts, resulting in potential loss of assets.

Recommendation: Modify the **bind_token_callback** function to check that **deploy_token.emitter_address** matches the known factory address for the chain of the token address. This ensures that only events emitted by trusted factories are accepted.

```
require!(
    self.factories.get(&deploy_token.token_address.get_chain()) == Some(deploy_token.emitter_address),
    "ERR_UNKNOWN_FACTORY"
);
```

Status: Resolved.

Finding: #4

Issue: One signature could be reused for different purposes

Severity: Medium

Where: BridgeTokenFactory.sol#L142-L147

Impact: Users can exploit the same signature signed by **nearBridgeDerivedAddress** to perform unauthorized actions across different functions, such as using a signature from **deployToken()** to claim fees in **claimNativeFee()**, potentially draining the contract balance.

Description: In the **BridgeTokenFactory** contract, three functions require a signature from **nearBridgeDerivedAddress** to execute. Specifically:

- **claimNativeFee()**: Allows the caller to claim ETH fees.
- **deployToken()**: Enables the deployment of new tokens.

The lack of a domain separator in the encoded value allows signatures from one function to be reused in another, given that the **borshEncoded** values are identical.

```
function deployToken(bytes calldata signatureData, MetadataPayload calldata metadata) payable external retu
    bytes memory borshEncoded = bytes.concat(
        Borsh.encodeString(metadata.token),
        Borsh.encodeString(metadata.name),
        Borsh.encodeString(metadata.symbol),
        bytes1(metadata.decimals)
    );
    bytes32 hashed = keccak256(borshEncoded);

    if (ECDSA.recover(hashed, signatureData) != nearBridgeDerivedAddress) {
        revert InvalidSignature();
    }
    // ...
}

function claimNativeFee(bytes calldata signatureData, ClaimFeePayload memory payload) external {
    bytes memory borshEncodedNonces = Borsh.encodeUint32(uint32(payload.nonces.length));

    for (uint i = 0; i < payload.nonces.length; ++i) {
        uint128 nonce = payload.nonces[i];
        if (claimedFee[nonce]) {
            revert NonceAlreadyUsed(nonce);
        }

        claimedFee[nonce] = true;
        borshEncodedNonces = bytes.concat(
            borshEncodedNonces,
            Borsh.encodeUint128(nonce)
        );
    }

    bytes memory borshEncoded = bytes.concat(
        borshEncodedNonces,
        Borsh.encodeUint128(payload.amount),
        bytes1(omniBridgeChainId),
        Borsh.encodeAddress(payload.recipient)
    );
    bytes32 hashed = keccak256(borshEncoded);

    if (ECDSA.recover(hashed, signatureData) != nearBridgeDerivedAddress) {
        revert InvalidSignature();
    }
    // ...
}
```

Recommendation: Including the function name into the signature to give it the context, thus cannot be used in different functions.

Status: Resolved.

Finding: #5

Issue: Lack of storage cost accounting when adding entry to **tokens_to_address_mapping**

Severity: Medium

Where: lib.rs#L670-L673

Impact: The lack of storage cost accounting when adding an entry to the **tokens_to_address_mapping** can lead to DoS in case the contract's NEAR balance is insufficient to cover the storage staking costs.
Moreover, this implies a loss for the protocol owner since the storage cost for binding a new token should be covered by the user/relayer.

Description: In contrast to the **pending_transfers** and **finalised_transfers** mappings of the **nep141-locker** contract, the storage staking costs for adding an entry to the **tokens_to_address_mapping** are not taken into account within the **bind_token_callback** which is invoked through the permissionless **bind_token** method.
Furthermore, anyone can invoke the overall process of deploying a new token on ETH side and binding it on NEAR side by calling the permissionless **log_metadata** method.

Recommendation: It is suggested to benchmark the storage usage via **env::storage_usage()** when adding an entry to the **tokens_to_address_mapping** and enforce an attached deposit accordingly.
For reference, see the usage of the internal **update_storage_balance** method when adding an entry to the **pending_transfers** and **finalised_transfers** mappings.

Status: Resolved

Finding: #6

Issue: Insufficient target chain validation of **native_fee_recipient**

Severity: Medium

Where: lib.rs#L474-L480
lib.rs#L621-L627

Impact: Insufficient validation of message origin chain and **native_fee_recipient** chain kind can lead to a mistmatch in terms of fee USD value when claimed on the wrong target chain. Moreover, this leads to a disbalance of native amounts, which are intended to cover native fees, in the respective EVM side **BridgeTokenFactory** contracts .

Description: The amount of native fee is always part of the transfer message as **message.fee.native_fee** and therefore subject to verification via signature. Moreover, the actual USD value of this native fee depends on the target chain where this native fee is claimed, since the native currencies of Ethereum, Solana, Arbitrum, Base, etc. partially have different valuations in terms of USD and decimals.
Therefore, it is crucial to enforce that the target chain of **native_fee_recipient**, which can be freely & permissionlessly specified by the user/relayer, matches the source chain of the transfer message, see also get_origin_chain method.
However, such a check is never performed and the origin chain of the message could differ from the chain kind of **native_fee_recipient**.

Recommendation: It is suggested to enforce that the chain kind of **native_fee_recipient** matches the origin chain of the transfer message before adding the native fee entry to the **finalised_transfers** mapping.

Status: Resolved

Finding: #7

Issue: Incorrect Fee Difference Calculation Causes **update_transfer_fee** to Always Revert When Increasing Native Fee

Severity: Medium

Where: lib.rs#L280-L284

Impact: Users are unable to increase the native fee for their transfers because the function always reverts due to an incorrect calculation. This prevents users from adjusting fees to expedite or facilitate their transfers, leading to potential delays and a poor user experience.

Description: The **update_transfer_fee** function allows users to update the fees associated with a transfer. When a user wants to increase the native fee, they should attach the difference as a deposit. However, there is a logical error in the calculation of the fee difference, causing the function to revert whenever the user attempts to increase the native fee.

The problematic code is as follows:

```
let diff_native_fee = current_fee
    .native_fee
    .0
    .checked_sub(fee.native_fee.0)
    .sdk_expect("ERR_LOWER_FEE");
```

In this calculation, **current_fee.native_fee.**0 is subtracted by **fee.native_fee.0**. If the user is increasing the native fee (i.e., **fee.native_fee.0** is greater than **current_fee.native_fee.0**), the result of the subtraction will be negative. Since **checked_sub** returns **None** on underflow, the **sdk_expect("ERR_LOWER_FEE")** is triggered, causing the function to revert with an error.

As a result, users cannot increase the native fee, because the function incorrectly interprets the action as an attempt to lower the fee. This logical error effectively blocks users from modifying the native fee upwards.

Recommendation: Reverse the order of subtraction to calculate the correct difference when increasing the native fee. The calculation should be:

```
let diff_native_fee = fee.native_fee.0
    .checked_sub(current_fee.native_fee.0)
    .sdk_expect("ERR_LOWER_FEE");
```

This way, when the user increases the native fee, the difference will be positive, and the function can proceed correctly.

Status: Resolved.

Finding: #8

Issue: Renouncing ownership or admin role could affect the normal operation of Omni-Bridge

Severity: Low

Where: BridgeTokenFactory.sol#L15

Impact: Renouncing ownership or the roles(**DEFAULT_ADMIN_ROLE**, **PAUSABLE_ADMIN_ROLE**) in OmniBridge's **BridgeTokenFactory** contracts could lead to a complete loss of administrative control, rendering the system inoperable. Without an owner/admin, essential functions like updating the metadata (name and symbol) of a bridge token, pausing related functions and whitelisting functionality etc., This creates a significant risk of system paralysis, severely impacting operations and user experience. Therefore, the system's stability and recovery mechanisms would be compromised, leading to potentially irrecoverable issues.

Description: The **BridgeTokenFactory** contract inherits a function called **revokeRole**.

1. If the Owner renounces their role and the only remaining Admin also uses **revokeRole** to revoke their Admin role, the contract is left without both an Owner and Admin.
2. This results in the loss of all administrative control.
3. Critical functions such as pausing/unpausing, updating the contract, and managing whitelisting become non-functional.
4. This is just one example, but several other essential operations are also impacted when both the Admin and Owner roles are revoked.

Similar issues may arise when the user voluntarily calls **renounceRole.**

Recommendation:

1. Review if the **renounceRole** function is required or remove it if it is not needed.
2. Revise the **revokeRole** function to ensure that at least one user always remains in the system who will be responsible for managing all critical operations in case the owner renounces the role.

Status: Acknowledged.

Finding: #9

Issue: Avoid Using **assert_eq!** and **assert!** in Production Code

Severity: Low

Where: src/lib.rs

Impact: Relying on **assert_eq!** and **assert!** can lead to silent failures in production.

Description: The current implementation utilizes **assert_eq!** and **assert!** macros for validating conditions within the code. While assertions are useful during development for catching programming errors, they are removed in release builds, leading to potential silent failures. This practice can result in unexpected behavior in production environments, where critical conditions may not be adequately checked or reported.

Recommendation: Replace all instances of **assert_eq!** and **assert!** with proper error handling mechanisms that return meaningful error messages or types. Utilise Rust's **require**! to handle all these scenarios.

Status: Resolved

Finding: #10

Issue: Signature of **deployToken()** can be reused on different chains

Severity: Low

Where: BridgeTokenFactory.sol#L148

Impact: The signature used in the **deployToken()** function may be reused across different EVM chains, potentially leading to security vulnerabilities.

Description: Unlike the signatures of **finTransfer()** and **claimNativeFee()**, the **deployToken()** function does not incorporate the chain ID (**omniBridgeChainId**). This absence makes it possible for a signature generated on one chain to be valid on another, which could pose risks in a multi-chain environment.

```
function deployToken(bytes calldata signatureData, MetadataPayload calldata metadata) payable external retu
    bytes memory borshEncoded = bytes.concat(
        Borsh.encodeString(metadata.token),
        Borsh.encodeString(metadata.name),
        Borsh.encodeString(metadata.symbol),
        bytes1(metadata.decimals)
    );
    bytes32 hashed = keccak256(borshEncoded);

    if (ECDSA.recover(hashed, signatureData) != nearBridgeDerivedAddress) {
        revert InvalidSignature();
    }
```

The current implementation allows for potential misuse, as signatures can be validated across different chains.

Recommendation: Consider including **omniBridgeChainId** in the signature of **deployToken()** function.

Status: Acknowledged

Finding: #11

Issue: Unrestricted **fee_recipient** in **sign_transfer** method incentivizes front-running.

Severity: Low

Where: lib.rs#L364

Impact: The permsissionless nature of the **sign_transfer** method and its unrestricted **fee_recipient** parameter incentivizes front-running among participants.
Another side-effect of **sign_transfer** being able to create multiple valid signatures for the same message but different **fee_recipient** is that the first one to call **finTransfer** (on ETH side) is not necessarily the same who first calls **claim_fee** (on NEAR end), therefore the payload.feeRecipient of the **FinTransfer** event is potentially misleading/wrong.

Description: The **sign_transfer** method of the **nep141-locker** contract is permissionless and creates a signature for a pending transfer message which can be used to finalize the transfer on the ETH side. This functionality is intended to be used by relayers.
However, the method's **fee_recipient** parameter is unrestricted which incentivizes everyone to be the first to call **sign_transfer** & **claim_fee** while specifying themselves as **fee_recipient**.
Note that **sign_transfer** is able to create multiple valid signatures for the same message but different **fee_recipient**, therefore it is crucial to also be the first one to call **claim_fee**.

Recommendation: It is suggested to implement a whitelist for relayers to mitigate the front-running potential, which is currently possible for everyone.

Status: Acknowledged

Finding: #12

Issue: Unaccounted **ONE_YOCTO** Deposit May Lead to Contract Insolvency Over Time

Severity: Low

Where: lib.rs#L497-L522

Impact: The contract does not account for the **ONE_YOCTO** deposit required for each token transfer when updating fees and balances. While the amount is minimal per transaction, over time, these unaccounted deposits can accumulate, potentially leading to contract insolvency or unintended depletion of funds allocated for storage staking. This could affect the contract's ability to function properly and handle user transactions reliably.

Description: In the **fin_transfer_callback** function, the contract performs token transfers using **ft_transfer** and **ft_transfer_call**, each of which requires an attached deposit of **ONE_YOCTO** (1 yoctoNEAR). However, the contract does not charge the user for these deposits when calculating the required fees or updating the storage balance. Here's the relevant code snippet:

```
// Token transfer to the recipient
let amount_to_transfer = U128(transfer_message.amount.0 - transfer_message.fee.fee.0);
let mut promise = match recipient.message {
    Some(message) => ext_token::ext(transfer_message.token.clone())
        .with_static_gas(FT_TRANSFER_CALL_GAS)
        .with_attached_deposit(ONE_YOCTO)
        .ft_transfer_call(recipient.target, amount_to_transfer, None, message),
    None => ext_token::ext(transfer_message.token.clone())
        .with_static_gas(FT_TRANSFER_GAS)
        .with_attached_deposit(ONE_YOCTO)
        .ft_transfer(recipient.target, amount_to_transfer, None),
};

// Token transfer to the fee recipient (if applicable)
if transfer_message.fee.fee.0 > 0 {
    promise = promise.then(
        ext_token::ext(transfer_message.token.clone())
            .with_static_gas(FT_TRANSFER_GAS)
            .with_attached_deposit(ONE_YOCTO)
            .ft_transfer(
                predecessor_account_id.clone(),
                transfer_message.fee.fee,
                None,
            ),
    );
}
```

The attached deposits of **ONE_YOCTO** for each transfer are necessary but not accounted for in the user's fees or storage balance updates. This means the contract covers these costs from its own balance. Over numerous transactions, this can lead to the contract's balance becoming insolvent, affecting its ability to operate.

Recommendation: Modify the contract to include an additional 2 **yoctoNEAR** charge to the user—one for the recipient transfer and one for the fee recipient transfer. This ensures that the contract does not bear the cost of the attached deposits.

Status: Resolved.

Finding: #13

Issue: **PUSH0** opcode Is Not Supported on popular L2s like Linea etc.,

Severity: Quality Assurance

Where: [bridge-token-factory/contracts]

Impact: Deploying the protocol on Linea or any other EVM chain that does not support **PUSH0** opcode using Solidity version **0.8.24** may lead to unexpected behavior or failures.

Description: The current codebase is compiled with Solidity version **0.8.24**, which generates bytecode containing the **PUSH0** opcode. Since the protocol is a multi-chain asset bridge, the contracts will be currently deployed on Arbitrum, Base etc but in future It may be deployed on several EVM-based Layer 2s, some of which, like Linea, do not yet support the **PUSH0** opcode. This can result in unexpected behavior or even outright failures when deploying and running the smart contracts.

Recommendations: Consider using version **0.8.19** to compile for networks that does not support **PUSH0** opcode.
Also run the given PoC against the target network to verify if **PUSH0** opcode is supported.

Status: Acknowledged.

Finding: #14

Issue: npm dependencies are outdated or deprecated

Severity: Quality Assurance

Where: bridge-token-factory/package.json

Impact: Using an old version can lead to worse gas performance, known contract issues, and potential security and 0-day issues.

Description: Using outdated npm libraries in smart contract development, such as older OpenZeppelin versions, poses significant risks. These libraries may contain security vulnerabilities, lack support for newer EVM opcodes, and may not be compatible with the latest Solidity versions, leading to potential deployment issues. Furthermore, outdated libraries may no longer receive updates or community support, making it harder to fix bugs or address security flaws. They also miss out on important performance optimizations and new features. In smart contract environments where security and efficiency are critical, using up-to-date libraries is essential to ensure reliability and safety.

Recommendations: Update the outdated npm libraries to the latest based on compatibility.

Status: Unresolved.

Finding: #15

Issue: Missing Events in Important Functions

Severity: Quality Assurance

Where: BridgeTokenFactory.sol#L356-L387

Description: The functions **enableWhitelistMode**, **disableWhitelistMode**, **addAccountToWhitelist** and **removeAccountFromWhitelist** in the **BridgeTokenFactory** contract are responsible for toggling the state of the whitelist mode. However, these functions do not emit any events to signal that a change has occurred. Events are crucial in smart contracts as they provide a way to log significant state changes, allowing external observers (like front-end applications or monitoring tools) to react accordingly.

Recommendations: To enhance the transparency and traceability of state changes in the contract, it is recommended to emit events in the **enableWhitelistMode**, **disableWhitelistMode**, **addAccountToWhitelist** and **removeAccountFromWhitelist**. This will allow users and developers to track when the whitelist mode is enabled or disabled.

Status: Resolved.

Finding: #16

Issue: Eliminating Magic Numbers for Improved Code Clarity

Severity: Quality Assurance

Where: storage.rs#L118

Description: In the provided Rust code, several instances of "magic numbers" are present, particularly in the calculations of **key_len** and **value_len** within various methods. Magic numbers are hard-coded values that appear without context, making the code less readable and maintainable. They can lead to confusion for anyone reading the code, as the significance of these numbers is not immediately clear. This can also increase the risk of errors if the same value needs to be updated in multiple places.

Recommendations: To enhance code clarity and maintainability, it is recommended to replace magic numbers with named constants or enums that describe their purpose. For example, instead of using **64 + 4** directly in the code, define a constant like const **KEY_LENGTH: u64 = 64;** and use it in calculations. This approach not only makes the code more self-documenting but also simplifies future modifications, as changes to the constant will automatically propagate throughout the codebase.
Simple Fix:

```
const KEY_LENGTH: u64 = 64;
const VALUE_LENGTH_BASE: u64 = 4;
```

Status: Unresolved.

Finding: #17

Issue: Unbounded Loop in **claimNativeFee**

Severity: Quality Assurance

Where: BridgeTokenFactory.sol#L273

Impact: The unbounded loop in the **claimNativeFee** function can lead to out-of-gas errors if a user submits a large array of nonces, resulting in transaction failures and potential denial of service. Additionally, processing a large number of nonces increases gas costs, which may deter users from utilizing the function effectively.

Description: In the **claimNativeFee** function, there is a loop that iterates over an array of nonces provided in the **ClaimFeePayload** structure. The loop processes each nonce to check if it has already been claimed and marks it as claimed if it hasn't.
The loop iterates over **payload.nonces**, which is an array of nonces. There is no upper limit enforced on the size of this array, meaning that a user could potentially pass a very large array of nonces.

Recommendations: To mitigate this risk, impose a maximum limit on the size of the nonce array (e.g., 100 nonces) to prevent excessive gas consumption. Consider implementing batch processing to allow users to claim fees in smaller groups if they need to process more nonces than the limit allows.

Status: Acknowledged.

Finding: #18

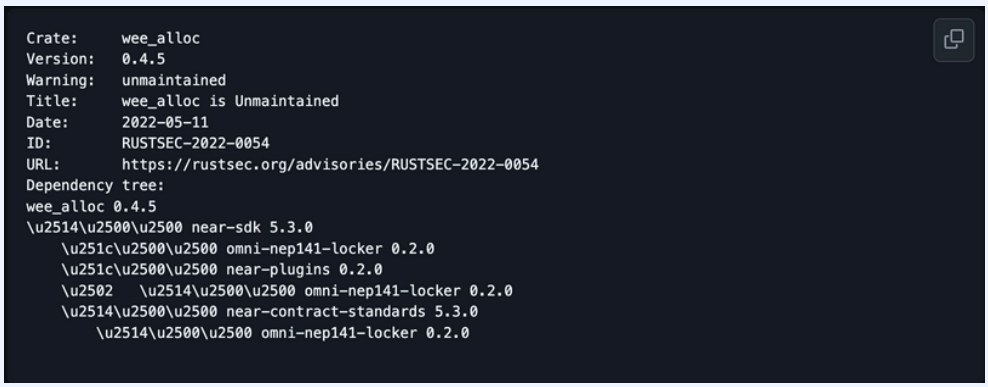
Issue: The **wee_alloc** dependency part of the **near-sdk** is vulnerable to memory leak issues

Severity: Quality Assurance

Where: Cargo.toml#L27

Impact: The crate may not be maintained.
The crate has open issues including memory leaks and may not be suitable for production use.

Description: Two of the maintainers have indicated that the crate may not be maintained.
The crate has open issues including memory leaks and may not be suitable for production use.
Last release seems to have been three years ago.



Recommendations: Even though the code might not be directly using the external library, it is better to switch to latest version. Visit https://crates.io/crates/near-sdk/versions and download the lates the version of **near-sdk.**

Status: Acknowledged.

Finding: #19

Issue: Missing **disableInitializers** call in upgradeable contract constructor

Severity: Quality Assurance

Where: BridgeTokenFactory.sol#L111

Impact: An uninitialized implementation contract can be taken over by an attacker.

Description: A concern arises due to the usage of a proxy upgradeable contract without calling **disableInitializers** in the constructor of the logic contract. This oversight introduces a severe risk, allowing potential attackers to initialize the implementation contract itself.

Recommendations: Call **disableInitializers**: Include a call to **disableInitializers** in the constructor of the logic contract as recommended by OpenZeppelin here https://github.com/OpenZeppelin/openzeppelin-contracts/blob/72c152dc1c41f23d7c504e175f5b417fccc89426/contracts/proxy/utils/Initializable.sol#L43.

Status: Acknowledged.

Finding: #20

Issue: Unused constant

Severity: Quality Assurance

Where: BridgeTokenFactory.sol#L46

Impact: The constant **UNPAUSED_ALL** is defined but never used.

Description: The constant **UNPAUSED_ALL** is defined but never used.

```
uint constant UNPAUSED_ALL = 0; // @audit unused
uint constant PAUSED_INIT_TRANSFER = 1 << 0;
uint constant PAUSED_FIN_TRANSFER = 1 << 1;
```

Recommendations: Consider removing unused values.

Status: Unresolved.

Finding: #21

Issue: Typo error

Severity: Quality Assurance

Where: lib.rs#L213

Impact: Typo in function name

Description: The function name should be **log_metadata_callback** instead of **log_metadata_callback**.

```
pub fn log_metadata(&self, token_id: AccountId) -> Promise {
    ext_token::ext(token_id.clone())
        .with_static_gas(LOG_METADATA_GAS)
        .ft_metadata()
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(LOG_METADATA_CALLBCAK_GAS)
                .with_attached_deposit(env::attached_deposit())
                .log_metadata_callbcak(token_id), // @audit typo
        )
}

#[private]
#[result_serializer(borsh)]
pub fn log_metadata_callbcak( // @audit typo callbcak
```

Recommendations: Consider fixing the typo.

Status: Resolved

Finding: #22

Issue: Ever-growing **finalised_transfers** mapping incurs increasing storage costs

Severity: Quality Assurance

Where: lib.rs#L125

Impact: The perpetual growth of the **finalised_transfers** mapping incurs increasing storage costs which have to be paid by relayers.

Description: In contrast to the **pending_transfers** mapping of the **nep141-locker** contract, entries of the **finalised_transfers** mapping are never removed but only added.

Recommendations: It seems that nonces on the ETH side as welll as on the NEAR side are strictly monotonically increasing. Therefore, it is suggested to save storage costs by enforcing a minimum nonce per chainkind instead of keeping arbitrarily old entries in the **finalised_transfers** mapping. Those "old" entries (below minimum nonce) could be removed when signing the native fee claiming, similar to how it is done with **pending_transfers**.

Status: Acknowledged

Finding: #23

Issue: Lack of message recovery mechanism can lead to irrecoverably stuck funds

Severity: Quality Assurance

Where: Design suggestion affecting whole protocol.

Impact: Funds can become irrecoverably stuck on NEAR/ETH side in case of failing finalization.

Description: In case the finalization of a transfer message fails on NEAR/ETH due to a systematic error and therefore cannot be resolved by retries, the protocol lacks a recovery mechanism which allows to cancel the message and retrieve funds on the source chain.

Recommendations: It is suggested to implement such a recovery mechanism. Referring to well-known bridging protocols here which typically offer a recovery mechanism.

Status: Acknowledged

Finding: #24

Issue: Missing setter for **tokenImplementationAddress**

Severity: Quality Assurance

Where: BridgeTokenFactory.sol#L347-L354

Impact: In case a new bridge token implementation is desired, all freshly deployed bridge tokens have to be manually upgraded after deployment.

Description: The **BridgeTokenFactory** contract offers a method to upgrade the implementation of existing bridge tokens. However, there does not exist a setter method to update the **tokenImplementationAddress** such that future tokens automatically use the recent implementation.

Recommendations: It is suggested to implement a setter method for **tokenImplementationAddress** which should only be accessible via the **DEFAULT_ADMIN_ROLE**.

Status: Unresolved.

08 - Disclaimer

The smart contracts provided to AuditOne have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions). The ethical nature of the project is not guaranteed by a technical audit of the smart contract. Any owner-controlled functions should be carried out by the responsible owner. Before participating in the project, all investors/users are recommended to conduct due research.

The focus of our assessment was limited to the code parts associated with the items defined in the scope. We draw attention to the fact that due to inherent limitations in any software development process and product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which cannot be free from any errors or failures. These preconditions can impact the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure, which adds further inherent risks as we rely on correctly executing the included third-party technology stack itself. Report readers should also consider that over the life cycle of any software product, changes to the product itself or the environment in which it is operated can have an impact leading to operational behaviors other than initially determined in the business specification.

Contact

auditone.io

@auditone_team

hello@auditone.io

A trust layer of our
multi-stakeholder world.