AUDITONE

Audit Report

AURORA

Near Connector

05.06.2024

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

Defec, Ubermensch3dot0 and Vendrell46, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Near Connector. 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

The Aurora environment consists of the Aurora Engine, a high performance EVM—Ethereum Virtual Machine—and the Rainbow Bridge, facilitating trustless transfer of ETH and ERC-20 tokens between Ethereum and Aurora, within a great user experience.

Aurora exists and is operated as an independent, self-funded initiative, but will continue to leverage the shared team DNA and continually evolving technology of the NEAR Protocol.

The governance of Aurora will take a hybrid form of a Decentralized Autonomous Organization—the AuroraDAO—complemented by a traditional entity which will hold one of several seats in the AuroraDAO.

This audit focused on the Near connector, a generic ERC-20/NEP-141 connector for Rainbow Bridge.

02-Project and Audit Information

Term	Description
Auditor	Defec, Ubermensch3dot0 and Vendrell46
Reviewed by	Luis Buendia and Gracious Igwe
Type	Bridge
Language	Rust and Solidity
Ecosystem	Near
Methods	Manual Review
Repository	https://github.com/Near-One/rainbow-token-connector/
Commit hash (at audit start)	0b989fa2d252c4423eba17dd4f1a1fbf0affc0a5
Commit hash (after resolution)	b600af24e2b54608afcf34b29f4bcd49bc0286c3
Documentation	NA
Unit Testing	NA
Website	https://aurora.dev/
Submission date	20/05/2024
Finishing date	03/06/2024

03-Contracts in Scope

Forwarder contracts path

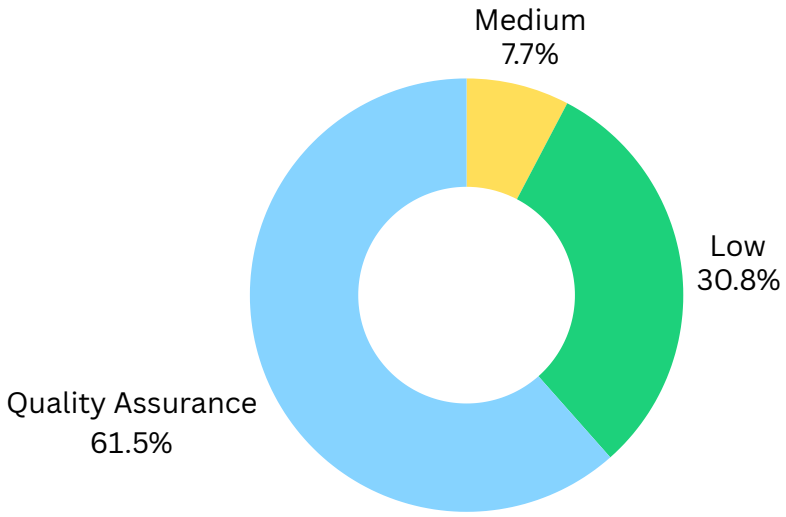
- forwarder/src/error.rs
- forwarder/src/lib.rs
- forwarder/src/params.rs
- forwarder/src/types/address.rs
- forwarder/src/types/promise.rs
- forwarder/src/types/mod.rs
- forwarder/src/types/account_id.rs
- forwarder/src/runtime/io.rs
- forwarder/src/runtime/handler.rs
- forwarder/src/runtime/env.rs
- forwarder/src/runtime/mod.rs
- forwarder/src/runtime/sys.rs
- fees/src/lib.rs
- tests/src/lib.rs
- tests/src/tests/native.rs
- tests/src/tests/mod.rs
- tests/src/tests/wrap.rs
- tests/src/sandbox/factory.rs
- tests/src/sandbox/forwarder.rs
- tests/src/sandbox/fungible_token.rs
- tests/src/sandbox/mod.rs
- tests/src/sandbox/erc20.rs
- tests/src/sandbox/aurora.rs
- utils/src/lib.rs
- factory/src/lib.rs

Controller contracts path

- utils/src/lib.rs
- src/keys.rs
- src/types.rs
- src/lib.rs
- src/event.rs
- src/utils.rs
- src/tests/mod.rs
- src/tests/workspace/upgrade.rs
- src/tests/workspace/delegate.rs
- src/tests/workspace/deploy.rs
- src/tests/workspace/release.rs
- src/tests/workspace/mod.rs
- src/tests/workspace/downgrade.rs
- src/tests/workspace/utils.rs
- src/tests/sdk/mod.rs
- src/tests/sdk/macros.rs

04-Executive summary

Near Connector smart contracts were audited between 20-05-2024 and 03-06-2024 by Defec, Ubermensch3dot0 and Vendrell46. 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	0	0	0
Medium	1	1	0
Low	4	2	2
Quality Assurance	8	1	7

05-Severity Definitions

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

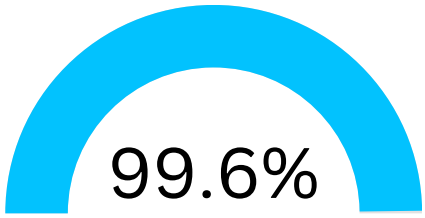
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

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.

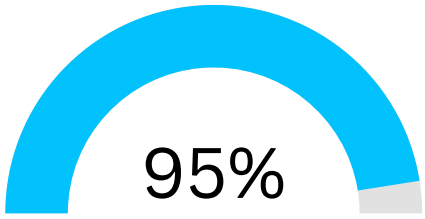
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



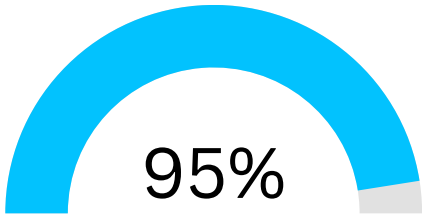
Security score

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: Lack of Critical Check on **nearTokenId** in **newBridgeToken** Function.

Severity: Medium

Where: contracts/BridgeTokenFactory.sol#L100-L134

Impact: The missing check on the **nearTokenId** argument can lead to a Denial of Service (DoS) attack. An attacker can submit a metadata proof with an incorrect **nearTokenId**, causing the **deposit** and **withdraw** functionalities to always revert. This is because the correct **nearTokenId** will not be found in the **_nearToEthToken** and **_isBridgeToken** mappings.

Description: The **newBridgeToken** function lacks a critical validation check for the **nearTokenId** argument. This argument should be required to equal result.token. The absence of this check allows any user to submit a metadata proof with an incorrect **nearTokenId**. As a result, the legitimate token ID will not be recognized in the **_nearToEthToken** and **_isBridgeToken** mappings. This causes the **deposit** and **withdraw** functions to always revert, effectively creating a Denial of Service (DoS) situation for these token functionalities.

Recommendations: Add a validation check in the **newBridgeToken** function to ensure that the **nearTokenId** argument equals **result.token**. This will prevent users from submitting incorrect **nearTokenId** values and ensure that only legitimate token IDs are recognized in the mappings. Another fix is using **result.token** directly and removing the **_nearToEthToken** argument.

Status: Resolved.

Finding: #2

Issue: Inconsistent Amount Type in **withdraw** Function

Severity: Low

Impact: If the amount is intended to be uint128 but is declared as uint256, there is a risk of integer overflow. If the amount value exceeds the maximum value that can be represented by uint128, it will overflow.

Description: In the provided code snippet, the withdraw function of the contract takes an amount parameter of type uint256. However, upon further analysis, it appears that the amount is being handled as uint128 in other parts of the contract.

Solidity Part:

```
    function withdraw(
        string memory token,
        uint256 amount,
        string memory recipient
    ) external whenNotPaused(PAUSED_WITHDRAW) {
        _checkWhitelistedToken(token, msg.sender);
        require(_isBridgeToken[_nearToEthToken[token]], "ERR_NOT_BRIDGE_TOKEN");

        address tokenEthAddress = _nearToEthToken[token];
        BridgeToken(tokenEthAddress).burn(msg.sender, amount);

        emit Withdraw(token, msg.sender, amount, recipient, tokenEthAddress);
    }
```

Near Side:

```
#[near_bindgen]
impl FungibleTokenReceiver for Contract {
    /// Callback on receiving tokens by this contract.
    /// msg: `Ethereum` address to receive the tokens on.
    #[pause(except(roles(Role::DAO, Role::UnrestrictedDeposit)))]
    fn ft_on_transfer(
        &mut self,
        sender_id: AccountId,
        amount: U128,
        msg: String,
    ) -> PromiseOrValue<U128> {
        self.check_whitelist_token(env::predecessor_account_id(), sender_id);
        // Fails if msg is not a valid Ethereum address.
        let eth_address = validate_eth_address(msg);
        // Emit the information for transfer via a separate callback to itself.
        // This is done because there is no event prover and this function must return integer value per FT sta
        ext_self::ext(env::current_account_id())
            .with_static_gas(FT_FINISH_DEPOSIT_GAS)
            .finish_deposit(env::predecessor_account_id(), amount.0, eth_address);
        PromiseOrValue::Value(U128(0))
    }
}
```

Recommendations: Modify the function signature to **function withdraw(string memory token, uint128 amount, string memory recipient) external whenNotPaused(PAUSED_WITHDRAW).**

Status: Resolved

Finding: #3

Issue: Lack of Storage Gaps in **SelectivePausableUpgradable** Contract for Future Upgrades.

Severity: Low

Where: contracts/SelectivePausableUpgradable.sol#L17-L41

Impact: The absence of storage gaps in the **SelectivePausableUpgradable** contract can lead to storage collisions with the parent contract if new state variables are added in future upgrades.

Description: The **SelectivePausableUpgradable** contract is designed to be upgradeable but currently lacks storage gaps. Storage gaps are unused storage slots that can be reserved for future state variables. Without these gaps, any new variables added in future upgrades will collide with existing storage variables in the parent contract.

Recommendations: Introduce storage gaps in the **SelectivePausableUpgradable** contract to reserve space for future state variables. This will ensure that new variables can be added safely without causing storage collisions. The implementation can be updated as follows:

```
abstract contract SelectivePausableUpgradable is Initializable, ContextUpgradeable {
    ...
    ...
    ...
+   // Reserved storage slots to allow for layout changes in the future.
+   uint256[50] private __gap;
}
```

Status: Acknowledged.

Finding: #4

Issue: Single-Step Ownership Transfer in **BridgeToken** Contract

Severity: Low

Where:

- **File:** BridgeToken.sol
- **Contract: BridgeToken**
- **Inheritance: Extends OwnableUpgradeable**

Impact: The **BridgeToken** contract currently uses a single-step ownership transfer mechanism by extending the **OwnableUpgradeable** contract. This means that when ownership is transferred, the new owner is immediately given full control. If an incorrect address is specified during this transfer, ownership can be irretrievably lost. This can have severe consequences for methods marked with **onlyOwner**, which include critical protocol functions.

Description: In the **BridgeToken** contract, ownership is managed using the **OwnableUpgradeable** contract, which implements a single-step transfer of ownership. This approach directly assigns the new owner without any confirmation step. If the address passed to the transfer function is incorrect, it could result in the permanent loss of control over the contract.

Example from **BridgeToken:**

```
contract BridgeToken is Initializable, UUPSUpgradeable, ERC20Upgradeable, OwnableUpgradeable {
    // Contract implementation
}
```

Recommendations: It is a best practice to use a two-step ownership transfer pattern. This involves setting the new owner to a "pending" state, and the new owner must explicitly accept the ownership. This mitigates the risk of accidentally transferring ownership to an incorrect address.

Consider using OpenZeppelin's **Ownable2StepUpgradeable** contract, which implements the two-step ownership transfer pattern.

Steps to implement:

- Update the Inheritance: Replace **OwnableUpgradeable** with **Ownable2StepUpgradeable**.
- Modify the Contract: Update the contract to use **Ownable2StepUpgradeable** for ownership management.

Status: Resolved

Finding: #5

Issue: Silent Failure in **add_account_to_whitelist** Method

Severity: Low

Where: token-locker/src/whitelist.rs#L32-L33

Impact: The caller of the method is not informed whether the account was successfully added to the whitelist or if it already existed.

Description: The **add_account_to_whitelist** method in the provided code has a silent failure vulnerability. The method attempts to add an account to the whitelist using the **insert** function, but it does not handle the case when the account already exists in the whitelist.
When the **insert** function is called with a key that already exists in the map, it returns **false**. However, the **add_account_to_whitelist** method does not check the return value of **insert** and does not provide any feedback or error handling mechanism to indicate that the account was not added to the whitelist due to its presence.

Recommendations: If the account already exists in the whitelist, panic with a meaningful error message indicating that the account is already whitelisted.

Example:

```
pub fn add_account_to_whitelist(&mut self, token: AccountId, account: AccountId) {
    assert!(
        self.whitelist_tokens.get(&token).is_some(),
        "The whitelisted token mode is not set",
    );
    let token_account_key = get_token_account_key(token, account);
    assert!(
        self.whitelist_accounts.insert(&token_account_key),
        "Account is already whitelisted for the given token"
    );
}
```

Status: Acknowledged.

Finding: #6

Issue: Unused Private Variable _pausedFlags in **SelectivePausableUpgradable** Contract.

Severity: QA

Where: contracts/SelectivePausableUpgradable.sol#L41

Impact: The presence of an unused private variable **_pausedFlags** can cause confusion during future development. Developers might mistakenly use this variable instead of **SelectivePausableStorage._pausedFlags**, leading to potential errors or unintended behavior in the contract's functionality.

Description: The **SelectivePausableUpgradable** contract utilizes unstructured storage to avoid storage collision by storing the **SelectivePausableStorage** struct in a specific storage slot determined by **keccak256(abi.encode(uint256(keccak256("aurora.SelectivePausable")) - 1)) & ~bytes32(uint256(0xff))**. However, the contract also defines a private variable **_pausedFlags** that is not used anywhere in the code. This unused variable can cause confusion for developers in future development, as it might be mistaken for the **SelectivePausableStorage._pausedFlags**, leading to potential errors.

Recommendations: Remove the unused private variable **_pausedFlags** from the **SelectivePausableUpgradable** contract to eliminate any confusion and ensure clarity in the code. This will prevent future developers from mistakenly using the wrong variable.

Status: Resolved

Finding: #7

Issue: Deposit Function Lacks Whitelisted Token Check

Severity: QA

Where: BridgeTokenFactory.sol#L150

Impact: Without a whitelisted token check, the deposit function allows the deposit of any token, regardless of its whitelisting status. This means that potentially malicious, unauthorized, or untrusted tokens can be deposited into the contract, compromising the integrity and security of the bridge ecosystem.
Even proof is submitted through relayer, Its recommended to have a this check on the each chain.

Description: The deposit function in the BridgeTokenFactory contract allows users to deposit tokens from the NEAR blockchain to the Ethereum blockchain. However, it has been identified that the function does not include a check to verify if the token being deposited is whitelisted.

Recommendations: Modify the deposit function to include a call to the **_checkWhitelistedToken** function before allowing the deposit of tokens.

Status: Acknowledged.

Finding: #8

Issue: Uninitialized Variables in **ProofConsumer** Contract

Severity: QA

Where: The issue is found in the **ProofConsumer** contract, specifically affecting the variables **prover**, **nearTokenLocker**, and **minBlockAcceptanceHeight**.

Impact: Uninitialized variables can cause significant operational issues, including:

- The contract will revert on critical functions due to failed validations, rendering it unusable.
- The **prover** variable pointing to **address(0)** will cause the **proveOutcome** function to always fail, leading to the failure of any proof validation.
- The **nearTokenLocker** being empty will cause the keccak256 hash comparison to fail, preventing legitimate proofs from being accepted.
- The **minBlockAcceptanceHeight** defaulting to zero will cause the height check in **_parseAndConsumeProof** to always revert, as the block height will always be considered "ancient."

Description: The **ProofConsumer** contract lacks a constructor, resulting in the critical state variables **prover**, **nearTokenLocker**, and **minBlockAcceptanceHeight** being left uninitialized. This leads to:

- The **prover** being set to the zero address.
- The **nearTokenLocker** being an empty byte array.
- The **minBlockAcceptanceHeight** being set to zero.

These uninitialized variables cause the **_parseAndConsumeProof** function to fail in various require statements, which are crucial for the proper validation and processing of proofs from the NEAR blockchain.

Recommendations:

- **Add a Constructor**: Ensure all critical variables are properly initialized during contract deployment.

```
constructor(
    bytes memory _nearTokenLocker,
    INearProver _prover,
    uint64 _minBlockAcceptanceHeight
) {
    require(
        _nearTokenLocker.length > 0,
        "Invalid Near Token Locker address"
    );
    require(address(_prover) != address(0), "Invalid Near prover address");

    nearTokenLocker = _nearTokenLocker;
    prover = _prover;
    minBlockAcceptanceHeight = _minBlockAcceptanceHeight;
}
```

- **Use Initializers for Upgradeable Contracts**: If the contract is intended to be upgradeable, use an initializer function instead of a constructor.

```
function initialize(
    bytes memory _nearTokenLocker,
    INearProver _prover,
    uint64 _minBlockAcceptanceHeight
) external initializer {
    require(_nearTokenLocker.length > 0, "Invalid Near Token Locker address");
    require(address(_prover) != address(0), "Invalid Near prover address");

    nearTokenLocker = _nearTokenLocker;
    prover = _prover;
    minBlockAcceptanceHeight = _minBlockAcceptanceHeight;
}
```

- **Add Setter Functions:** Provide functions to set these variables if they need to be changed after deployment, with appropriate access control.

```
function setProver(INearProver _prover) external onlyOwner {
    require(address(_prover) != address(0), "Invalid Near prover address");
    prover = _prover;
}

function setNearTokenLocker(bytes memory _nearTokenLocker) external onlyOwner {
    require(_nearTokenLocker.length > 0, "Invalid Near Token Locker address");
    nearTokenLocker = _nearTokenLocker;
}

function setMinBlockAcceptanceHeight(uint64 _minBlockAcceptanceHeight) external onlyOwner {
    minBlockAcceptanceHeight = _minBlockAcceptanceHeight;
}
```

Status: Unresolved

Finding: #9

Issue: Unused **Ownable** Library Import in **ProofConsumer** Contract

Severity: QA

Where:
- **File:** ProofConsumer.sol
- **Contract: ProofConsumer**
- **Import: @openzeppelin/contracts/access/Ownable.sol**

Impact: The **ProofConsumer** contract imports the **Ownable** library from OpenZeppelin but does not utilize it. This can lead to unnecessary bloat in the contract and potential confusion about the contract's intended functionality.

Description: The **ProofConsumer** contract includes an import statement for the **Ownable** library from OpenZeppelin. However, the contract does not extend **Ownable** or use any of its functionalities. This import is therefore redundant and may indicate that the **Ownable** functionality was either forgotten or not needed.

Recommendations:

1. **Evaluate Necessity**: Determine if the contract needs to include ownership control.
- If the contract should include ownership functionality, consider using **Ownable2Step** for a safer two-step ownership transfer process.
- If the ownership control is not required, remove the import statement to clean up the code.

2. **Implementing Ownership Control:**
- If ownership control is needed, update the contract to extend **Ownable2Step** and implement ownership-specific functions.

3. **Removing Unnecessary Import:**
- If ownership control is not needed, simply remove the import statement.

Status: Unresolved.

Finding: #10

Issue: Missing Parameter Validation in **initialize** Function of **BridgeTokenFactory.**

Severity: QA

Where:

- **File:** BridgeTokenFactory.sol
- **Function: initialize**

Impact: The **initialize** function in the **BridgeTokenFactory** contract lacks essential parameter validations for **_tokenImplementationAddress** and **_minBlockAcceptanceHeight**. Without these validations, the contract can be initialized with invalid parameters, leading to potential vulnerabilities or unintended behavior. Specifically:

- Setting **_tokenImplementationAddress** to **address(0)** can cause the contract to point to an invalid implementation.
- Allowing **_minBlockAcceptanceHeight** to be zero or an excessively high value can result in logical errors in the contract's operation.

Description: In the **initialize** function of the **BridgeTokenFactory** contract, the parameters **_tokenImplementationAddress** and **_minBlockAcceptanceHeight** are not validated. This oversight can lead to initialization with an invalid implementation address or an inappropriate block acceptance height.
The current implementation of the **initialize** function:

```
function initialize(
    address _tokenImplementationAddress,
    bytes memory _nearTokenLocker,
    INearProver _prover,
    uint64 _minBlockAcceptanceHeight
) external initializer {
    require(_nearTokenLocker.length > 0, "Invalid Near Token Locker address");
    require(address(_prover) != address(0), "Invalid Near prover address");

    nearTokenLocker = _nearTokenLocker;
    prover = _prover;
    minBlockAcceptanceHeight = _minBlockAcceptanceHeight;
    tokenImplementationAddress = _tokenImplementationAddress;

    __UUPSUpgradeable_init();
    __AccessControl_init();
    __Pausable_init_unchained();
    _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _grantRole(PAUSABLE_ADMIN_ROLE, _msgSender());
}
```

Recommendations:

- Validate **_tokenImplementationAddress**: Ensure that the **_tokenImplementationAddress** is not set to **address(0).**
- Validate **_minBlockAcceptanceHeight**: Ensure that **_minBlockAcceptanceHeight** is greater than zero and within a reasonable range.

```
function initialize(
    address _tokenImplementationAddress,
    bytes memory _nearTokenLocker,
    INearProver _prover,
    uint64 _minBlockAcceptanceHeight
) external initializer {
    require(_tokenImplementationAddress != address(0), "Invalid token implementation address");
    require(_nearTokenLocker.length > 0, "Invalid Near Token Locker address");
    require(address(_prover) != address(0), "Invalid Near prover address");
    require(_minBlockAcceptanceHeight > 0, "Block acceptance height must be greater than zero");
    require(_minBlockAcceptanceHeight <= type(uint64).max, "Block acceptance height exceeds uint64 max");

    nearTokenLocker = _nearTokenLocker;
    prover = _prover;
    minBlockAcceptanceHeight = _minBlockAcceptanceHeight;
    tokenImplementationAddress = _tokenImplementationAddress;

    __UUPSUpgradeable_init();
    __AccessControl_init();
    __Pausable_init_unchained();
    _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _grantRole(PAUSABLE_ADMIN_ROLE, _msgSender());
}
```

Status: Unresolved

Finding: #11

Issue: Unnecessary Gas Consumption in **_authorizeUpgrade** Function

Severity: QA

Where:

- **File:** BridgeToken.sol (and any other relevant Solidity files)
- **Function: _authorizeUpgrade**

Impact: The function **_authorizeUpgrade** currently has an empty block, which leads to unnecessary gas consumption without performing any useful operations. This results in inefficiency and minor additional costs for users interacting with the contract.

Description: In the Solidity contracts, the **_authorizeUpgrade** function is defined but contains an empty block. While this does not introduce any critical vulnerabilities, it leads to inefficient gas usage. Users or other contracts calling this function will incur gas costs for executing an empty block, which could be avoided.

Recommendations: Consider emitting an event within the **_authorizeUpgrade** function to provide meaningful output and make the gas consumption justified. For example:

```
function _authorizeUpgrade(address newImplementation) internal override onlyOwner {
    emit UpgradeAuthorized(newImplementation);
}
```

Add the following event definition:

```
event UpgradeAuthorized(address indexed newImplementation);
```

This change will ensure that the function does something meaningful (emitting an event) and provides transparency regarding upgrades while avoiding unnecessary gas consumption.

Status: Unresolved.

Finding: #12

Issue: Unutilized Memo Field in Fungible Token Transfer Methods

Severity: QA

Where: src/lib.rs#L306-L307

Impact: The memo field is intended to provide additional context or information for transfers, but since it is not being utilized, this functionality is effectively unused. This can lead to missed opportunities for capturing relevant metadata or implementing features that rely on the memo field.

Description: The provided code snippet includes two methods for transferring fungible tokens: ft_transfer and ft_transfer_call. Both methods take a memo parameter, which is described as optional and intended for use cases that may benefit from indexing or providing additional information for a transfer. However, upon further inspection, it appears that the memo field is not actually utilized within the implementation of these methods.

```rust
#[private]
#[payable]
pub fn finish_withdraw(
    &mut self,
    #[callback]
    #[serializer(borsh)]
    verification_success: bool,
    #[serializer(borsh)] token: AccountId,
    #[serializer(borsh)] recipient: Recipient,
    #[serializer(borsh)] amount: Balance,
    #[serializer(borsh)] proof_key: Vec<u8>,
) -> Promise {
    assert!(verification_success, "Failed to verify the proof");
    let required_deposit = self.record_proof(&proof_key);
    assert!(env::attached_deposit() >= required_deposit);

    let Recipient { target, message } = recipient;

    match message {
        Some(message) => ext_token::ext(token)
            .with_attached_deposit(near_sdk::ONE_YOCTO)
            .with_static_gas(FT_TRANSFER_CALL_GAS)
            .ft_transfer_call(target, amount.into(), None, message),
        None => ext_token::ext(token)
            .with_attached_deposit(near_sdk::ONE_YOCTO)
            .with_static_gas(FT_TRANSFER_GAS)
            .ft_transfer(target, amount.into(), None),
    }
}
```

Recommendations: Determine the specific use cases or scenarios where the memo field can provide value and incorporate the necessary logic to handle and process the memo information accordingly.

Status: Unresolved

Finding: #13

Issue: Solidity version **0.8.20+** may not work on other chains due to **PUSH0**

Severity: QA

Where: BridgeToken.sol#L2

Impact: The issue can lead to the contracts needing to be deployable on targeted chains.

Description: The compiler for Solidity **0.8.20** switches the default target EVM version to Shanghai, which includes the new **PUSH0** opcode. This opcode may not yet be implemented on all L2s, so deployment on these chains will fail. To work around this issue, use an earlier EVM version. While the project itself may or may not compile with **0.8.20**, other projects with which it integrates or extends this project may, and those projects will have problems deploying these contracts/libraries.

Recommendations: We recommend using the correct solidity version for each targeted chain.

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.