AUDITONE

Audit Report

AURORA

Forwarder and Controller Factory

10.05.2024

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, Kur0yuk1, Ubermensch3dot0 and Berndartmueller, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of AuroraF-C. 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 Fast Bridge, one-way semi-decentralized bridge created to speed up transfers from Near to Ethereum.

02-Project and Audit Information

Term	Description
Auditor	Defec, Kur0yuk1, Ubermensch3dot0 and Berndartmueller
Reviewed by	Luis Buendia and Gracious Igwe
Type	Forwarder and Controller
Language	Rust
Ecosystem	Near
Methods	Manual Review
Repository	https://github.com/aurora-is-near/aurora-forwarder-contractrs https://github.com/aurora-is-near/aurora-controller-factory
Commit hash (at audit start)	2f24862d6aa818279c795f9876415aacf435a669 f924927810d905ca5c7f7a6ec301ad85cdb27f4c
Commit hash (after resolution)	7554e3bdc2ddea87da43c77d2c82077e1ac723c9 f924927810d905ca5c7f7a6ec301ad85cdb27f4c
Documentation	[Added once the whitepaper is published by the project]
Unit Testing	NA
Website	https://aurora.dev/
Submission date	11.03.2024
Finishing date	10.05.2024

03-Contracts in Scope

Forwarder contracts path

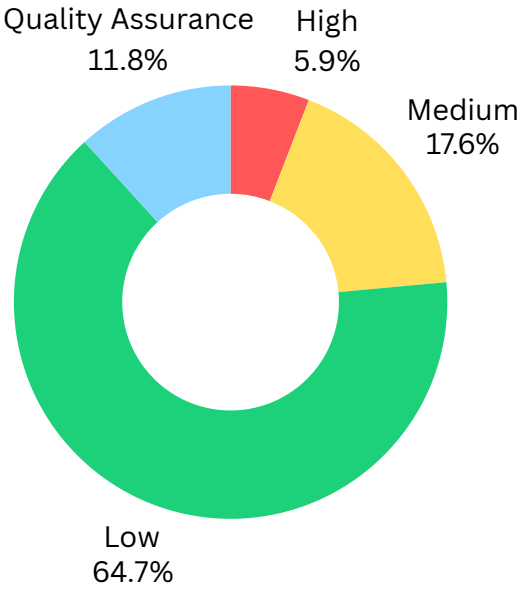
- forwarder/src/error.rs
- forwarder/src/lib.rs
- forwarder/src/params.rs
- forwarder/src/types/address.rs
- forwarder/src/types/promise.rs
- forwarder/src/types/mod.rs
- forwarder/src/types/account_id.rs
- forwarder/src/runtime/io.rs
- forwarder/src/runtime/handler.rs
- forwarder/src/runtime/env.rs
- 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

AuroraF-C smart contracts were audited between 05-01-2024 and 07-02-2024 by Defec, Kur0yuk1, Ubermensch3dot0 and Berndartmueller. 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	1	1	0
Medium	3	1	2
Low	11	2	9
Quality Assurance	2	0	2

05-Severity Definitions

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

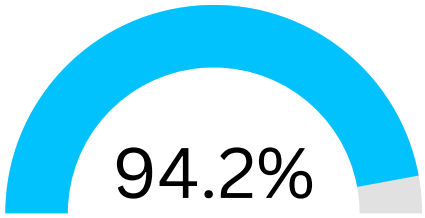
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

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.

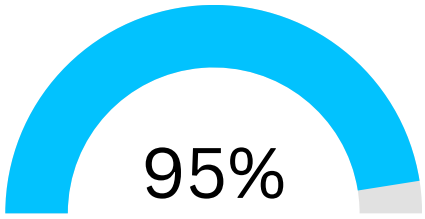
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



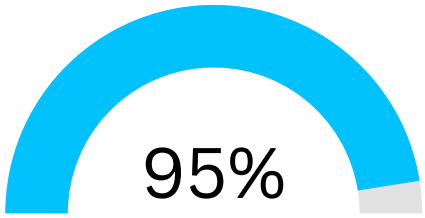
Security score

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: Anyone can add and remove supported fee tokens for the **FeesCalculator** contract, resulting in bypassing forwarder fees

Severity: High

Where: add_supported_token and remove_supported_token functions in fees/src/lib.rs#L104-L121

Impact: Forwarder fees can be bypassed, resulting in the protocol missing out on fees.

Description: The **add_supported_token** and **remove_supported_token** functions in the **FeesCalculator** contract do not have any kind of authorization check, anyone is able to invoke those functions and add/remove supported fee tokens.
Removing an already supported fee token would result in zero fees being charged for the forwarding attempt. Consequently, a user who is aware of this issue can avoid paying fees by first removing the fee token via the **remove_supported_token** function , followed by the actual token forwarding. As a result, zero fees are paid by the user.

Recommendations: In **add_supported_token** and **remove_supported_token**, consider checking if the caller is the contract owner, e.g.,:

```
assert_eq!(env::predecessor_account_id(), self.owner);
```

Status: Resolved.

Finding: #2

Issue: Lack of Comparison with Target Network and Token ID in Forwarding Logic

Severity: Medium

Where: forwarder/src/lib.rs#L113-L114

Description: The smart contract's forwarding logic does not include comparison checks for the target network and token ID when processing transactions. This oversight could lead to scenarios where transactions are forwarded without verifying whether the target network and the token ID are correct and intended for the transaction. Such a lack could potentially cause transactions to be executed in unintended networks or with incorrect tokens, leading to loss of funds or other operational issues.

```rust
#[no_mangle]
pub extern "C" fn calculate_fees_callback() {
    let mut io = Runtime;
    io.assert_private_call().sdk_unwrap();

    let params: ForwardParams = io.read_input_borsh().sdk_unwrap();
    let state = State::load(&io).sdk_expect("No state");
    let amount: u128 = match io.promise_result(0).sdk_expect("No promise result") {
        PromiseResult::Successful(v) => params::vec_to_number(&v).sdk_unwrap(),
        _ => panic_utf8(b"FEE RESULT IS NOT READY"),
    };

    let promise_id = unsafe {
        let promise_id = io.promise_create_and_combine(&[PromiseCreateArgs {
            target_account_id: state.fees_contract_id,
            method: "calculate_fees",
            args: types::to_borsh(&FeesParams {
                amount,
                token_id: &params.token_id,
                target_network: &state.target_network,
                target_address: state.target_address,
            })
            .sdk_unwrap(),
            attached_balance: ZERO_YOCTO,
            attached_gas: CALCULATE_FEES_GAS,
        }]);

        io.promise_attach_callback(
            promise_id,
            &PromiseCreateArgs {
                target_account_id: io.current_account_id(),
                method: "finish_forward_callback",
                args: types::to_borsh(&FinishForwardParams {
                    amount,
                    token_id: params.token_id,
                    promise_idx: 0,
                })
                .sdk_unwrap(),
                attached_balance: 2,
                attached_gas: FINISH_FORWARD_GAS,
            },
        )
    };
```

Recommendations: It is crucial to implement explicit checks within the **forward_native_token** and **forward_nep141_token** functions to ensure that the token ID and target network are as expected before proceeding with the transaction forwarding.

Status: Acknowledged

Finding: #3

Issue: Fee Bypass Due to Rounding in **calculate_fees** Method

Severity: Medium

Where: fees/src/lib.rs#L50-L68

Impact: The current implementation of the fee calculation allows for the possibility that fees can be entirely bypassed for small transaction amounts due to rounding. This occurs because the calculation method does not account for amounts that, when multiplied by the fee percentage and divided by 10000, result in a value less than 1. As a result, transactions of small amounts could potentially avoid incurring any fee, leading to revenue loss for the service and potentially enabling users to exploit this loophole by breaking up transactions into smaller amounts to avoid fees (depending on whether it's worth it from a gas perspective).

Description: The **calculate_fees** method calculates the fee based on a percentage of the transaction amount. However, due to the use of integer division, any calculation that results in a value between 0 and 1 will be rounded down to 0. This means that for small amounts of the transaction, the calculated fee can be zero, effectively bypassing the intended fee mechanism. This issue primarily affects transactions involving small quantities of tokens, where the fee, although nominally applicable, is nullified by the rounding behavior of integer arithmetic.

Recommendations: To address this issue, implement a minimum amount threshold such that any transaction below this threshold should not be permitted to be forwarded.

Status: Resolved.

Finding: #4

Issue: Unverified Receiver Storage Deposit

Severity: Medium

Where: forwarder/src/lib.rs#L164-L172

Impact: If the receiver (**state.target_network** or **state.fees_contract_id**) does not have a storage deposit for the specified **token_id**, the call to **finish_forward_callback** will fail. This failure can disrupt the intended token transfer process.

Description: The contract fails to verify whether the receiver specified by **state.target_network** has made a necessary storage deposit for the **token_id** in question. According to NEP-121 standards, it's crucial to ensure that a storage deposit exists to accommodate the tokens before attempting transfers. This omission can lead to scenarios where the **finish_forward_callback** call fails because the receiver's account does not have the required storage space allocated for the token, disrupting the intended functionality. The same issue goes for the **state.fees_contract_id.**

Recommendations: To address this issue, it is recommended to implement a preliminary check using the **storage_balance_of** method before executing the token transfer. This check should verify that the receiver (**state.target_network** and/or **state.fees_contract_id**) has a sufficient storage deposit for the **token_id** being transferred. If the check fails, the contract should call the **storage_deposit** (When NEAR balance is available) on behalf of the receiver or use the **sdk_expect** to inform the caller using a clear error message.

Status: Acknowledged.

Finding: #5

Issue: Lack of two-factor authentication

Severity: Low

Where: All Privileged functions - https://github.com/aurora-is-near/aurora-controller-factory/blob/master/src/lib.rs

Impact: 2FA will provide additional protection for the compromise of the account.

Description: Privileged functions should check whether one yocto NEAR is attached. This will enable the 2FA in the NEAR wallet for security concerns.
This can be implemented in the contract by adding **assert_one_yocto**, which is recommended for all privileged functions.

Recommendations: Consider using **assert_one_yocto** for 2FA authentication on the privileged functions.

Status: Acknowledged.

Finding: #6

Issue: Inflexible Fee Calculation Mechanism Across Different Tokens

Severity: Low

Where: lib.rs#L50

Impact: This one-size-fits-all approach to fee calculation may not be optimal for all tokens supported by the contract. Different tokens may have different volatility, liquidity, and transaction sizes, which could justify different fee percentages.

Description: The current implementation of the **calculate_fees** function within the **FeesCalculator** contract uses a single percentage to calculate fees for all token types. This approach lacks flexibility, as it applies the same fee percentage across different NEP-141 tokens, without considering the potential need for varying fee rates based on token characteristics or market conditions.

```
#[must_use]
 #[result_serializer(borsh)]
 pub fn calculate_fees(
     &self,
     #[serializer(borsh)] amount: U128,
     #[serializer(borsh)] token_id: &AccountId,
     #[serializer(borsh)] target_network: &AccountId,
     #[serializer(borsh)] target_address: Address,
 ) -> U128 {
     let _ = (target_network, target_address);

     if self.percent.is_none() || !self.supported_tokens.contains(token_id) {
         0.into()
     } else {
         u128::from(self.percent.unwrap().0)
             .checked_mul(amount.0)
             .unwrap_or_default()
             .saturating_div(10000)
             .into()
     }
 }
```

Recommendations: Modify the contract's state to include a mapping from token IDs to their respective fee percentages. This allows for the specification of unique fee percentages for each supported token.

Status: Acknowledged

Finding: #7

Issue: Hardcoded Gas Amount and Storage Deposit May Lead to Operation Failures

Severity: Low

Where: lib.rs#L11

Impact: If the storage deposit is insufficient, the contract may fail to allocate necessary storage, leading to failed contract initializations or data storage operations

Description: The AuroraForwarderFactory contract relies on hardcoded values for gas (**FORWARDER_NEW_GAS**) and storage deposit (**STORAGE_BALANCE_BOUND**) to manage the creation of forwarder contracts and storage deposits. This approach assumes that these values will always suffice for the operations they're intended for.

Recommendations: Consider adding setter function for these variables.

Status: Acknowledged

Finding: #8

Issue: Lacking min account Id check in **AccountId deserialize_reader** function

Severity: Low

Where: [aurora-forwarder-contractrs/forwarder/src/types/account_id.rs] (forwarder/src/types/account_id.rs#L67)

Impact: Lacking min length check might cause unexpected issue if the input length of **account id** is less than the minimum value when parsing **account Id**

Description: In **account_id.rs** file, the function **deserialize_reader** only has check for **len > MAX_ACCOUNT_ID_LEN**, while there is no check for **MIN_ACCOUNT_ID_LEN.**

Recommendations: Add one more check for **len < MIN_ACCOUNT_ID_LEN**

Status: Acknowledged

Finding: #9

Issue: Risk of Unauthorized Blob Override Due to Hash Collision in **add_release_blob**

Severity: Low

Where: In the **add_release_blob** function, specifically at the point where blobs are inserted into the contract without length limitation or access control.

Impact: The current implementation of the **add_release_blob** function introduces a significant security risk due to the potential for hash collisions. Without a limit on the byte length of the input blob and lacking sufficient access controls, there is a theoretical possibility for an attacker to brute force a hash collision. This would allow the submission of malicious or incorrect code that matches the hash of legitimate contract code. If successful, this could lead to unauthorized overrides of the code blobs stored in the contract, compromising the integrity of the contract deployment process and potentially enabling further attacks.

Description: The **add_release_blob** function accepts smart contract code (blob), calculates its SHA-256 hash, and stores it if the hash matches a pre-existing release hash in the contract. The absence of checks on the blob's length or content, combined with the lack of access controls, raises the possibility of an attacker generating a malicious blob whose hash collides with that of genuine code. Given the cryptographic strength of SHA-256, while the likelihood of finding such a collision is low, the absence of preventative measures significantly increases the risk profile, especially considering the high value and trust placed in smart contract systems.

Recommendations:

- **Implement Access Control:** Restrict the ability to submit blobs to trusted addresses only.
- **Verify Return Value of blobs.insert:** Modify the function to check the return value of **self.blobs.insert(&hash, &blob)**. This method usually returns a value indicating whether the insert operation resulted in an override of an existing value. If an override is detected, the function should revert the operation or flag an error, preventing unauthorized blob replacement.
- **Limit Blob Size**: Introduce limitations on the size of blobs that can be submitted. While this does not directly prevent hash collisions, it can reduce the feasibility of brute-forcing a collision by increasing the computational effort required to generate a matching hash for larger blobs.

Status: Acknowledged

Finding: #10

Issue: Redundant **is_fee_allowed** Check Due to Fee Calculation Constraints

Severity: Low

Where: [fees/src/lib.rs#L50-L68]

Impact: The **is_fee_allowed** function is designed to ensure that fees do not exceed a predefined maximum percentage (**MAX_FEE_PERCENT**). However, given the current fee calculation logic in **calculate_fees** that always rounds down and the fact that the fee percentage cannot exceed 10%, this check becomes redundant. The current logic ensures that the calculated fee will never exceed the maximum allowed percentage, making the **is_fee_allowed** check unnecessary. This redundancy could lead to unnecessary computational overhead and complexity in the codebase, although it does not directly impact the functionality or security of the contract.

Description: The **is_fee_allowed** function is intended as a safeguard to prevent fees from exceeding a certain percentage of the transaction amount. However, due to the way fees are calculated (with rounding down involved) and the constraint that the fee percentage cannot exceed 10%, the conditions checked by **is_fee_allowed** will never be met. The fee, as calculated, will always be within the allowed bounds, rendering the check superfluous. This situation highlights a misalignment between the purpose of the **is_fee_allowed** function and the practical implications of the fee calculation logic and its constraints.

Recommendations: While the redundancy of the **is_fee_allowed** function does not pose a risk to the contract's functionality or security, simplifying the codebase by removing unnecessary checks can improve readability and efficiency. Consider the following actions:

- Remove the **is_fee_allowed** Check: If analysis confirms that the fee calculation logic (considering the rounding down and the maximum fee percentage) inherently ensures compliance with the **MAX_FEE_PERCENT** constraint, then the **is_fee_allowed** function can be safely removed.
- Reevaluate Fee Constraints: If there is a future possibility of adjusting the fee calculation logic or the **MAX_FEE_PERCENT** constraint, retain the **is_fee_allowed** function but review its necessity and implementation as part of those changes.

Status: Resolved

Finding: #11

Issue: Unverified **wNEAR** Contract Account ID in Factory Contract.

Severity: Low

Where: factory/src/lib.rs#L69

Impact: The lack of verification for the wNEAR contract account ID introduces a risk. By accepting an unverified account ID from the **parameters** argument, the contract is susceptible to an incorrect wNEAR contract to be specified.

Description: The factory contract's approach to obtaining the wNEAR contract account ID directly from the **parameters** argument without any form of verification poses a risk. Given the importance of interacting with the correct wNEAR contract for operations involving NEAR wrapping and unwrapping, any error to provide a false account ID could result in a DoS over the created forwarder for native token forwarding.

Recommendations: This issue can be fixed by hardcoding the wNEAR contract account ID and verifying **params.wnear_contract_id** using it.

Status: Acknowledged

Finding: #12

Issue: Inefficient Promise Handling with **promise_create_and_combine**.

Severity: Low

Where: forwarder/src/lib.rs#L107-L119

Impact: Utilizing **promise_create_and_combine** with only one promise instead of the intended two or more introduces unnecessary complexity and overhead. This function is designed to wait for multiple promises to complete and combines their results. When used with a single promise, it does not offer any benefit over **promise_create_call**, and its use in such a context can lead to confusion about the intent of the code and potentially degrade the performance due to the extra overhead associated with handling promise combinations.

Description: The smart contract makes use of **promise_create_and_combine** in a scenario where only a single promise is passed as an argument. This approach diverges from the intended use of **promise_create_and_combine**, which is to handle situations where there are multiple promises whose results need to be combined. The NEAR platform provides **promise_create_call** for scenarios where a single promise is created, which is more appropriate for the use case observed in the contract. The use of **promise_create_and_combine** in this context does not align with best practices for efficient and clear promise handling.

Recommendations: Replace the use of **promise_create_and_combine** with **promise_create_call** in instances where only a single promise is being handled.

Status: Resolved

Finding: #13

Issue: Mismatch Between Code Documentation and Implementation Regarding **parameters** Length Check

Severity: Low

Where: factory/src/lib.rs#L43-L55
factory/src/lib.rs#L16

Impact: The discrepancy between the code documentation and the actual implemented logic could lead to confusion for developers working on or with the smart contract. While the code allows up to 12 elements in the **parameters** list (as indicated by the MAX_NUM_CONTRACTS constant being 12), the documentation incorrectly states that no more than 10 elements are accepted.

Description: The **create** function in the factory contract includes a precondition check that asserts the length of the **parameters** list does not exceed MAX_NUM_CONTRACTS. However, there is a discrepancy between the documentation (comments) and the code. The comments state that the function does not accept a list of **parameters** with more than 10 elements, whereas the code compares the length of **parameters** against MAX_NUM_CONTRACTS, which is actually set to 12. This inconsistency could cause confusion and mislead developers regarding the contract's capabilities and limitations.

Recommendations: To resolve this issue, the documentation (comments) should be updated to accurately reflect the implemented logic. If the intended behavior is to accept up to 12 elements in the **parameters** list, the comment should be corrected to match this logic, stating clearly that the maximum number of elements allowed is 12, not 10. Conversely, if the original intention was indeed to limit the list to 10 elements, then the code should be adjusted accordingly by setting MAX_NUM_CONTRACTS to 10. Ensuring consistency between the documentation and the code will improve clarity and developer experience.

Status: Acknowledged

Finding: #14

Issue: Misuse of Standard **expect** Over **sdk_expect** for Error Handling

Severity: Low

Where: forwarder/src/lib.rs#L193-L197

Impact: The use of Rust's standard **expect** method for error handling in a smart contract context does not leverage the NEAR platform's ability to properly communicate errors. The **expect** method from the standard library merely panics with a generic error message, which may not be efficiently captured or communicated back to the caller in a blockchain context. This could lead to less informative error messages and potentially hinder debugging or interaction with the smart contract, affecting developer experience and user interaction.

Description: In the **factory** and **forwarder** contracts, the standard library's **expect** method is used for error handling in several places. However, for smart contracts deployed on the NEAR platform, it's recommended to use **sdk_expect**, which utilizes the **panic_utf8** method from the **MiscellaneousAPI**. The **panic_utf8** method ensures that execution terminates with a **GuestPanic**, where the panic message is a given UTF-8 encoded string. This method of error handling is more suitable for smart contracts on NEAR, as it ensures that errors are communicated clearly and effectively, improving the debugging process and user experience.

Recommendations: Replace the instances of the standard library's **expect** method with the NEAR SDK's **sdk_expect** method for error handling. This change will ensure that errors are handled in a way that leverages the NEAR platform's capabilities, leading to more informative error messages and a better overall experience for developers and users interacting with the smart contract.

Status: Resolved

Finding: #15

Issue: Removing the release info and blob for the latest release results in an error when retrieving the latest release blob via **get_latest_release_blob.**

Severity: Low

Where: src/lib.rs#L235-L241

Impact: Retrieving the latest release blob via the **get_latest_release_blob** function errors as the corresponding blob got deleted.

Description: The **remove_release** function removes the release info and the blob associated with the specified **hash**.
However, if the **hash** (i.e., the release) corresponds to the latest release, removing the release info and the blob will cause storage discrepancies. Specifically, the latest release is still stored, but the corresponding blob was deleted. Retrieving this blob, e.g., when calling the **get_latest_release_blob** function, results in a panic ("the latest release hash hasn't been set yet") in line 261 as the blob does not exist anymore in storage.
While this does not impose any security issues, this behavior is misleading as the latest release is still stored without being able to retrieve its blob.

Recommendations: Either delete the latest release when calling **remove_release**, or, prevent removing the release when calling **remove_release** with associated withated to the latest release.

Status: Acknowledged

Finding: #16

Issue: Custom Access Control Implementation in Smart Contract

Severity: Quality Assurance

Where: fees/src/lib.rs#L77

Impact: Custom implementations may inadvertently introduce security vulnerabilities that are less likely in standardized solutions.

Description: The smart contract contains a custom implementation for access control, including fee setter (owner) functionality owner access rights. While custom implementations provide flexibility, they can introduce complexity and potential security risks if not carefully designed and tested.
Using established Aurora Near plugins, can offer more security and maintainability.

```rust
/// Set the percent of the fee.
///
/// # Panics
///
/// Panics if the invoker of the transaction is not owner.
#[allow(clippy::needless_pass_by_value)]
pub fn set_fee_percent(&mut self, percent: Option<String>) {
    assert_eq!(env::predecessor_account_id(), self.owner);

    match parse_percent(percent.as_deref()) {
        Ok(value) => self.percent = value,
        Err(e) => env::panic_str(&format!("Couldn't parse percent: {e}")),
    }
}
```

Recommendations: Replace the custom access control logic with Aurora near plugins.

Status: Acknowledged

Finding: #17

Issue: Potential Hash Collision in Forwarder Prefix Generation

Severity: Quality Assurance

Where: factory/src/lib.rs#L60-L64
factory/src/lib.rs#L130-L139
utils/src/lib.rs#L5-L20

Impact: While the likelihood of a hash collision is low given the current implementation, the potential exists for two different sets of input parameters (**address**, **target_network**, and **fees_contract_id**) to produce the same hash result. This could lead to a scenario where two forwarders are inadvertently assigned the same prefix, which will cause the forwarder to be impossible to deploy. Although the probability of such a collision is minimal due to the cryptographic strength of the hash function used, the consequences could be significant, warranting precautionary measures.

Description: The **forwarder_prefix** function constructs a byte array from the concatenated **address**, **target_network**, and **fees_contract_id**, and then applies the **keccak256** hash function to this array, encoding the result in **base58**. The approach assumes that the hash function will provide a unique output for each unique input combination. However, in the rare event of a hash collision, where two different inputs produce the same output hash, this could cause the forwarder to be impossible to deploy.

Recommendations: To mitigate the risk of hash collisions, consider incorporating an additional unique element, such as a salt, into the byte array before hashing. The salt could be a timestamp, a nonce that increments with each forwarder creation, or another unique identifier that ensures each input to the hash function is distinct.

Status: Acknowledged

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.