AUDITONE

Audit Report

AURORA

09.05.2023

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

Csanuragjain, Defsec, Cryptoninja and Ubermensch, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Aurorafastbridge. 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 plugins, which are implementations of common patterns used for NEAR smart contracts.

02-Project and Audit Information

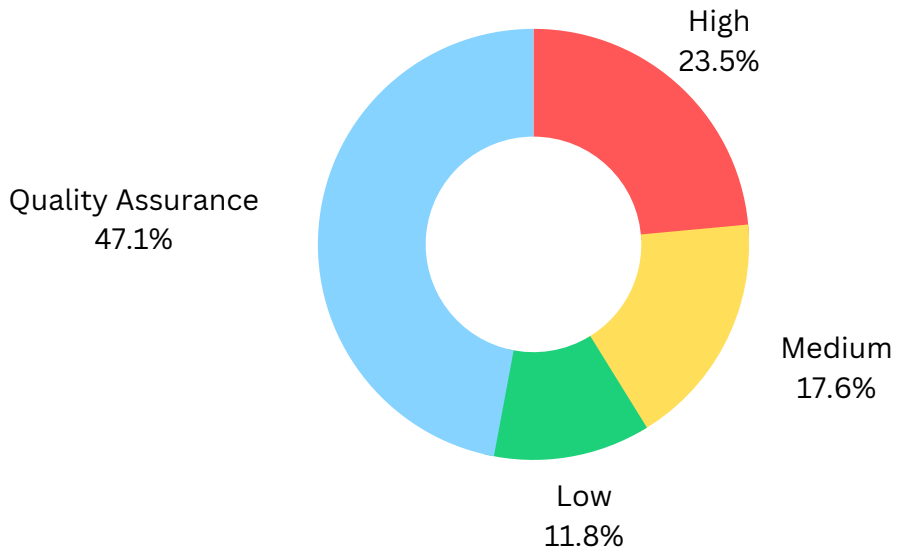
Term	Description
Auditor	Csanuragjain, Defsec, Cryptoninja and Ubermensch
Reviewed by	Gracious
Type	Bridge
Language	Rust
Ecosystem	NEAR and Ethereum networks
Methods	Manual Review
Repository	https://github.com/aurora-is-near/fast-bridge-service/
Commit hash (at audit start)	bcc6b12c84f6131e357cd498f25e0f463a6fcae7
Commit hash (after resolution)	ff3c98ba1d212bc64c4c14bf8841057f9f1231e9
Documentation	[Added once the whitepaper is published by the project]
Unit Testing	No
Website	https://aurora.dev/
Submission Time	01-03-2023
Finishing Time	09-05-2023

03-Contracts in Scope

- *https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/transfer.rs*
- *https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/lib.rs*
- *https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/async_redis_wrapper.rs*
- *https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/event_processor.rs*
- *https://github.com/aurora-is-near/fast-bridge-service/blob/5ed18302ae675e849e0f34b955f0646028946406/src/event_processor.rs*
- *https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/whitelist.rs*
- *fast-bridge-protocol\near\contracts\bridge\src\lib.rs*
- *https://github.com/aurora-is-near/fast-bridge-service/blob/5ed18302ae675e849e0f34b955f0646028946406/src/near_events_tracker.rs*
- *https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/unlock_tokens.rs*
  https://github.com/aurora-is-near/fast-bridge-service/blob/master/Cargo.toml

04-Executive summary

Aurorafastbridge plugin's smart contracts were audited between 2022-12-06 and 2023-03-14 by Csanuragjain, Defsec, Cryptoninja and Ubermensch. 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	4	4	0
Medium	3	3	0
Low	2	2	0
Quality Assurance	8	8	0

05-Severity Definitions

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

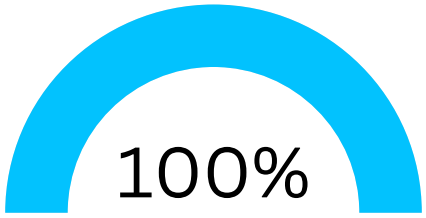
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

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.

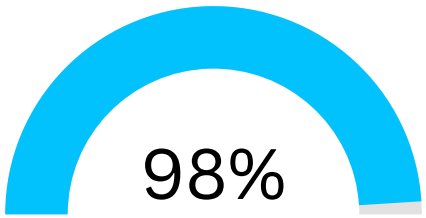
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



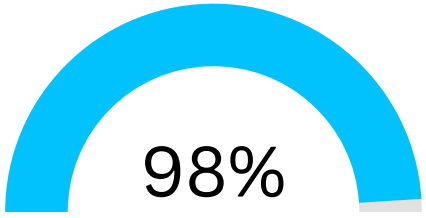
Security score

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: Block Reorg Can Allow For Double Spending

Severity: High

Where: Business logic

Impact: Block reorgs can harm the protocol's logic by invalidating previously confirmed transactions. For example, if a transaction is included in a block that is later reorganized, the transaction may no longer be valid. This can lead to double-spending or other issues if the transaction was used to transfer tokens or execute a smart contract function.

In the case of the Fast Bridge project, block reorgs can cause problems if they occur during the transfer of tokens between Ethereum and NEAR. Specifically, if a block reorg occurs after the tokens have been locked on one chain but before they are unlocked on the other chain, the tokens may be lost or double-spent. This can happen because the LP-Relayer may have already released the tokens on the other chain, assuming that the transaction was confirmed, but the reorg means that the transaction is no longer valid.

Description: Block reorg, also known as blockchain reorganization, is a situation where a competing chain replaces the main blockchain. This can happen when multiple miners find valid blocks at the same time, and the network has to decide which block to include in the blockchain. In some cases, the network may choose to include a block that is not in the main blockchain, resulting in a reorganization of the chain.

Recommendations:
To mitigate the risk of block reorgs, the Fast Bridge project may need to implement additional measures, such as waiting for multiple confirmations before proceeding with token transfers or implementing a fallback mechanism in case of a block reorg.

Status: Resolved

Finding: #2

Issue: Potential for Race Condition between Unlock Time and Proof Verification leading to Double Spending

Severity: High

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/transfer.rs

Impact: If there is a race condition, it means that the **LP-Relayer** may release the tokens before the proof of transaction has been fully verified. This can happen if the **LP-Relayer** is attempting to double spend the tokens, or if they are hacked or compromised and someone else is trying to steal the tokens.
If the **LP-Relayer** releases the tokens before the proof of transaction has been fully verified, then the sender's tokens will be unlocked on the NEAR side, and they will be able to spend them again. This means that there is a potential for double spending, as the **LP-Relayer** can then use the same tokens to make another transaction.

Description: In the Fast Bridge project, a race condition between unlock time and proof verification can lead to double spending. Here's how it can happen:
When a user sends tokens from Ethereum to NEAR, the tokens are locked on the Ethereum side, and the **LP-Relayer** is responsible for releasing the tokens on the NEAR side once the proof of transaction has been received. The **LP-Relayer** has a specific time window in which to release the tokens, which is set by the sender when they initiate the transaction.

Recommendations:
To prevent this from happening, it's important that the **LP-Relayer** is trusted and secure, and that there are appropriate measures in place to verify that the proof of transaction is valid before the tokens are released. This can include using secure verification processes, and having multiple parties involved in verifying the proof before the tokens are released.

Status: Resolved

Finding: #3

Issue: User can drain all funds by calling withdraw multiple times

Severity: High

Where: https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/lib.rs#L628-L668

Impact: Attacker can withdraw more than his actual balance

Description: Currently **withdraw** function transfer tokens and after its success, the **callback** function decreases user's balance. As main call and callback handler are independent transaction, a malicious user can call withdraw repeatedly, before callback function called.
As user's balance is still not decreased, user can still transfer tokens.

Recommendations:

- We should follow CEI pattern in this case.
- We should move **decrease_balance** to **withdraw**, above **ft_transfer**.

Status: Resolved

Finding: #4

Issue: Malicious user can double-unlock his locked funds

Severity: High

Where: https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/lib.rs#L352-L396

Impact: Loss of funds. Wrong deductions of **pending_transfers_balances.**

Description: Currently **unlock** function checks whether pending transfer of the nonce does exist or not, after its success, the callback function increases balance and remove transfer.
In Near, main call and callback handler are independent transaction, so a malicious user can call unlock repeatedly, before callback function finished.
As pending transfer is not removed yet, user can still pass the check in second call.
Second call of **remove_transfer** in callback will not panic but only return None.
As a result, user can double-unlock his funds.

```rust
#[pause(except(roles(Role::UnrestrictedUnlock)))]
pub fn unlock(&self, nonce: U128, proof: near_sdk::json_types::Base64VecU8) -> Promise {
    let proof = UnlockProof::try_from_slice(&proof.0)
        .unwrap_or_else(|_| env::panic_str("Invalid borsh format of the `UnlockProof`"));

    let (recipient_id, transfer_data) = self
        .get_pending_transfer(nonce.0.to_string())
        .unwrap_or_else(|| near_sdk::env::panic_str("Transfer not found"));

    let storage_key_hash = utils::get_eth_storage_key_hash(
        transfer_data.transfer.token_eth,
        transfer_data.recipient,
        eth_types::U256(nonce.0.into()),
        eth_types::U256(transfer_data.transfer.amount.0.into()),
    );

    let expected_storage_value = vec![];

    ext_prover::ext(self.prover_account.clone())
        .with_static_gas(utils::tera_gas(50))
        .with_attached_deposit(utils::NO_DEPOSIT)
        .verify_storage_proof(
            proof.header_data,
            proof.account_proof,
            self.eth_bridge_contract.to_vec(),
            proof.account_data,
            storage_key_hash,
            proof.storage_proof,
            proof.storage_proof,
            expected_storage_value,
            transfer_data.valid_till_block_height,
            None,
            false,
        )
        .then(
            ext_self::ext(current_account_id())
                .with_static_gas(utils::tera_gas(50))
                .with_attached_deposit(utils::NO_DEPOSIT)
                .unlock_callback(
                    nonce,
                    env::predecessor_account_id(),
                    transfer_data,
                    recipient_id,
                ),
        )
}
```

Recommendations:
We should check if the nonce still exists in the **pending_transfers** at **unlock_callback.**

Status: Resolved

Finding: #5

Issue: Subscribed message may get lost

Severity: Medium

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/async_redis_wrapper.rs#L148-L150

Impact: If an error occurs while sending the received event message, then this event would get lost and would not be processed

Description:

- The **subscribe** function will stop if there is any error while sending the **pubsub_msg**.

```
while let Some(s) = pubsub_stream.next().await {
            let pubsub_msg: T = s.get_payload().expect("Failed to fetch the message");
            if let Err(_e) = sender.send(pubsub_msg).await {
                break;
            }
        }
    });
    Ok(receiver)
```

- Restarting the **subscribe** function may take time
- Within that time all published events would get lost and wont be processed.

Recommendations:
Probably an offchain component can keep track of all lost event messages and Admin could send those lost event messages so that they could be processed

Status: Resolved

Finding: #6

Issue: Redis db connection issue - Relayer fund loss

Severity: Medium

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/event_processor.rs#L71

Impact: If transaction was executed on ethereum meaning token were transferred but due to connection issue redis db was not updated then user can unlock near token as well

Description:

- User A use fast bridge to get token on ETH
- Relayer executes the transaction on ETH but due to redis issue, the **PENDING_TRANSACTIONS** entry could not be made in Redis
- This causes Relayer to be unaware about this issue and Relayer now wont issue **lp_unlock** on near side
- After bridge request expire user can unlock the token. This means user gets both token on eth and near side

Recommendations: Revert if redis connection issue is present

Status: Resolved.

Finding: #7

Issue: Lack of Validation for **valid_till_block_height** on FastBridge Service

Severity: Medium

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/5ed18302ae675e849e0f34b955f0646028946406/src/event_processor.rs

Impact: With the correct validation in place, FastBridge Service can ensure that transactions are executed within the specified **valid_till_block_height**, reducing delays and enhancing efficiency.

Description: The FastBridge Service, which is responsible for managing transfers between the NEAR and Ethereum networks, does not validate the **valid_till_block_height** parameter. This parameter is used to set an expiration block height for the transfer, and if not validated properly, it could result in transfers being processed after they have expired.

```
let _res = process_transfer_event(
    nonce,
    near_account.parse().unwrap(),
    TransferMessage {
        valid_till,
        transfer,
        fee,
        recipient,
        valid_till_block_height: None,
        aurora_sender: None
    },
    &settings.lock().await.clone(),
    &mut redis,
    get_eth_erc20_fast_bridge_proxy_contract_address(),
    relay_eth_key.clone(),
    eth_erc20_fast_bridge_contract_abi.clone(),
    near_account,
)
```

Recommendations: To address this issue, it is recommended that the FastBridge Service implement proper validation for the **valid_till_block_height** parameter. This could include checks to ensure that the current block height is not greater than the **valid_till_block_height** value.

Status: Resolved.

Finding: #8

Issue: Missing comparison between **lock_time_min** and **lock_time_max** in **set_lock_time** function

Severity: Low

Where: https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/lib.rs

Impact: Without checking the relationship between **lock_time_min** and **lock_time_max**, the code might allow an invalid configuration state that could lead to unexpected behavior or runtime errors in other parts of the application that rely on the lock duration.

Description: In the **set_lock_time** function, the **lock_time_min** and **lock_time_max** values are not compared before setting the **lock_duration**. This might lead to a situation where **lock_time_min** is greater than **lock_time_max**, which could be an invalid state for the intended logic of the application.

```rust
#[access_control_any(roles(Role::ConfigManager))]
pub fn set_lock_time(&mut self, lock_time_min: String, lock_time_max: String) {
    let lock_time_min: u64 = parse(lock_time_min.as_str())
        .unwrap()
        .as_nanos()
        .try_into()
        .unwrap();
    let lock_time_max: u64 = parse(lock_time_max.as_str())
        .unwrap()
        .as_nanos()
        .try_into()
        .unwrap();

    self.lock_duration = LockDuration {
        lock_time_min,
        lock_time_max,
    };
}
```

Recommendation: To prevent potential issues with invalid lock duration configurations, we recommend adding a comparison between **lock_time_min** and **lock_time_max** before updating the **lock_duration**. If **lock_time_min** is greater than **lock_time_max**, the function should return an error or panic to indicate that the provided values are invalid. Here's an example of how you can implement this check:

```rust
#[access_control_any(roles(Role::ConfigManager))]
pub fn set_lock_time(&mut self, lock_time_min: String, lock_time_max: String) {
    let lock_time_min: u64 = parse(lock_time_min.as_str())
        .unwrap()
        .as_nanos()
        .try_into()
        .unwrap();
    let lock_time_max: u64 = parse(lock_time_max.as_str())
        .unwrap()
        .as_nanos()
        .try_into()
        .unwrap();

    // Ensure lock_time_min is less than or equal to lock_time_max
    assert!(
        lock_time_min <= lock_time_max,
        "lock_time_min must be less than or equal to lock_time_max"
    );

    self.lock_duration = LockDuration {
        lock_time_min,
        lock_time_max,
    };
}
```

Finding: #9

Issue: Lack of Check for Same **token_eth** and **recipient** in the NEAR Contract

Severity: Low

Where: https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/lib.rs

Impact: If the **token_eth** and **recipient** fields are the same, the contract might execute transfers that are not intended or logically incorrect, leading to unexpected outcomes and inaccurate records.

Description: In the provided code snippet, a JSON object is created, containing the details for a transfer, including the **token_eth** and **recipient** fields. However, there is no check to ensure that the **token_eth** and **recipient** fields are not the same. Allowing the same value for both fields could lead to potential issues in the contract's execution, as it might not be the intended behavior for a valid transfer.

```
let current_timestamp = block_timestamp() + contract.lock_duration.lock_time_min + 20;
let token = "alice_near";
let msg = json!({
    "valid_till": current_timestamp,
    "transfer": {
        "token_near": token,
        "token_eth": "71c7656ec7ab88b098defb751b7401b5f6d8976f",
        "amount": "100"
    },
    "fee": {
        "token": token,
        "amount": "100"
    },
    "recipient": "71c7656ec7ab88b098defb751b7401b5f6d8976f"
});
```

Recommendations: To address this issue, it is recommended that the NEAR contract includes a check to ensure that the **token_eth** and **recipient** fields are not the same.

Status: Resolved.

Finding: #10

Issue: Missing comments on Code

Severity: Quality Assurance

Where: Almost all contracts

Impact: Lack of comments make it harder to understand what a specific function is doing.

Description: Most of the contract functions are missing comments.

Recommendations: Recommended to add comments on all contract functions.

Status: Resolved.

Finding: #11

Issue: Unexpected token unlock could happen

Severity: Quality Assurance

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/transfer.rs#L115-L124

Impact: Token will get unlocked before User expectation causing transfer to not work. User will need to again start the transfer using only **valid_till** argument, wasting gas and time of user.

Description:

- As per docs, max of **valid_till** or **valid_till_block_height** is always taken to derive the unlock time:
  "**valid_till_block_height:** Option<u64> — the same as **valid_till**, but in block height, not in nanoseconds. If both values are provided, tokens will be locked on the max of the two values. (In that stage for User only None value makes sense)".
- But seems like **valid_till_block_height** is not actually used while checking valid time validity
- As we can see in below function unlock time validation is only performed on **valid_till** parameter and **valid_till_block_height** param is not used.

```
let transaction_unlock_time_ns = transfer_message.valid_till as u128;
        ...

        if current_time_ns + min_time_before_unlock_ns > transaction_unlock_time_ns {
            return Err(CustomError::NotEnoughTimeBeforeUnlock);
        }
```

Recommendations: **valid_till_block_height** should also be used to derive the unlock time.

Status: Resolved.

Finding: #12

Issue: Improve handling of **CheckToken** case in **check_whitelist_token_and_account** function

Severity: Quality Assurance

Where: https://github.com/aurora-is-near/fast-bridge-protocol/blob/master/near/contracts/bridge/src/whitelist.rs

Impact: The current implementation with an empty block might lead to confusion for developers who are reading or maintaining the code. They might not understand the purpose of the **CheckToken** variant and why there is no action associated with it. This could potentially lead to future bugs or unnecessary code modifications.

Description: In the **check_whitelist_token_and_account** function, the **CheckToken** variant of the **WhitelistMode** enum has an empty block, which might be unclear to readers of the code. This variant is intended to indicate that only the token needs to be checked against the whitelist, and no action is required for the associated account. However, the current implementation with an empty block might not effectively communicate this intent.

```rust
pub fn check_whitelist_token_and_account(&self, token: &AccountId, account: &AccountId) {
    if !self.is_whitelist_mode_enabled {
        return;
    }

    let token_whitelist_mode = self.whitelist_tokens.get(token).unwrap_or_else(|| {
        env::panic_str(format!("The token `{}` is not whitelisted", token).as_str())
    });

    match token_whitelist_mode {
        WhitelistMode::CheckAccountAndToken => {
            let token_account_key = get_token_account_key(Some(token), account);
            require!(
                self.whitelist_accounts.contains(&token_account_key)
                    || self.whitelist_accounts.contains(&account.to_string()),
                format!(
                    "The token `{}` isn't whitelisted for the account `{}`",
                    token, account
                ),
            );
        }
        WhitelistMode::CheckToken => {}
        WhitelistMode::Blocked => {
            env::panic_str(format!("The token `{}` is blocked", token).as_str())
        }
    }
}
```

Recommendations: To improve the clarity and maintainability of the code, we recommend using the **_ => {}** pattern in the match statement for the **CheckToken** variant. This pattern makes it explicit that no action is needed for this case. Alternatively, you can add a comment within the empty block to explain why no action is required. Here's the updated match statement:

```rust
match token_whitelist_mode {
    WhitelistMode::CheckAccountAndToken => {
        // ...
    }
    WhitelistMode::Blocked => {
        // ...
    }
    _ => {} // No action needed for CheckToken, as the token is already checked in whitelist_tokens
}
```

Status: Resolved.

Finding: #13

Issue: Unnecessary Initialization verification

Severity: Quality Assurance

Where:
fast-bridge-protocol\near\contracts\bridge\src\lib.rs

Impact: This redundancy in the code can lead to gas wastage and possibly cause confusion for future developers who may not understand the reason for the additional check.

Description: The **require!** statement in the new function is used to check if the contract has already been initialized before. However, the function also uses the init macro which is responsible for initializing the contract. Since the init macro can only be called once per contract, there is no need for an additional check using **require!** to verify whether the contract has been initialized before.

```
#[near_bindgen]
impl FastBridge {
    #[init]
    #[private]
    pub fn new(
        eth_bridge_contract: String,
        prover_account: AccountId,
        eth_client_account: AccountId,
        lock_time_min: String,
        lock_time_max: String,
        eth_block_time: Duration,
    ) -> Self {
        require!(!env::state_exists(), "Already initialized");
```

Recommendations: It is recommended to remove the unnecessary **require!** statement to ensure a cleaner and more efficient codebase.

Status: Resolved.

Finding: #14

Issue: Lack of Configurability for Timeouts in Fast Bridge Service

Severity: Quality Assurance

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/5ed18302ae675e849e0f34b955f0646028946406/src/event_processor.rs
https://github.com/aurora-is-near/fast-bridge-service/blob/5ed18302ae675e849e0f34b955f0646028946406/src/near_events_tracker.rs

Impact: The impact of this issue could result in frustration and inconvenience for users, who may have to retry transactions or experience delays in the confirmation process. Additionally, the lack of configurability may make it difficult for the Fast Bridge project to adapt to changing transaction conditions or network congestion, which could lead to reduced adoption and usage of the system.

Description: The Fast Bridge project has a lack of configurability for timeouts related to transactions. In the code snippet provided, there is a hardcoded timeout of 60 seconds, which may not be sufficient for all transactions. This lack of configurability could result in delays or even failures of transactions, especially in cases where longer confirmation times are required.

```
let timeout_duration = std::time::Duration::from_secs(60);
let _result = timeout(timeout_duration, worker).await;
```

Recommendations: To address this issue, it is recommended that the Fast Bridge project implement a more configurable approach for timeouts related to transactions.

Status: Resolved.

Finding: #15

Issue: Wrong parameter on **lp_unlock** near call at **unlock_tokens** in Relayer

Severity: Quality Assurance

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/src/unlock_tokens.rs#L24-L27

Impact: Bad Code readability

Description: From Relayer, it passes **{ nonce, proof }** as param to **lp_unlock** call.

```
async fn unlock_tokens(
    server_addr: url::Url,
    account: near_crypto::InMemorySigner,
    contract_account_id: String,
    proof: fast_bridge_common::Proof,
    nonce: u128,
    gas: u64,
) -> Result<(FinalExecutionStatus, CryptoHash), CustomError> {
    tracing::info!("Start lp unlock for token with nonce={}", nonce);
    let result = near_client::methods::change(
        server_addr,
        account,
        contract_account_id,
        "lp_unlock".to_string(),
        near_sdk::serde_json::json!({
            "nonce": near_sdk::json_types::U128(nonce),
            "proof": proof,
        }),
        gas,
        0,
    )
    .await
    .map_err(|err| CustomError::FailedExecuteUnlockTokens(err.to_string()))?;
```

However, actual near implementation has just one param proof.

```
#[pause(except(roles(Role::UnrestrictedLpUnlock)))]
pub fn lp_unlock(&mut self, proof: Proof) -> Promise {
    let parsed_proof = lp_relayer::EthTransferEvent::parse(proof.clone());
    assert_eq!(
        parsed_proof.eth_bridge_contract,
        self.eth_bridge_contract,
        "Event's address {} does not match the eth bridge address {}",
        hex::encode(parsed_proof.eth_bridge_contract),
        hex::encode(self.eth_bridge_contract),
    );
```

Recommendations: As **proof** includes **nonce** and everything looks good at Near side, we should keep consistency and remove **nonce** from **lp_unlock** call from Relayer.

Status: Resolved.

Finding: #16

Issue: Limitations and Risks for Users in the Fast Bridge Project

Severity: Quality Assurance

Where: Documentation

Impact: These limitations and risks can impact the user experience and adoption of the Fast Bridge project. Users may be discouraged from using the bridge due to the higher price and limitations on the tokens that can be transferred. Additionally, the need to contact support to unlock stuck tokens can cause inconvenience and delays.

Description: The Fast Bridge project allows for the transfer of tokens between the NEAR Protocol and Ethereum networks. However, there are several limitations and risks for users. For example, if there is no relayer available to handle the transaction, the tokens will be locked until the valid_till time, and users will need to contact support to unlock stuck tokens on NEAR. The transaction size is limited by relayer liquidity, and only tokens from the whitelist can be transferred. Additionally, the price is significantly higher than the original bridge.
On the other hand, The Fast Bridge project relies on LP-Relayers to hold locked tokens on the Ethereum side and release them on the NEAR side. However, there are several risks and limitations for relayers. For example, there is a risk of double unlock if the relayer already transferred tokens on the Ethereum side and switches off for a while. The relayer is an off-chain part that needs extra maintenance, including increasing liquidity and key management protection. The relayer can also have issues with the internet or server uptime.

Recommendation: To mitigate these risks, it is recommended that the Fast Bridge project implement a more decentralized approach that reduces reliance on the relayer and provides greater flexibility and scalability for users.

Status: Resolved.

Finding: #17

Issue: Out-of-date Rust Crate Detected with Cargo Audit

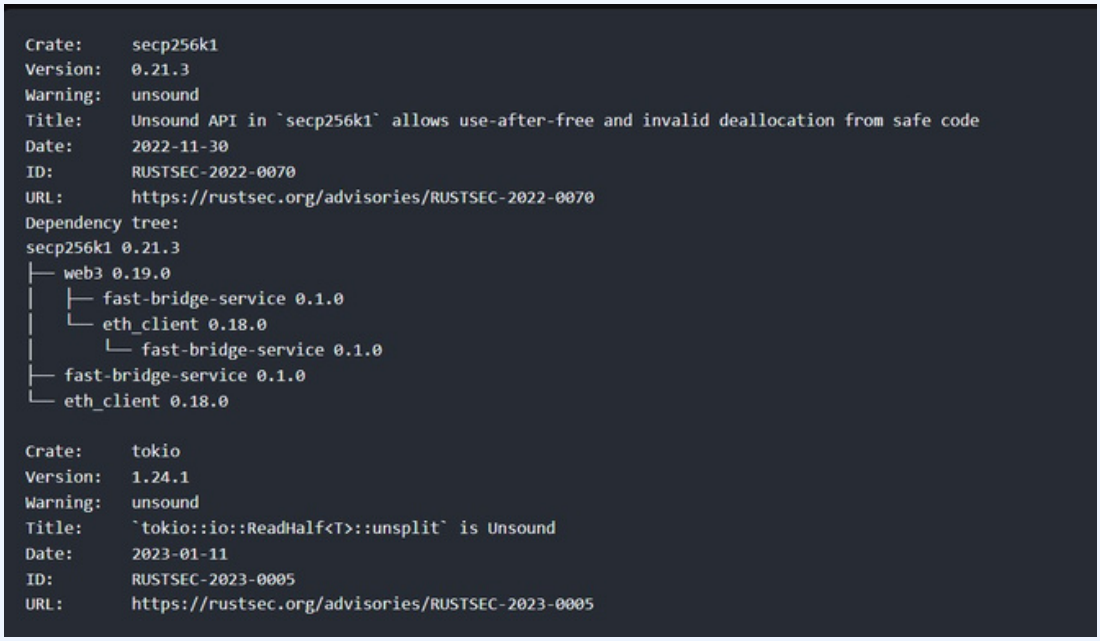
Severity: Quality Assurance

Where: https://github.com/aurora-is-near/fast-bridge-service/blob/master/Cargo.toml

Impact:

- **Security Vulnerabilities:** Outdated crates can contain known security vulnerabilities that have been fixed in the newer versions. Using these outdated dependencies can expose the project to potential attacks, compromising the application's security and integrity.
- **Missing Features and Bug Fixes:** Outdated crates may lack new features, optimizations, or bug fixes introduced in the newer versions. This can lead to suboptimal performance, unexpected behavior, or application crashes.

Description: A recent cargo audit has detected an out-of-date Rust crate within the project. Cargo audit is a tool that analyzes the Rust project's dependency tree and reports any known security vulnerabilities or outdated dependencies. Using outdated crates can introduce potential risks and negatively affect the performance, stability, and security of the application.



```
Crate:     secp256k1
Version:   0.21.3
Warning:   unsound
Title:     Unsound API in `secp256k1` allows use-after-free and invalid deallocation from safe code
Date:      2022-11-30
ID:        RUSTSEC-2022-0070
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0070
Dependency tree:
secp256k1 0.21.3
├── web3 0.19.0
│   ├── fast-bridge-service 0.1.0
│   └── eth_client 0.18.0
│       └── fast-bridge-service 0.1.0
├── fast-bridge-service 0.1.0
└── eth_client 0.18.0

Crate:     tokio
Version:   1.24.1
Warning:   unsound
Title:     `tokio::io::ReadHalf<T>::unsplit` is Unsound
Date:      2023-01-11
ID:        RUSTSEC-2023-0005
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0005
```

Recommendations: To address this issue, it is recommended to update the out-of-date crate to its latest version, ensuring that any security vulnerabilities, bug fixes, or new features are incorporated into the project.

Status: Resolved

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.