Blaize.Security

October 5th 2022 / V. 1.0

AURORA

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

Rainbow Bridge contract's source code was taken from the repository provided by the Aurora team.

SCORE **9.6**/10

The scope of the project includes **Rainbow Bridge** set of contracts**:**

1/ BridgeToken.sol
2/ BridgeTokenFactory.sol
3/ BridgeTokenProxy.sol
4/ ProofConsumer.sol
5/ ResultsDecoder.sol
6/ ERC20Locker.sol
7/ Locker.sol
8/ ERC20MetadataLogger.sol

1/ bridge-common/src/lib.rs
2/ bridge-common/src/prover.rs
3/ token-locker/src/lib.rs
4/ token-locker/src/token_receiver.rs
5/ token-locker/src/unlock_event.rs

Repository:

https://github.com/aurora-is-near/rainbow-token-connector

Initial commit:

- fd83d33ade1e5b6e79370805e328f06528f7665d

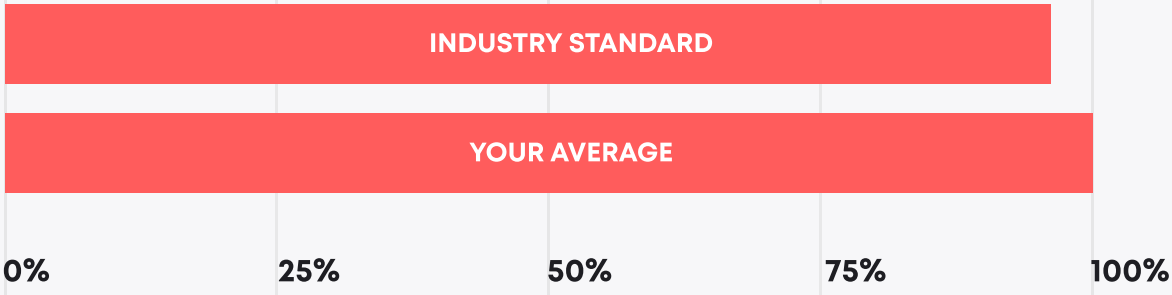
Last-audited commit:

- a40537036d12ede147753c6b2487f27fad9af28d

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the Rainbow Bridge protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **Rainbow Bridge** smart contracts conducted during **August 17th, 2022 - October 5th, 2022.**

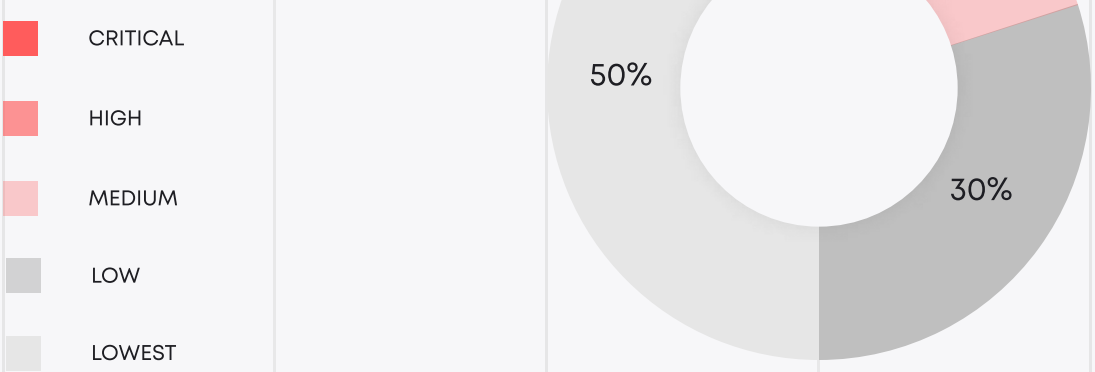
Testable code



The testable code has sufficient coverage to correspond to the industry standard of 95%.

The scope of the audit includes the unit test coverage, which is based on the smart contracts code, documentation, and requirements presented by the Rainbow Bridge team. The coverage is calculated based on the set of the Hardhat framework tests and scripts from additional testing strategies. Also, it needs to mentioned that in order to ensure the full security of the contract, the Aurora team has the Immunefi bug bounty program runnning. It encourages further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 10 problems were found. All issues were fixed or verified by the Rainbow Bridge team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	1	1
Medium	1	1
Low	3	3
Lowest	5	5

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and might cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, but best practices should be implemented.

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

We have scanned this smart contract for the commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

We checked the contract for the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets the best practices in the efficient use of gas, code readability.

Automated analysis:

Scanning contract by several publicly available automated analysis tools such as Mythril, Solhint, Slither and Smartdec. Manual verification of all the issues found with these tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. We checked smart contract logic and compared it with the one described in the documentation.

EXECUTIVE SUMMARY

During the audit, the Blaize Security team has audited both Solidity and Rust parts of the Rainbow Bridge protocol. The protocol represents a bridge between Ethereum and Near blockchains.

The Solidity part of the protocol consists of the Bridge Factory, Bridge tokens, and ERC20 locker contract. There is also an additional Proof consumer contract, which validates all the proof neccesary for bridging tokens between networks.

There were no critical issues found during the audit. There was one high and several low-severity issues found in the contracts. The high-severity issue was connected with the ability of the admin of the ERC20 locker contract to withdraw any tokens from the contract. According to the team, such functionality is neccesary for the contract and the role of the admin will be granted to multi-signature wallet to ensure better safety of the funds. All other issues were verified or fixed as well.

The overall security of the Solidity contracts is high enough. Contracts are well-written and tested by the Rainbow Bridge team. Though it should be mentioned that Bridge Factory and Bridge Token contracts are upgradable and their logic can be changed in the future. The Blaize Security team has also prepared its own set of unit-tests to ensure correctness of the logic of Solidity contracts.

Notes for the table:

* Gas usage is set to 9.3 because while the code is well structured, it still uses the obsolete SafeMath library, though it has 0.8.x compiler version.

** Native test coverage is quite high and covers core scenarios though it is still lower than the industry standards. Also, the simulation tests for the Rust part detected a few missing scenarios. Yet, the team of auditors has provided additional scenarios.

EXECUTIVE SUMMARY

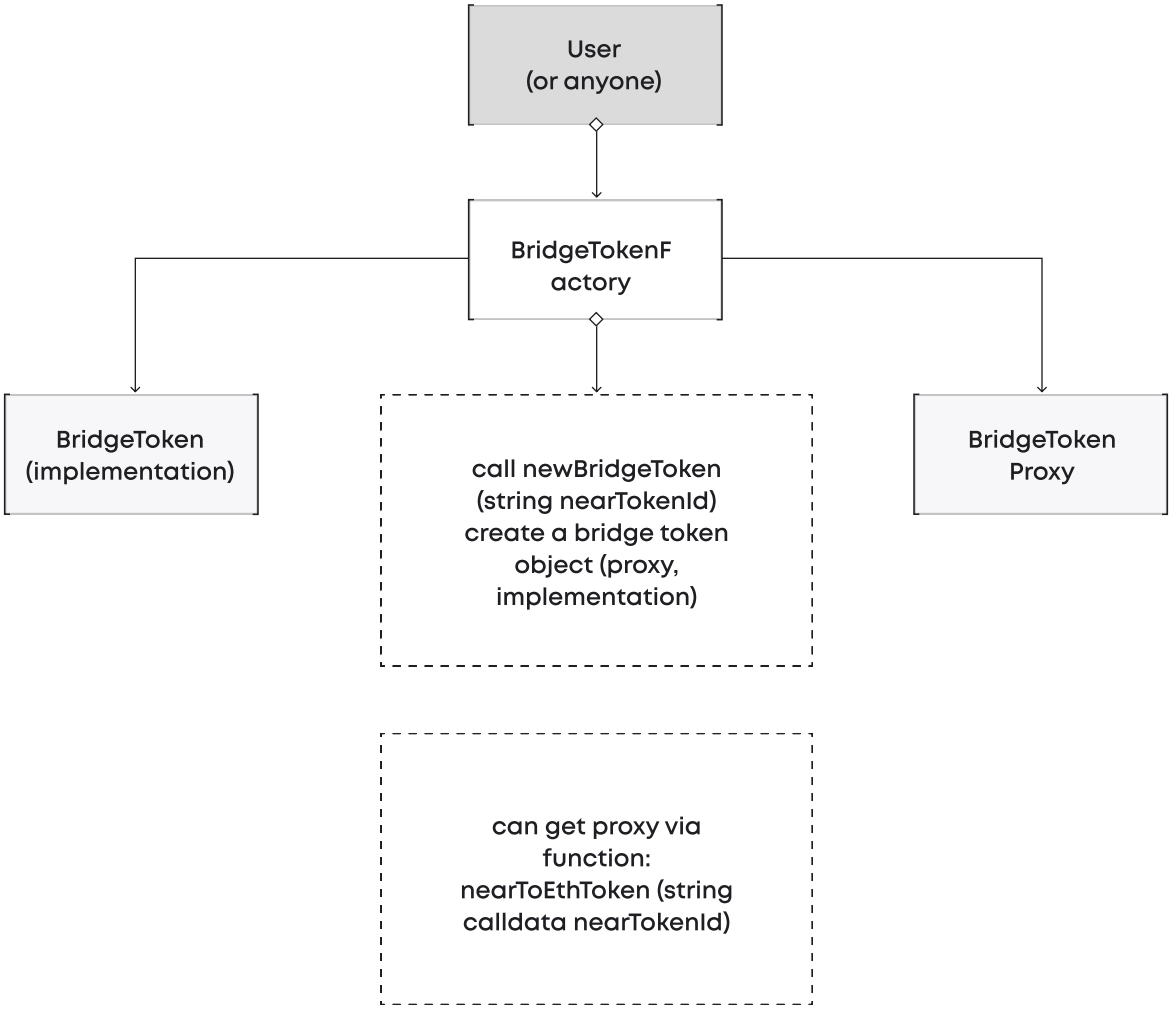
The Rust part of the report refers to the contract on the NEAR side, which is responsible for the native tokens transfer between blockchains. The contract can receive NEAR native tokens via any ERC-20 (NEP-141) token and generate lock`event for a third-party node. Though while the audit of the 3rd party elements (the worker that proceeds with events processing) is out of the scope, the team of auditors has also conducted additional testing to check the system as a whole. Thus, the team has prepared a set of simulation tests and has conducted several runs of manual exploratory testing over the contracts deployed on local testnets.

The Rust part of the audit contains findings connected with the incorrect withdrawal processing, several best practices violations, and unclear functionality.

The Rainbow bridge team has verified/resolved all of the issues.

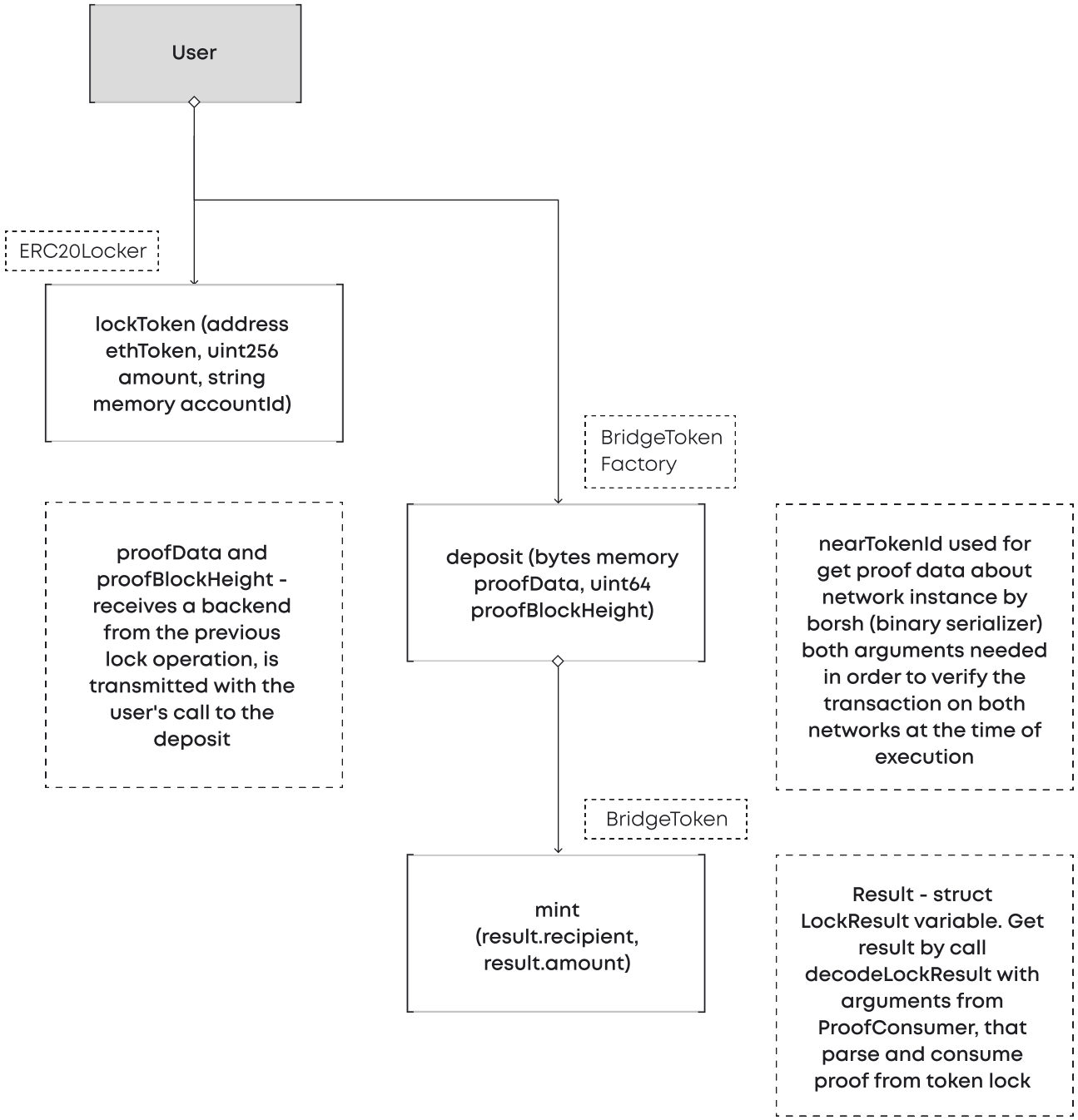
	RATING
Security	9.7
Gas usage and logic optimization*	9.3
Code quality	9.5
Test coverage**	9.5
Total	9.6

CREATION


User
(or anyone)
BridgeTokenF
actory
BridgeToken
(implementation)
call newBridgeToken
(string nearTokenId)
create a bridge token
object (proxy,
implementation)
BridgeToken
Proxy
can get proxy via
function:
nearToEthToken (string
calldata nearTokenId)

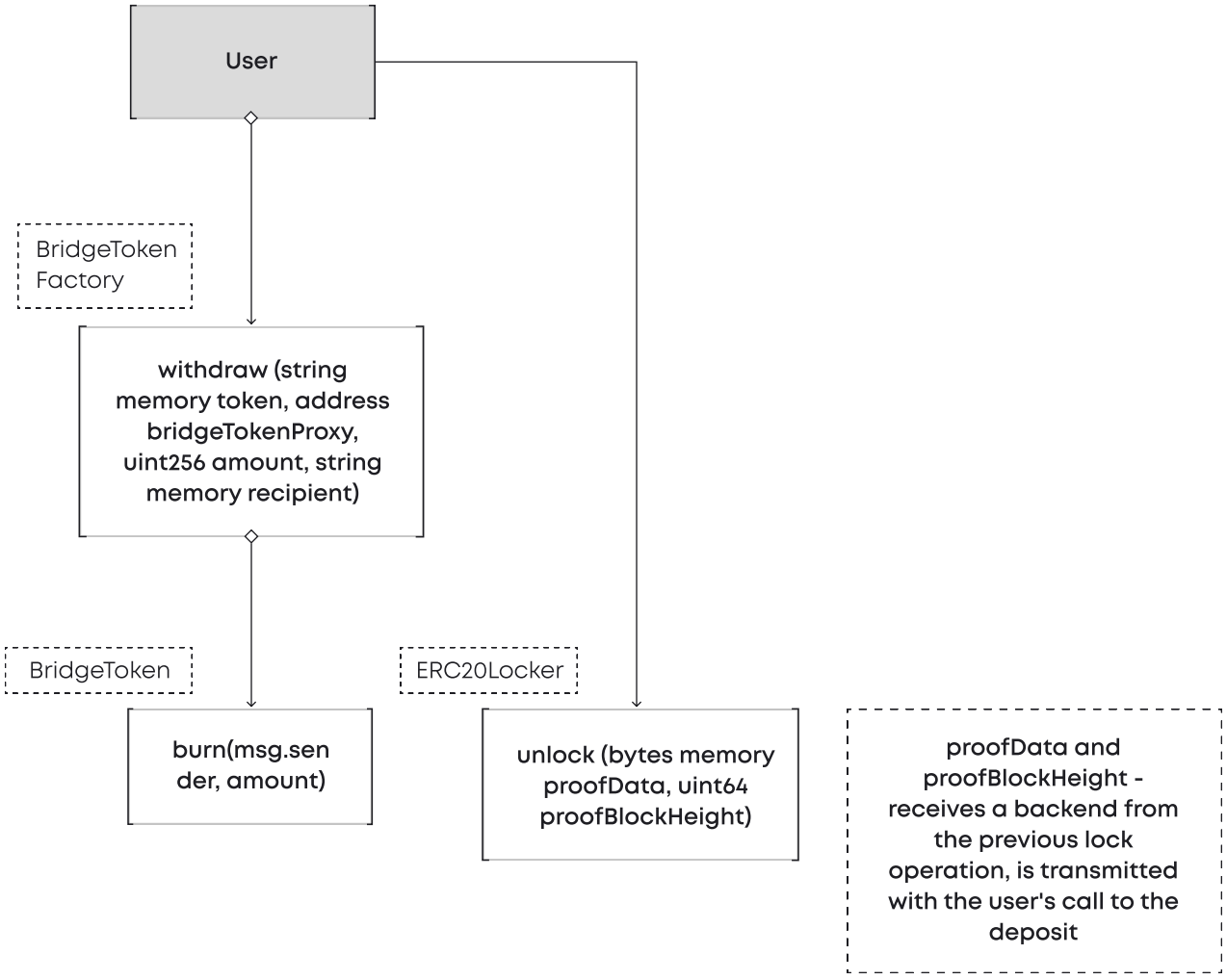
BRIDGE TOKEN PROCESS

GET ETH TOKEN FROM NEAR TOKEN

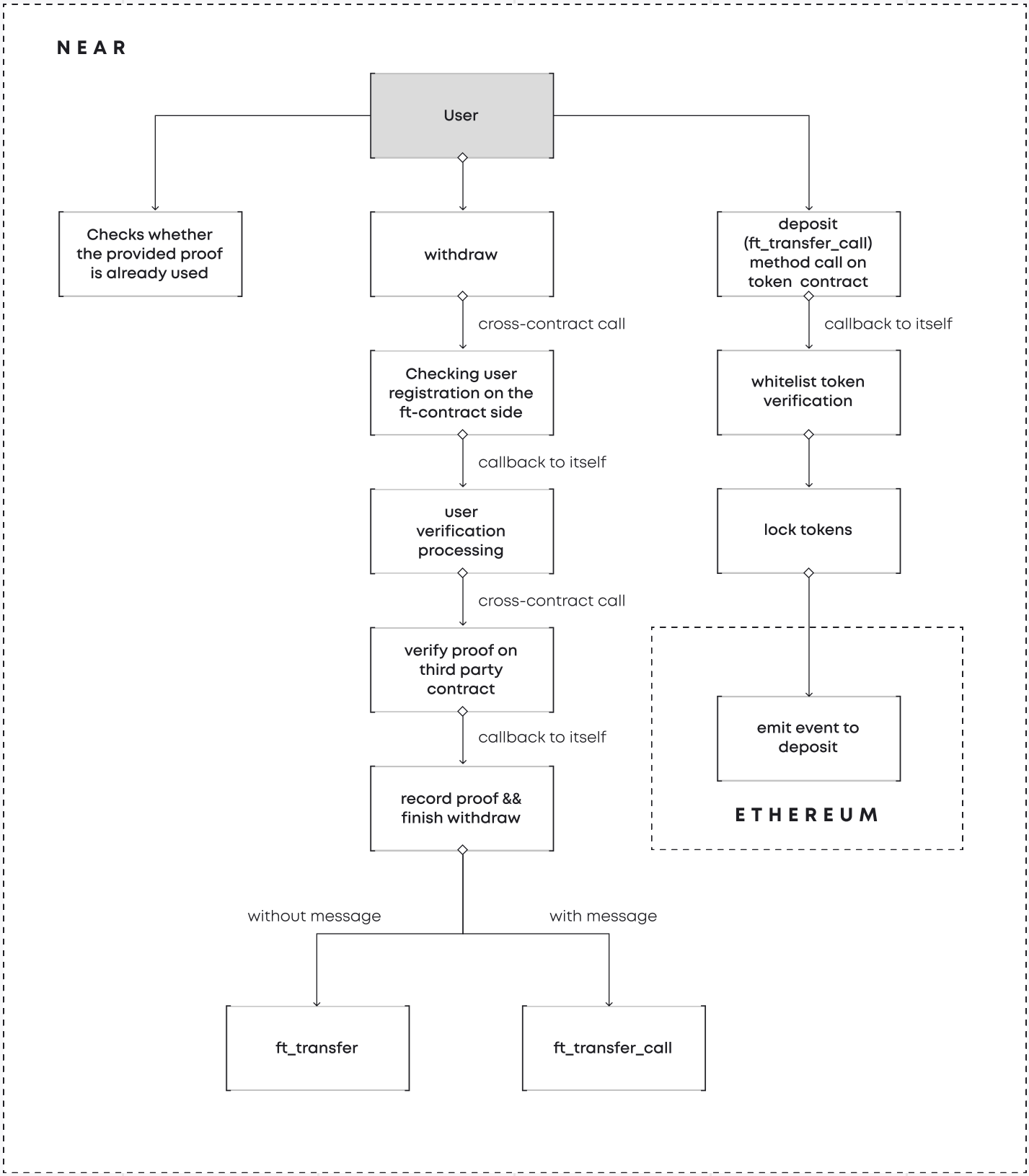


BRIDGE REVERSE PROCESS

GET NEAR TOKEN FROM ETH TOKEN

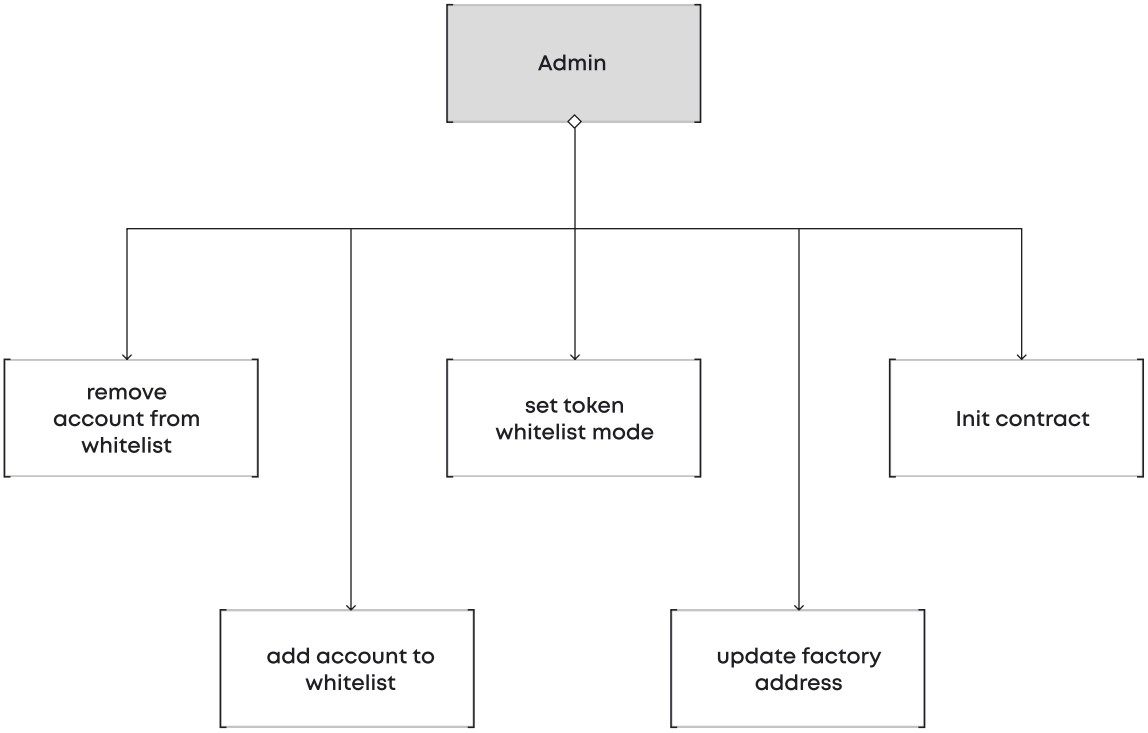


NEAR TO ETHEREUM TOKEN-LOCKER



NEAR TO ETHEREUM TOKEN-LOCKER

NEAR CONTRACT



COMPLETE ANALYSIS (SOLIDITY)

During the audit, the team has found several problems of different severity levels. These problems include the following:

- **High severity.** Unrestricted parseAndConsumeProof() function in ProofConsumer.sol
- **Low severity.** Lack of validation of the parameter in the initialize() function in BridgeTokenFactory.sol
- **Informational.** Commented code in the initialize() function in BridgeTokenFactory.sol

These issues were resolved by the team during the code update. All of these issues were removed from the report after reviewing the commit f7f3fea1b8a0eb4410d62bf5eb82c93c23aa61a4 as they were already fixed by the Rainbow bridge team. Nevertheless, the team has provided additional checks against them.

HIGH-1	✔ Verified

The admin can withdraw any token.

ERC20Locker.sol: adminTransfer().
The contract is supposed to lock tokens on one chain in order to mint them on another one. However, the admin has rights to withdraw any token in any quantity from the contracts.

Recommendation:

Consider adding restrictions to ensure that the admin cannot withdraw certain tokens that represent users' funds. Consider using a multisig or DAO contract as the admin.

From the client.

According to the Rainbow bridge team, such functionality is neccesary in case migration of funds is required (as the contract is not upgradable). Also, in order to provide a better safety for funds, the admin's role is granted to a multi-sig wallet.

LOW-1	✔ Verified

SafeMath usage can be omitted.

ERC20Locker.sol: function lockToken(), line 56.
The set of contracts uses Solidity 0.8.x, which was built in support of overflow and underflow warnings. Thus, SafeMath library can be omitted for gas savings and code simplification.

Recommendation:

Remove SafeMath library.

Post-audit:

The Rainbow team has created an issue for mitigation https://github.com/aurora-is-near/rainbow-token-connector/issues/161
Since the issue is not critical, it can always be fixed later.

LOW-2	✔ Verified

Usage of arbitrary ERC20 tokens.

ERC20Locker.sol: function lockToken().
Anyone can execute lockToken() with any ERC20 token. A malicious actor can create an ERC20 token with reentrancy to exceed the maximum limit of locked tokens (Check in require in line 56). Such a malicious token can also be used to flood the protocol with invalid events, which can prevent the protocol from processing valid events.

Recommendation:

Consider adding a whitelist of allowed tokens or a blacklist for disabling any malicious users or tokens.

From the client.

Such functionality is intended since any token can be bridged from NEAR. It's expected to allow any token to be bridged to NEAR back and forth. Flooding events make no harm to the protocol because it is user's responsibility to finalize the needed event (even though we have an event-relayer).

LOW-3	✔ Resolved

Bridge tokens cannot be paused/unpaused.

BridgeToken.sol: functions pause(), unpause().
The roles of the pauser and the admin are granted to Bridge Token Factory during initialization of the token. However, there is no interface in BridgeTokenFactory.sol to pause or unpause the Bridge token. There is also no interface in BridgeTokenFactory.sol to grant the pauser role to other accounts. Thus, Bridge tokens can't be paused at the moment. The issue is marked as low since both BridgeToken.sol and BridgeTokenFactory.sol are upgradeable and this functionality can always be added later.

Recommendation:

Add the interface to BridgeTokenFactory for pausing/unpausing Bridge tokens.

Post-audit:

An interface for pausing/unpausing tokens was added to BridgeTokenFactory.

LOWEST-1	✔ Verified

Anyone can create a bridge token.

BridgeTokenFactory.sol: function newBridgeToken().
The function for creating bridge tokens is not restricted, so anyone can call it and create bridge tokens. The issue is marked as informational since it might be an intended functionality and should be verified.

Recommendation:

Verify that function should not be restricted or use the onlyRole modifier to restrict it.

From the client:

Due to the logic of the contract, the function for creating bridge tokens should not be restricted.

COMPLETE ANALYSIS (RUST)

MEDIUM-1 ✔ Resolved

Crash on AccountId parse
Line. 405, token-locker/src/lib.rs, function ‘withdraw’

The contract has the withdraw functionality, the flow of which is to check the given proof and give money to the recipient user account. The logic of this movement is a bit extended with new features. The ETH side can add additional messages to the event (data will be a part of the proof), and it will be contained in the same data field where the recipient account is stored. Normally, these strings should be separated with “:”, and the NEAR side should parse it well. Yet, the current implementation contains an issue. The problem appears when the proof from byte array is parsed. It is casted to the NEAR ‘AccountId’ type where the ‘:’ symbol is not allowed. So if the contract receives proof with this additional message, it will crash while trying to make AccountId type from this string. The actual function that should parse this data field receives ‘String’ type and then returns two separate pieces: user’s account and the message itself. This case leads to a crash on a withdraw call and completely blocks money refund to the user. If there is no message, the contract will call ‘ft_transfer’, and the user will not be aware of the money refund. But in the current case with the message passed, the contract will call ‘ft_transfer_call’ and it will notify the user about the money. The team of auditors assumes that this will be the regular way of usage.

Recommendation:

Change the type of the ‘recipient’ field in ‘EthUnlockedEvent’ struct (token-locker/src/lib.rs, line 13) to ‘String’

Post-audit:

The Rainbow team has created and resolved the issue for mitigation https://github.com/aurora-is-near/rainbow-token-connector/issues/163

LOWEST-1	✔ Resolved

Whitelist mode is always enabled
Line. 109, token_locker/src/lib.rs, function ǹew`

Due to the code implementation, the whitelist flag is always enabled, and it cannot be changed in case the owner wants to disable it. In case it should always be on, there is no need to add this flag and its checks through the code.

Recommendation:

Create a possibility to change the whitelist flag or remove all the code related to its usage if the flag is always 'true'.

Post-audit:

The Rainbow team has created and resolved the issue for mitigation https://github.com/aurora-is-near/rainbow-token-connector/issues/160

LOWEST-2	✔ Verified

Little note on the withdrawal flow. As we can see, when the user on the ETH side makes a deposit, the node creates a proof based on the event. Then, the relayer sends that proof into the withdraw function on the NEAR side to get user's money on the current side. Our minor concern is that native NEAR tokens are transferred to the user's account via an unknown ERC-20 token, which is not a part of the whitelist (or at least the code does not check it), and there is no proof that this token has enough funds. This case has to be shared with the dev team just in case.

Post audit.

The tokens are expected to use the withdraw functionality only if they were previously bridged from NEAR to Ethereum and back. So we always have the needed supply available on Ethereum (unless the contract was exploited in some unknown way or has been migrated to some other contract).

LOWEST-3	✔ Verified

The team of auditors has manually tested the entire users flow on the Token locker contract (deposit and withdraw functionality) since initial tests from the client's dev team remain on the unit level, which unfortunately doesn't cover all NEAR callbacks that appear to be in each user flow. There might have been critical issues with gas consumption, but after manual testing, all callbacks were proven to work well. This is just for the information purposes as a reminder about a few uncovered callbacks in the original test suite.

LOWEST-4	✔ Resolved

Withdraw attached deposit does not return change
Line. 181, token_locker/src/lib.rs, function `withdraw`

When the user calls `withdraw`, they will attach deposit as payment for the storage of their proof record. However, if the user sends too much along with the call, there is no way to get the change back.

Recommendation:

Verify the behavior or provide the validation for the change for extensive amount passed.

Post-audit:

The team has verified that all the interaction (deposit/withdrawal) with the storage balance happens in the token contract using the NEP-145 standard.

	BridgeToken.sol	BridgeToken Factory.sol	BridgeToken Proxy.sol
✔ Re-entrancy	Pass	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass	Pass
✔ Default Public Visibility	Pass	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass	Pass
✔ External Contract Referencing	Pass	Pass	Pass
✔ Short Address/Parameter Attack	Pass	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass	Pass
✔ Race Conditions/Front Running	Pass	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass	Pass
✔ Floating Points and Precision	Pass	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass	Pass
✔ Signatures Replay	Pass	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass	Pass

	Proof Consumer.sol	Results Decoder.sol	ERC20Locker.sol
✓ Re-entrancy	Pass	Pass	Pass
✓ Access Management Hierarchy	Pass	Pass	Pass
✓ Arithmetic Over/Under Flows	Pass	Pass	Pass
✓ Delegatecall Unexpected Ether	Pass	Pass	Pass
✓ Default Public Visibility	Pass	Pass	Pass
✓ Hidden Malicious Code	Pass	Pass	Pass
✓ Entropy Illusion (Lack of Randomness)	Pass	Pass	Pass
✓ External Contract Referencing	Pass	Pass	Pass
✓ Short Address/Parameter Attack	Pass	Pass	Pass
✓ Unchecked CALL Return Values	Pass	Pass	Pass
✓ Race Conditions/Front Running	Pass	Pass	Pass
✓ General Denial Of Service (DOS)	Pass	Pass	Pass
✓ Uninitialized Storage Pointers	Pass	Pass	Pass
✓ Floating Points and Precision	Pass	Pass	Pass
✓ Tx.Origin Authentication	Pass	Pass	Pass
✓ Signatures Replay	Pass	Pass	Pass
✓ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass	Pass

	Locker.sol	ERC20MetadataLogger.sol
✔ Re-entrancy	Pass	Pass
✔ Access Management Hierarchy	Pass	Pass
✔ Arithmetic Over/Under Flows	Pass	Pass
✔ Delegatecall Unexpected Ether	Pass	Pass
✔ Default Public Visibility	Pass	Pass
✔ Hidden Malicious Code	Pass	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass	Pass
✔ External Contract Referencing	Pass	Pass
✔ Short Address/Parameter Attack	Pass	Pass
✔ Unchecked CALL Return Values	Pass	Pass
✔ Race Conditions/Front Running	Pass	Pass
✔ General Denial Of Service (DOS)	Pass	Pass
✔ Uninitialized Storage Pointers	Pass	Pass
✔ Floating Points and Precision	Pass	Pass
✔ Tx.Origin Authentication	Pass	Pass
✔ Signatures Replay	Pass	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES BY THE RAINBOW TEAM (SOLIDITY)

BridgeToken

- ✓ can create an empty token (91ms)
- ✓ cant create a token if the token already exists (68ms)
- ✓ can update metadata (132ms)
- ✓ cannot update metadata with old block height (138ms)
- ✓ cannot update metadata of a nonexistent token (74ms)
- ✓ cannot update metadata when paused (93ms)
- ✓ user cannot update metadata (68ms)
- ✓ deposit token (120ms)
- ✓ can't deposit if the contract is paused (93ms)
- ✓ withdraw token (133ms)
- ✓ can't withdraw a token when paused (129ms)
- ✓ can deposit and withdraw after unpausing (170ms)
- ✓ upgrade the token contract (239ms)
- ✓ user can't upgrade the token contract (208ms)

14 passing (4s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
BridgeToken.sol	89.47	100	80
BridgeTokenFactory.sol	93.88	62.5	84.62
BridgeTokenProxy.sol	100	100	100
ProofConsumer.sol	64.71	25	100
ResultsDecoder.sol	0	100	0
All files	**76**	**43.75**	**79.31**

Contract: TokenLocker

- ✓ lock to NEAR (413ms)
- ✓ unlock from NEAR (603ms)
- ✓ should not exceed max token limit (5366ms)
- ✓ cannot unlock, the proof is from the ancient block (5821ms)
- ✓ admin functions (1156ms)

Pausability

- ✓ Lock method (6142ms)
- ✓ Unlock method (7429ms)

Contract: UpdateProver

- ✓ updateContract (612ms)

8 passing (1m)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
ERC20Locker.sol	100	75	83.33
Locker.sol	100	56.25	100
All files	**100**	**65.62**	**91.66**

Contract: ERC20MetadataLogger

- ✓ Should log erc20 metadata (383ms)

1 passing (385ms)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
ERC20MetadataLogger.sol	100	100	100
All files	**100**	**100**	**100**

CODE COVERAGE AND TEST RESULTS FOR ALL FILES BY THE BLAIZE.SECURITY TEAM (SOLIDITY)

Contract: BridgeToken

- ✓ Should return true if the provided address is bridge token and false otherwise (107ms)
- ✓ Should return near token for eth token (47ms)
- ✓ Should revert returning near token for eth token if provided token is not registered (45ms)
- ✓ Should revert returning eth token for near token if provided near token is not registered
- ✓ Should revert depositing if token is not registered (58ms)
- ✓ Should revert withdrawing if token is not registered
- ✓ Should revert updating implementation of token if token is not registered

Contract: ProofConsumer

- ✓ Should revert creating proof consumer if nearTokenFactory is empty
- ✓ Should revert creating proof consumer if prover is zero address
- ✓ Should revert if proof is not valid (57ms)
- ✓ Should revert if proof is from the ancient block (68ms)
- ✓ Should revert if proof is used more than once (96ms)
- ✓ Should revert if proof produced from wrong near factory (61ms)

Contract: ResultsDecoder

- ✓ Should decode lock result

14 passing (1s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
BridgeToken.sol	89.47	100	80
BridgeTokenFactory.sol	100	100	100
BridgeTokenProxy.sol	100	100	100
ProofConsumer.sol	100	87.5	100
ResultsDecoder.sol	100	100	100
All files	**97.9**	**97.5**	**96**

Contract: ERC20Locker

- ✓ Should not create ERC20Locker if prover address = zero address (1007ms)
- ✓ Should not create ERC20Locker if token factory address is invalid (1073ms)
- ✓ Should not unlock token if flag != 0 (2303ms)
- ✓ .tokenFallback() function - always passing (1398ms)
- ✓ Should not reused burn event proof (2652ms)

5 passing (1s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
ERC20Locker.sol	100	100	100
Locker.sol	100	75	100
All files	**100**	**87.5**	**100**

CODE COVERAGE AND TEST RESULTS FOR ALL FILES BY THE RAINBOW TEAM (RUST)

bridge-common/src/result_types.rs

- ✓ generate_result_prefixs

token-locker/src/lib.rs

- ✓ test_lock_unlock_token
- ✓ test_only_admin_can_pause
- ✓ test_blocked_token - should panic
- ✓ test_account_not_in_whitelist - should panic
- ✓ test_token_not_in_whitelist - should panic
- ✓ test_account_in_whitelist
- ✓ test_remove_account_from_whitelist - should panic
- ✓ test_tokens_in_whitelist
- ✓ test_accounts_in_whitelist

token-locker/src/unlock_event.rs

- ✓ fuzzing_eth_unlocked

CODE COVERAGE AND TEST RESULTS FOR ALL FILES BY THE BLAIZE.SECURITY TEAM (RUST)

bridge-common/src/lib.rs

- ✓ test_parse_recipient_with_colon
- ✓ test_parse_recipient_without_colon

bridge-common/src/prover.rs

- ✓ test_validate_eth_address
- ✓ test_validate_eth_address_wrong_address - should panic
- ✓ test_get_key
- ✓ test_from_log_entry_data
- ✓ test_to_log_entry_data

bridge-common/src/result_types.rs

- ✓ test_constructors

token-locker/src/lib.rs

- ✓ test_add_empty_token_to_whitelist - should panic
- ✓ test_whitelist_mode_disabled
- ✓ test_log_metadata
- ✓ test_withdraw_wrong_address - should panic
- ✓ test_account_not_registered- should panic
- ✓ test_withdraw_event_reusing - should panic
- ✓ test_used_proof
- ✓ test_update_factory_address
- ✓ test_eth_account_message
- ✓ test_deposit_simulation - simulation test
- ✓ test_withdraw_simulation - simulation test

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim that the investigated product can meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool that helps investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.