QuillAudits

AUDIT REPORT

November 2025

For

Onestable

Table of Content

Executive Summary 03

Number of Security Issues per Severity 05

Summary of Issues 06

Checked Vulnerabilities 07

Techniques and Methods 09

Types of Severity 11

Types of Issues 12

Severity Matrix 13

High Severity Issues 14

1. Missing Domain Separation in Relayer Signatures 14

Medium Severity Issues 15

2. Bridge does not support fee-on-transfer or rebasing tokens 15

Low Severity Issues 16

3. Admin can set unsafe confirmation periods leading to unstable behaviour 16

4. Lock and burn records are kept forever and never cleaned up 17

5. Initialization pattern silently requires initializer caller to be the default admin 18

6. Confirm functions bypass expiry, creating inconsistent timing semantics 19

Informational Issues 20

7. No event emitted when maxConfirmationPeriod is updated 20

Automated Tests 21

Closing Summary & Disclaimer 22

Executive Summary

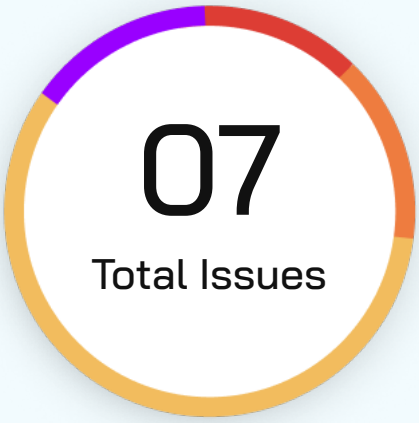
Project Name	OneStable
Protocol Type	Cross Chain Bridge
Project URL	https://onestable.io/
Overview	The Onestable suite implements a two-sided cross-chain bridge with relayer adapters that coordinate sending and receiving transferable messages and tokens between a source and destination chain. The Source and Destination Bridgecontracts handle deposit/withdraw flows, message hashing, and finalization checks, while the RelayerAdapter contracts provide relayer-facing entrypoints, signature verification, and gas/payment accounting. The system enforces replay protection, nonce and id tracking, and uses packed calldata/hash checks to ensure integrity of cross-chain payloads. Access control and fee handling are delegated to adapter layers so the core bridge logic stays minimal and auditable. Event emissions at each step provide on-chain proofs for external relayers and watchers. Overall the design separates core finalization logic from relayer/payment mechanics for clearer security boundaries.
Audit Scope	The scope of this Audit was to analyze the OneStable Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/OneStable-limited/onestable-bridge
Branch	Main
Contracts in Scope	OnestableRelayerAdapter.sol OnestableSourceBridge.sol OnestableSourceRelayerAdapter.sol OnestableDestinationBridge.sol OnestableDestinationRelayerAdapter.sol
Commit Hash	c65b2b0efd7bbe9d8b99e0da2eab54551e390e44
Language	Solidity
Blockchain	EVM

Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	18th November 2025
Updated Code Received	19th November 2025
Review 2	24th November 2025 - 25th November 2025
Fixed In	15927638b65d723756a6111d6467c3cea2a6e6bb

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0(0.0%)
High	1 (14.0%)
Medium	1 (14.0%)
Low	4 (58.0%)
Informational	1 (14.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged Design Intended	0	0	**1**	0	**1**
Partially Resolved	0	0	0	0	0
Resolved	0	**1**	0	**4**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Missing Domain Separation in Relayer Signatures	High	Resolved
2	Bridge does not support fee-on-transfer or rebasing tokens	Medium	Acknowledged - Design Intended
3	Admin can set unsafe confirmation periods leading to unstable behaviour	Low	Resolved
4	Lock and burn records are kept forever and never cleaned up	Low	Resolved
5	Initialization pattern silently requires initializer caller to be the default admin	Low	Resolved
6	Confirm functions bypass expiry, creating inconsistent timing semantics	Low	Resolved
7	No event emitted when maxConfirmationPeriod is updated	Informational	Acknowledged - Design Intended

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Missing Domain Separation in Relayer Signatures

Resolved

Path

OnestableRelayerAdapter.sol

Function

`verifySignature`

Description

The adapter verifies signatures over a raw keccak256 hash that contains only the message fields.

The signature does not include any domain separation such as:

- the bridge contract address
- the protocol name/version
- the actual chainId of the execution environment

Although fields like srcChainId appear in the payload, these are usercontrolled data fields and do not bind the signature to this specific contract or deployment. Without proper domain separation, a signature intended for one deployment, environment, or contract instance may be valid on another contract that uses the same message format. This breaks a core guarantee of EIP-712, which exists specifically to prevent cross-deployment and cross-protocol signature replay:

EIP-712 Standard:**https://eips.ethereum.org/EIPS/eip-712**

Impact

- A signature intended for Bridge A can be reused on Bridge B if both share the same signer and message format.
- Signatures may be replayed across forks, testnets, or cloned deployments.
- Future upgrades cannot safely reuse the same signer without introducing replay risk.
- Long-term decentralization or validator rotation becomes unsafe.

Recommendation

Adopt EIP-712 typed structured signatures and include a domain separator containing:

- protocol name
- version
- block.chainid
- address(this)

This ensures the signature is cryptographically bound to this specific contract and cannot be replayed elsewhere.

Medium Severity Issues

Bridge does not support fee-on-transfer or rebasing tokens

Acknowledged
Design Intended

Path

OnestableSourceBridge.sol, OnestableDestinationBridge.sol

Function Name

`lockTokens, releaseTokens, overall mint/burn accounting logic`

Description

The bridge treats the configured token as a standard ERC20 and assumes that the amount the user specifies is exactly the amount the contract will receive. When a user locks tokens, the contract increases totalLockedSupply by the requested amount and then performs safeTransferFrom with the same value. The code also relies on the assumption that token.balanceOf(address(this)) will always match totalLockedSupply.

This assumption does not hold for fee-on-transfer tokens or rebasing tokens. Such tokens may deduct a fee during transfers or may adjust balances automatically over time. In either case, the actual number of tokens held by the bridge can differ from the number expected by the contract's internal accounting.

Impact

If a non-standard ERC20 token is used, the contract's internal supply tracking will diverge from the real token balance. This can cause unlock or release operations to fail, even when the contract believes sufficient tokens are available. In extreme cases, the discrepancy can permanently trap a portion of the locked assets, causing loss of redeemable value.

Example

1. A token deducts a 1% transfer fee.
2. The user calls lockTokens with an amount of 100.
3. totalLockedSupply is set to 100, but only 99 tokens are actually received.
4. Later, releaseTokens attempts to transfer 100 tokens to the user.

The contract does not have 100 tokens, leading to a failed release or broken parity checks.

Recommendation

If the bridge will only be used with standard ERC20 tokens, this requirement should be clearly documented and enforced through governance and deployment procedures.

Otherwise, update the accounting logic to rely on the actual balance received rather than the user-provided amount. This ensures the bridge remains compatible with fee-on-transfer or rebasing token behaviour.

OneStable Team's Comment

Onestable Bridge is strictly built for only official USDC tokens, so support for fee-on-transfer or rebasing tokens is not needed as per the requirements.

Low Severity Issues

Admin can set unsafe confirmation periods leading to unstable behaviour

Resolved

Path

OnestableSourceBridge.sol, OnestableDestinationBridge.sol

Function Name

`setMaxConfirmationPeriod, setMaxConfirmationPeriod`

Description

The confirmation period is fully controlled by the administrator through setMaxConfirmationPeriod. There are no lower or upper bounds enforced by the contract.

If the confirmation period is set too low, for example one or two seconds, it becomes very difficult for relayers to process locks and burns before expiry. Many operations will time out and require reverts. If the period is set extremely high, for example many months, users may need to wait an unreasonably long time before timeout based recovery paths become available.

Impact

Incorrect configuration can cause real operational problems. Small values can produce frequent timeouts and failed bridging. Large values can cause users to wait a very long time before any revert is allowed if the relayer stops working.

Since configuration is fully trusted and under admin control, this is not an exploit by an attacker, but a genuine operational risk for the project.

Recommendation

Enforce reasonable minimum and maximum bounds for maxConfirmationPeriod at the contract level, or define those bounds clearly in documentation and commit to operational policies that respect them.

Lock and burn records are kept forever and never cleaned up

Resolved

Path

OnestableSourceBridge.sol, OnestableDestinationBridge.sol

Function Name

`lockRecords mapping, burnRecords mapping`

Description

Every lock on the source bridge and every burn on the destination bridge creates a record that is stored in a mapping. Once a record has been confirmed or reverted, it remains in storage permanently. There is no mechanism to delete or recycle it.

Over time, especially in a high volume system, this will cause the bridge contracts to accumulate a very large state.

Impact

This does not directly affect correctness of the protocol, but it increases the storage footprint and long term gas costs for the network.

Recommendation

Consider deleting lock and burn records once they have reached a terminal state and are no longer needed. If full history is required off chain, events can be used as the source of truth while storage is kept lean.

Initialization pattern silently requires initializer caller to be the default admin

Resolved

Path

OnestableDestinationBridge.sol

Function Name

`initialize and setAllowedSourceTokens`

Description

The destination bridge initialize function assigns the DEFAULT_ADMIN_ROLE to the _defaultAdmin address and then immediately calls setAllowedSourceTokens in a loop. The setAllowedSourceTokens function is protected by onlyRole(DEFAULT_ADMIN_ROLE), which checks the role of msg.sender, not _defaultAdmin.

During initialization, msg.sender is the account that calls initialize. If this caller is not the same address as _defaultAdmin, then setAllowedSourceTokens will revert because the caller does not yet have the admin role, even though _defaultAdmin has been granted it. As a result, the entire initialize call fails unless msg.sender and _defaultAdmin are the same address.

This creates a hidden requirement on deployment: the initializer caller must be the same account that is being set as default admin, otherwise initialization cannot complete.

Recommendation

Refactor setAllowedSourceTokens into an internal helper that can be called during initialize without role checks and an external admin function that simply wraps the internal helper and enforces onlyRole(DEFAULT_ADMIN_ROLE). This avoids initializer reverts while still keeping post-deployment configuration restricted to the default admin.

Confirm functions bypass expiry, creating inconsistent timing semantics

Resolved

Path

OnestableSourceBridge.sol, OnestableDestinationBridge.sol

Function Name

`confirmLock, confirmBurn`

Description

When a user locks or burns tokens, the bridge records a maxConfirmationTimestamp. This value suggests that confirmations are expected to occur within a specific time window.

However, the confirmLock and confirmBurn functions do not check expiration at all. They accept confirmations indefinitely, as long as the record exists and has not been reverted or previously processed. In contrast, other parts of the system such as revertLockedTokens, revertBurnedTokens, releaseTokens, and mintTokens do enforce the timestamp strictly.

As a result, the protocol ends up with mixed timing rules: confirmations ignore expiry, but reverts and releases depend on it. This inconsistency can confuse integrators and does not match the intuitive meaning of a "confirmation deadline."

Impact

Although this behavior does not directly lead to loss of funds or token inflation, it weakens the conceptual model of time-bounded confirmations. A lock or burn considered "expired" by the user interface or monitoring systems may still be confirmed successfully long after the deadline, which can make the system harder to reason about and lead to operational misunderstandings

Recommendation

If the protocol intends for confirmations to be strictly time bounded, a timestamp check should be added so confirmations only succeed within the allowed window.

If, however, confirmations are intentionally allowed after expiry to preserve liveness during relayer outages or network delays, this behavior should be clearly documented as part of the protocol's design so users and integrators do not interpret the deadline as applying to confirmations.

OneStable Team's Comment

As per the protocol design its, needed, So confirmations are intentionally allowed after expiry to preserve liveness during relayer outages or network delays.

We added a dev note in methods to explain this to the integrators.

Informational Issues

No event emitted when maxConfirmationPeriod is updated

Acknowledged
Design Intended

Path
OnestableSourceBridge.sol ,OnestableDestinationBridge.sol

Function Name
`setMaxConfirmationPeriod, setMaxConfirmationPeriod`

Description
The bridge allows administrators to update the confirmation window using setMaxConfirmationPeriod. This value is important for determining when locks and burns expire. However, the function does not emit any event that records when this change occurred.

Without an event, off-chain systems such as monitoring dashboards, indexers, or operational tooling cannot easily track historical configuration changes related to timing behaviour.

Recommendation
Emit a dedicated event such as MaxConfirmationPeriodUpdated(uint256 oldValue, uint256 newValue) whenever the value is updated. This improves observability and simplifies off-chain monitoring.

OneStable Team's Comment
We don't need this event, because this value is used in lockTokens & burnTokens method & events emitted from these methods have setted value of maxConfirmationPeriod, which off-chain relayer uses for message processing.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of OneStable Smart Contracts. We performed our audit according to the procedure described above.

Issues of high, medium,low and informational severity. OneStable Team resolved a few and acknowledged the remaining issues mentioned in the report.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

November 2025

For

Onestable

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com