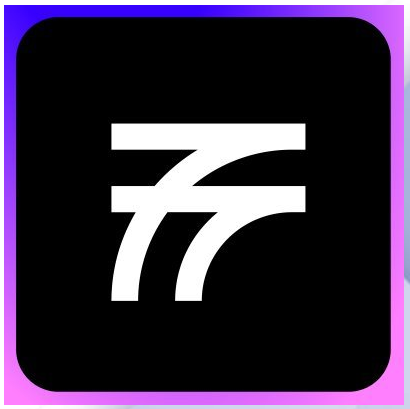
Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Bridge Contracts

Veridise Inc.

July 29, 2025

► **Prepared For:**

Fluent
https://www.fluent.xyz

► **Prepared By:**

Alberto Gonzalez
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

Jul. 29, 2025	V3
Jul. 08, 2025	V2
May. 30, 2025	V1

Contents

1 Executive Summary

From May. 15, 2025 to May. 22, 2025, Fluent engaged Veridise to conduct a security assessment of their Bridge Contracts. The project implements the core on-chain infrastructure for a rollup system, comprising smart contracts deployed on the parent (L1) chain. These contracts coordinate the submission and verification of L2 blocks and enable cross-chain messaging between the two chains. Veridise conducted the assessment over 12 person-days, with 2 security analysts reviewing the project over 6 days on commit `fd208df`.

The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project summary. The protocol is mainly composed of the following smart contracts:

- `Rollup.sol`. This contract serves as the main interface between L1 and L2, implementing a hybrid optimistic-zk rollup system. It ensures valid L2 state transitions through both a challenge-response mechanism (optimistic component) and zkVM proof verification (ZK component).
- `Bridge.sol`. This contract enables communication between the two chains through a messaging protocol using emitted events and merkle proofs against posted batches in the Rollup contract.
- `ERC20Gateway.sol`. This contract builds a token-specific layer on top of the generic message passing functionality provided by the Bridge contract. It handles all the complexity of token deployment, minting, burning, and maintaining the relationship between native and pegged tokens.
- `ERC20PeggedToken.sol`. This contract defines the pegged token standard built on top of the ERC20, used on the Bridge. It includes minting and burning capabilities restricted to authorized gateways.

Code Assessment. The Bridge Contracts developers provided the source code of the Bridge Contracts contracts for review. The codebase appears to be mostly original code written by the Bridge Contracts developers. While functional, it includes minimal inline documentation, with limited comments on functions, storage variables, and overall system behavior.

To facilitate the Veridise security assessment, the Bridge Contracts developers met with the audit team to go over a high-level walkthrough of the protocol. This session helped clarify the system's architecture, core components, trust model, and intended execution flow.

The codebase was accompanied by a test suite that sufficiently covered core functionalities such as message passing between layers, batch submissions, and challenge workflows. However, as noted in the recommendation section, the test suite lacked adversarial and edge case scenarios, which are essential for testing protocol robustness under hostile conditions.

Summary of issues detected. The audit uncovered 22 issues, 11 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, auditors discovered that missing access controls in both the batch submission process and the Queue contract, which could lead to liveness disruption and manipulation of the Bridge queue (V-FLNT-VUL-001, V-FLNT-VUL-002); lack of source chain sender verification in cross-chain message delivery, exposing the system to spoofing and potential loss of funds (V-FLNT-VUL-003); and replayability of rolled-back L1 messages as L2 messages, allowing double-spend attacks (V-FLNT-VUL-004).

Additional high-severity issues involve a flawed rollup corruption check that can misreport a healthy system as corrupted (V-FLNT-VUL-005); invalid batches being approved during the challenge window (V-FLNT-VUL-006); disruption of L1–L2 communication via crafted messages to the Bridge contract (V-FLNT-VUL-007); return bomb attacks halting cross-chain messaging (V-FLNT-VUL-008); and undeliverable deposits potentially causing rollup deadlocks (V-FLNT-VUL-009). Incompatibilities in block number comparisons affecting L1-to-L2 delivery (V-FLNT-VUL-010) and the lack of a decimals override in ERC20PeggedToken leading to possible integration issues (V-FLNT-VUL-011) were also flagged.

The auditors also identified 2 medium-severity issues. Notably, missing state updates in the forceRevertBatch function could lead to liveness failures and approval of invalid batches (V-FLNT-VUL-012), while flaws in the challenger reimbursement mechanism undermine the optimistic rollup feature (V-FLNT-VUL-013). Lastly, the audit reported 4 low-severity issues, 4 warnings, and 1 informational findings.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Bridge Contracts protocol security before deployment:

Additional testing. The source code includes test suites for all the contracts in the codebase including e2e tests. However, the Veridise team think these test suites are insufficient to fully prevent the introduction of bugs and to ensure correct execution under expected conditions:

- The Veridise analysts believe that some of the identified issues could have been prevented with happy path execution tests, such as: ensuring the system remains in a healthy state after different scenarios of challenge resolution (V-FLNT-VUL-005), confirming the expected decimals of a deployed pegged token are consistent with the native token (V-FLNT-VUL-011) and validating that the system can continue posting batches after a forced state revert (V-FLNT-VUL-012).
- Additionally, the Veridise analysts believe that some of the identified issues could have been prevented with negative path execution or adversarial tests, such as: ensuring proper access control in sensitive functionality (V-FLNT-VUL-001, V-FLNT-VUL-002). Checking that the system will prevent common attack vectors such as message spoofing and replay attacks (V-FLNT-VUL-003, V-FLNT-VUL-004).
- Finally, the Veridise team recommends adding test cases to verify that the fixes for the issues outlined in this report function as intended.

L1–L2 message ordering and availability. The current message-passing mechanism between L1 and L2 enforces strict nonce ordering. For instance, the L1 bridge contract maintains a queue of messages sent from L2, and messages are only removed from this queue once a corresponding batch has been posted. Similarly, messages sent from L2 to L1 must also follow a sequential nonce order.

This introduces a risk of denial-of-service scenarios. If a single message becomes undeliverable or its corresponding batch cannot be posted, it can block the processing of all subsequent messages. This issue was highlighted in several findings, including: V-FLNT-VUL-007, V-FLNT-VUL-008, V-FLNT-VUL-009, and V-FLNT-VUL-010.

The Veridise team recommends that the project evaluate all scenarios in which a cross-chain message could become stuck, and either prevent these conditions from happening or implement recovery procedures. Additionally, clearly documenting these procedures will help ensure that future contributors and auditors understand how the system is expected to handle such failure modes.

Protocol design documentation. The Veridise team recommends that the project expand its documentation regarding the expected behavior and design rationale of its hybrid rollup system. While the implementation of on-chain contracts was available and reviewed, there was limited information outlining the intended system architecture, security assumptions, operational guarantees, or trade-offs made during the design process. This is especially important for complex systems like rollups, where subtle design decisions can have significant security implications.

BlockCommitment aggregation guest code. At the time of the original review, the Rollup contract was integrated with a zkVM application responsible for aggregating BlockCommitments into batches and generating a proof. However, the guest (client-side) code for this aggregation phase was not available to the auditors. The Veridise team assumed that the Rollup contract integrated with this component correctly.

During the fix review, the developers introduced a notable change to the Rollup contract's design: instead of proving and challenging at the batch level, the updated approach allows proofs and challenges at the level of individual BlockCommitments within a batch. This change enables coupling the Rollup contract with the developed State Transition Function guest program (audited by Veridise)*. The Veridise team reviewed this design change during a time-limited fix review intended to validate that the changes addressed issues raised by the analysts. Due to the size and scope of this change and the severity of issues found during this review, we recommend comprehensive testing of the modified logic and suggest that the project consider additional external reviews.

* The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits-archive/`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Bridge Contracts	`fd208df`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May. 15–May. 22, 2025	Manual & Tools	2	12 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	4	4	4
High-Severity Issues	7	7	6
Medium-Severity Issues	2	2	2
Low-Severity Issues	4	4	4
Warning-Severity Issues	4	4	4
Informational-Severity Issues	1	1	1
TOTAL	22	22	21

Table 2.4: Category Breakdown.

Name	Number
Logic Error	10
Denial of Service	4
Data Validation	3
Maintainability	3
Access Control	1
Missing/Incorrect Events	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Bridge Contracts's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Can cross-chain interactions lead to unexpected intermediate states that result in a protocol deadlock?
- Does the bridge verify all relevant information, such as the sender's identity, when processing incoming cross-chain messages?
- Does the message data include all the necessary information for the bridge to properly validate and authenticate the message? And is the message data encoded correctly?
- Does the rollup's batch submission and challenge system provide proper incentives for honest actors to participate, and penalize malicious behavior effectively?
- Is the failure to execute bridge messages handled gracefully, allowing for rollback or retry mechanisms?
- Can cross-chain messages be processed out of order, potentially affecting protocol correctness or state consistency?
- When receiving a message with proof, are the withdrawal proof and the block proof verified correctly?
- Are the message and challenge deadlines implemented correctly?
- Is the challenge queue maintained properly?
- Is it possible to force the rollup into a corrupted state? And if so what actions can Fluent take to exit said state?
- Does the token gateway correctly manage origin tokens and pegged tokens? Are the pegged tokens initialized correctly?
- Is it possible to leverage the token bridge actions to perform a double spend?
- Are public inputs to the zk verifier properly validated to prevent invalid or malicious data from compromising the proof verification?

Additionally, during the assessment, the security analysts also aimed to verify if the code is vulnerable to any common Solidity and smart contract specific vulnerabilities, such as:

- Reentrancy
- Missing access control
- Front running risks
- Insufficient input parameter validation
- Replay attacks
- Missing events
- Return bomb attacks

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard. This tool is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of the security assessment was limited to the following files of the source code provided by the Bridge Contracts developers, which contains the smart contract implementation of the Bridge Contracts:

1. `contracts/interfaces`
2. `contracts/libraries`
3. `contracts/rollup/Rollup.sol`
4. `contracts/Bridge.sol`
5. `contracts/ERC20Gateway.sol`
6. `contracts/ERC20PeggedToken.sol`
7. `contracts/ERC20TokenFactory.sol`

During the security assessment, the Veridise security analysts reviewed the list of excluded files and assumed they were correctly implemented. Notable exclusions included `contracts/SP1Verifier.sol`, which implements the Groth16 zk-SNARK verifier, and `contracts/RestakerGateway.sol`, which implements the restaking pool logic.

Methodology. Veridise security analysts reviewed the reports of previous audits for L1 contracts and token bridges similar to the Bridge Contracts, and inspected the provided tests. They then began an in-depth manual review of the codebase, supported by static analysis tools.

Before the security assessment, the Veridise security analysts met with the Bridge Contracts developers to discuss the intended behavior of the protocol and to gain an accurate understanding of its design.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Bridge Contracts.

- The highly privileged roles described below are assumed to behave honestly, within the privilege allotted to them.
- Apart from the highly privileged roles, no other actors in the system are assumed to be trusted. This includes the sequencer that has limited privilege in posting new batches to be accepted, and the challenger that can challenge these batches and require them to be proven.

4.2 Privileged Roles.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensititivy. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assume that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles:
  - `Bridge.bridgeAuthority`. The `bridgeAuthority` can receive messages without a proof, provided they are received in order. This role cannot be reassigned.
  - `Bridge.rollup`. The `rollup` address is the only entity permitted to pop messages from the bridge message queue. It is assumed that the `deployer` sets this address to the correct rollup during contract deployment. This role cannot be reassigned.
  - `ERC20Gateway.bridgeContract`. The `bridgeContract` is capable of receiving both pegged and native tokens. It is assumed that the erc20 gateway `deployer` correctly initializes this address to the bridge during deployment. This role cannot be reassigned.
- Highly-privileged, non-emergency roles:
  - `Rollup.deployer` deploys the rollup contract and sets the initial rollup configuration, which includes the challenge and acceptance block deadlines, the verifier address, the data availability check, the challenge deposit amount, the program verification

key, the genesis hash, the bridge address, the approval block count and the batch size.
    - `Rollup.owner` can set the rollup configuration, including the bridge address, the data availability check and the verifier address. The `deployer` of the rollup is initially assigned owner privileges, which may then be transferred to another address.
    - `Bridge.deployer` deploys the bridge contract and sets the bridge authority, the rollup address, the message receival deadline and initializes a new bridge message queue.
    - `ERC20Gateway.deployer` deploys the gateway contract and initializes the bridge contract and the token factory.
    - `ERC20Gateway.owner` sets the addresses for the erc20 gateway on the other side, its token implementation, and its factory. It can also update the token mapping in the gateway which maps a pegged token to its origin token. The owner of the gateway is initially assigned to the `deployer`, but can be transferred to another address.
- Limited-authority, emergency roles:
    - `Rollup.sequencer` can post new batches for acceptance. If these batches are not challenged within the challenge deadline, they are considered to be approved and cannot be challenged anymore.

Operational Recommendations. Highly-privileged, non-emergency roles such as the rollup `deployer` and `owner` should be operated by a multi-sig contract or a decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations such as the `bridgeAuthority` should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary. Given the `sequencer`'s central role in protocol operation, it should also be rigorously tested under diverse conditions.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which use SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-FLNT-VUL-001	Missing access control in the batch . . .	Critical	Fixed
V-FLNT-VUL-002	Missing access control in the Bridge Queue . . .	Critical	Fixed
V-FLNT-VUL-003	Missing source chain sender in cross-chain . . .	Critical	Fixed
V-FLNT-VUL-004	Rolled-back L1 messages can be replayed . . .	Critical	Fixed
V-FLNT-VUL-005	Incorrect rollup corruption check	High	Fixed
V-FLNT-VUL-006	Approval of invalid batches under the . . .	High	Fixed
V-FLNT-VUL-007	Malicious message to Bridge address halts . . .	High	Fixed
V-FLNT-VUL-008	Return bomb attack can halt cross-chain . . .	High	Partially Fixed
V-FLNT-VUL-009	Deadlock scenario in rollup due to . . .	High	Fixed
V-FLNT-VUL-010	Incompatible block number comparison . . .	High	Fixed
V-FLNT-VUL-011	Missing decimals function override in the . . .	High	Fixed
V-FLNT-VUL-012	Missing state updates in force revert batch . . .	Medium	Fixed
V-FLNT-VUL-013	Challenger reimbursement flaw . . .	Medium	Fixed
V-FLNT-VUL-014	Counting of batch indices should start with . . .	Low	Fixed
V-FLNT-VUL-015	Free batch submission enables DoS and . . .	Low	Fixed
V-FLNT-VUL-016	Excess challenger deposits cannot be retrieved	Low	Fixed
V-FLNT-VUL-017	Incorrect update of tokenMapping may . . .	Low	Fixed
V-FLNT-VUL-018	Use of Solidity transfer is discouraged for . . .	Warning	Fixed
V-FLNT-VUL-019	Solidity best practices	Warning	Fixed
V-FLNT-VUL-020	Missing events	Warning	Fixed
V-FLNT-VUL-021	Fetched gateway address labeled incorrectly	Warning	Fixed
V-FLNT-VUL-022	Unused code	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-FLNT-VUL-001: Missing access control in the batch submission process

Severity	Critical	**Commit**	fd208df
Type	Access Control	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	acceptNextBatch()		
Confirmed Fix At	9162211		

The `Rollup` contract is designed to manage the submission, verification, and challenge of batches of the Fluent system. A key component of this contract is the `acceptNextBatch()` function, which is responsible for accepting new batches of block commitments and updating the rollup state.

Currently, this function relies on trusting on the batch submitter, as it has several ways to mess with the liveness of the rollup system. However, the implementation fails to correctly implement access control, allowing any user to submit a batch.

Impact The lack of access control on `acceptNextBatch()` is particularly severe because the batch submitter has significant privileges that can affect the system liveness:

1. Can prevent owner intervention by submitting many batches, as `forceRevertBatch()` must process all batches sequentially.
2. Can mark as delivered messages of the L1 Bridge queue using invalid batches.
3. Can submit invalid batches without any bond or stake at risk.

Recommendation Implement proper access control on the `acceptNextBatch()` function to ensure that only a designated account can submit batches.

Developer Response The developers have now implemented proper access control on the `acceptNextBatch()` function, as suggested.

5.1.2 V-FLNT-VUL-002: Missing access control in the Bridge Queue contract

Severity	Critical	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Bridge.sol, Queue.sol`		
Location(s)	enqueue(), dequeue()		
Confirmed Fix At	6ae1188		

The `Bridge` contract makes use of a queue to handle the bridge messages. Whenever a new bridge message is received, it is enqueued and after it is processed by the rollup it is dequeued. The `Bridge` contract makes use of the `Queue` contract to implement this queue. And the queue is deployed in the constructor of the `Bridge`, as can be seen in the snippet below.

```
constructor(
  address _bridgeAuthority,
  address _rollup,
  uint256 _receiveMessageDeadline
) {
  bridgeAuthority = _bridgeAuthority;
  rollup = _rollup;
  receiveMessageDeadline = _receiveMessageDeadline;
  if (rollup != address(0)) {
      sentMessageQueue = new Queue();
  }
}
```

Snippet 5.1: Snippet from the constructor in `Bridge.sol`

There is a critical oversight here as the queue is implemented through a contract and not a library. Although the `Bridge` implements access control within its functions which access the queue, since the `Queue` contract is deployed it can be accessed independently of the `Bridge`. Therefore anybody can call sensitive functions on the `Queue` directly as they are permission-less, and manipulate the bridge message queue. See snippet below for context.

```
function dequeue() public returns (bytes32) {
  require(!isEmpty(), "Queue is empty");
  bytes32 value = data[front];
  delete data[front];
  front++;
  return value;
}
```

Snippet 5.2: Snippet from function `dequeue()` in `Queue.sol`

Impact The lack of access control on the `Queue` contract allows to include or remove arbitrary messages from the Bridge queue.

Recommendation Have the bridge inherit the `Queue` contract and apply appropriate access control on the `enqueue()` and `dequeue()` functions.

Developer Response The developers have now correctly implement the `Queue` as a library, and the corresponding changes have been made in the `Bridge` contract as well.

5.1.3 V-FLNT-VUL-003: Missing source chain sender in cross-chain message delivery

Severity	Critical	**Commit**	fd208df
Type	Data Validation	**Status**	Fixed
File(s)	`Bridge.sol`, `ERC20Gateway.sol`		
Location(s)	_receiveMessage(), receivePeggedTokens(), receiveNativeTokens()		
Confirmed Fix At	16501b8		

The Bridge contract implements a cross-chain messaging system that allows communication between L1 and L2 chains. When a message is sent from one chain to another, it's processed by the `_receiveMessage()` function, which forwards the message payload to the target contract.

However, a security issue exists in the message delivery mechanism. When the Bridge contract delivers a message to the target contract, it fails to communicate the original sender's address from the source chain. This is problematic in the `_receiveMessage()` function:

```
1 (bool success, bytes memory data) = _to.call{value: _value}(_message);
```

Snippet 5.3: Code snippet from the `_receiveMessage()` function of the `Bridge.sol` contract

The function simply forwards the `_message` to the target contract without including any information about the original sender (`_from`). This means the receiving contract has no way to verify who initiated the cross-chain message on the source chain.

Impact A malicious user could use this vulnerability to drain the funds from the `ERC20Gateway` contract which relies on the Bridge contract to make ERC20 token transfers between chains. For example, in functions like `receivePeggedTokens()` and `receiveNativeTokens()`, there's no way to validate that the tokens were actually locked or burned by the claimed sender on the source chain.

An attacker could exploit this by:

1. Sending a malicious message in the Bridge directly targeting the `ERC20Gateway` contract in the destination chain.
2. The message would call `receivePeggedTokens()` or `receiveNativeTokens()`.
3. Since there's no sender validation, the attacker would receive tokens on the destination chain without having locked or burned any tokens on the source chain.

This effectively allows unauthorized minting or transferring of tokens on the destination chain.

Recommendation To address this issue is important to implement two types of fixes:

- The Bridge contract should provide the original sender's address to the target contract when delivering messages.
- Add a validation in the `ERC20Gateway` contract that the sender is the `ERC20Gateway` contract of the source chain.

Developer Response The developers now fetch the original sender address from the bridge and verify that the sender is the token gateway of the source chain.

5.1.4 V-FLNT-VUL-004: Rolled-back L1 messages can be replayed as L2 messages

Severity	Critical	**Commit**	fd208df
Type	Data Validation	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	receiveMessageWithProof()		
Confirmed Fix At	fa5d114		

The Bridge contract implements cross-chain messaging between L1 and L2, with two main flows:

1. L1 to L2 messaging via `sendMessage()` and `receiveMessage()`
2. L2 to L1 messaging via `sendMessage()` and `receiveMessageWithProof()`

There is also a rollback mechanism for L1->L2 messages through `rollbackMessageWithProof()` which allows retrieving funds when messages fail to be delivered to L2.

The key issue is that the same Merkle proof verification is used for both L2->L1 messages (`receiveMessageWithProof()`) and L1->L2 rollbacks (`rollbackMessageWithProof()`). Both functions verify proofs against the `withdrawalHash` in the block commitment.

Impact A malicious user could use this vulnerability to cause double spend of funds.

Consider the following scenario with an L1 Bridge contract state where:

- `nonce` equals 4 (4 messages sent from L1 to L2).
- `receivedNonce` equals 4 (4 messages received in L1 from L2).

An attacker can execute the following steps:

1. Send a message from L1 to L2 using `sendMessage()` with some ETH. This message gets assigned nonce 5.
2. Wait for this message to be included in an L2 batch (for a rollback), resulting in a `BlockCommitment` containing a `withdrawalHash` that can prove this message.
3. Call `rollbackMessageWithProof()` with:
   - Required batch data (`_batchIndex`, `_commitmentBatch`).
   - Valid `_rollback_proof` and `_block_proof` that pass `_verifyWithdrawal()`. This returns the originally sent ETH to the attacker.
4. Call `receiveMessageWithProof()` using:
   - The same batch data and proofs from step 3.
   - Pass the `_rollback_proof` as `_withdrawal_proof`. The call succeeds because:
     - The message nonce matches `receivedNonce` (both are 5).
     - The proof verification passes since it uses the same `withdrawalHash` and same data in step 3.

This results in the attacker receiving the ETH twice - once from the rollback and once from the fake L2->L1 message.

Recommendation As a long term solution, it is advisable to use a different hash for rollback messages instead of using the `withdrawalHash`. An easier to implement solution in the short term will be to add a verification step in `receiveMessageWithProof()` that prevents rolled back messages from being received.

Developer Response The developers now enforce a validation step in `receiveMessageWithProof()` to ensure that the chain ID encoded in the incoming message data **does not match** the chain ID of the current receiving chain. This effectively prevents the replay of rolled-back messages using the `receiveMessageWithProof()` function.

5.1.5 V-FLNT-VUL-005: Incorrect rollup corruption check

Severity	High	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	_rollupCorrupted()		
Confirmed Fix At	1b2bf21		

The `Rollup` contract uses the `challengeQueue` array to track batch indexes that have been challenged. This array follows a first-in-first-out (FIFO) order, meaning older challenges are placed earlier in the queue and must be resolved before newer ones, as they have earlier deadlines.

To resolve a challenge, the contract implements the `proofBatch` function. After a batch is successfully verified, its corresponding entry in the `challengeQueue` array is reset. However, if other challenges are still active, the array is not shortened and it remains the same length, with only the verified batch's value being reset.

The `Rollup` contract considers itself corrupted if a challenge is not resolved within its deadline. This is determined by checking whether the `challengeQueue` array is non-empty and whether the deadline for the first element in the array is less than the current `block.number`. This logic is implemented in the `_rollupCorrupted()` function:

```
function _rollupCorrupted() internal view returns (bool) {
    return
          challengeQueue.length != 0 &&
          challengeDeadline[challengeQueue[0]] < block.number;
}
```

Snippet 5.4: Function `_rollupCorrupted()` from the `Rollup.sol` contract.

This implementation assumes that the first element in `challengeQueue` corresponds to an unresolved challenge. However, it does not account for the possibility that this challenge has already been resolved using the `proofBatch()` function. As a result, the deadline of a cleared entry (e.g., reset to zero) may be used incorrectly in the corruption check.

Impact The contract may return a corrupted state even when all active challenges are resolved. For example, consider the following `challengeQueue` array:

- `[batchN, batchM]`

If `batchN` is successfully verified, the array is modified to:

- `[0, batchM]`

The `_rollupCorrupted()` function will then evaluate the deadline for batch index `0` instead of `batchM`.

Recommendation Update the `_rollupCorrupted()` function to use the `challengeQueueStart` variable instead of the hardcoded index `0`. The `challengeQueueStart` variable represents the position of the oldest active challenge in the queue.

Developer Response The developers have implemented the suggested fix.

5.1.6 V-FLNT-VUL-006: Approval of invalid batches under the challenge period

Severity	High	**Commit**	fd208df
Type	Data Validation	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	_approvedBatch()		
Confirmed Fix At	844d3b9		

The `Rollup` contract is responsible for managing batches of block commitments, ensuring their validity, and handling challenges to these batches. The function `_approvedBatch()` checks if a batch is accepted, and if either the block number difference since acceptance exceeds `approveBlockCount` (the optimistic window) or the batch is marked as `proofedBatch` then the batch is considered to be approved.

However, the current implementation does not account for batches that are under an ongoing challenge or are descendants of such batches. This oversight could lead to a situation where a batch is incorrectly marked as approved, potentially allowing invalid transactions to be processed.

```
function _approvedBatch(uint256 _batchIndex) internal view returns (bool) {
   uint256 blockAcceptBlockNumber = acceptedBlock[_batchIndex];

   return
       _acceptedBatch(_batchIndex) &&
       (block.number - blockAcceptBlockNumber > approveBlockCount ||
       proofedBatch[_batchIndex]);
}
```

Snippet 5.5: Function `_approvedBatch()` from the `Rollup.sol` contract.

For example, consider the following scenario:

1. Batch #10, which is an invalid batch, is posted at block N and it will be optimistically accepted at block N + 10.
2. At block N + 9, the batch is challenged.
3. At block N + 10, the batch will be considered as approved even if the challenge has not been resolved.

Impact If a batch is incorrectly marked as approved while under an unresolved challenge and this batch were to be invalid, then it will break the integrity of the rollup system allowing invalid withdrawals to be processed in the L1 chain.

Recommendation To address this issue, the `_approvedBatch()` function should be modified to include checks for ongoing challenges. Specifically, it should verify if the batch or any of its ancestors are currently under challenge before marking it as approved.

Developer Response The developers now verify that for an approved batch, no block commitment within that batch has an active challenge. This is coupled with an overall design

change which was added as part of a separate commit, where instead of proving/challenging each batch index, they now prove or challenge each block commitment within a batch.

5.1.7 V-FLNT-VUL-007: Malicious message to Bridge address halts L1<>L2 communication

Severity	High	**Commit**	fd208df
Type	Denial of Service	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	sendMessage(), _receiveMessage(), _rollbackMessage()		
Confirmed Fix At	a2ee41f		

The Bridge contract implements a cross-chain messaging system between L1 and L2 chains, allowing users to send messages from one chain to another. The contract enforces strict sequential processing of messages through nonce validation, which means messages must be processed in the exact order they were sent. This check is implemented in both `receiveMessage()` and `receiveMessageWithProof()`:

```
1 if (_nonce != _takeNextReceivedNonce())
2    revert MessageReceivedOutOfOrder();
```

Snippet 5.6: Sequential nonce enforcement check. Snippet from the `receiveMessage()` and `receiveMessageWithProof()` function.

The root cause of the issue exists in the `_receiveMessage()` and `_rollbackMessage()` functions, which both include a revert statement when the the recipient of the message (the `to` parameter) is equal to the Bridge contract:

```
1 if (_to == address(this)) revert ForbiddenSelfCall();
```

Snippet 5.7: Code snippet from `_receiveMessage()` and `_rollbackMessage()`.

An attacker can exploit this by:

1. Sending a message from L1 to L2 (or vice versa) with the `_to` address set to the Bridge contract address of the destination chain.
2. When the destination chain tries to process this message, it will hit the `ForbiddenSelfCall()` revert.
3. This message can never be processed or rolled back due to the same check in both functions.
4. All subsequent messages are permanently blocked since the nonce sequence is stuck.

Impact This vulnerability allows an attacker to permanently block all cross-chain message passing between L1 and L2.

Recommendation To mitigate this issue, the `sendMessage()` function should prevent to send a message with the `to` parameter set as the Bridge contract address of the destination chain.

Developer Response The developers now verify during the `sendMessage()` function that the destination address does not correspond to the bridge address on the source chain, or the bridge address on the target chain.

5.1.8 V-FLNT-VUL-008: Return bomb attack can halt cross-chain messaging

Severity	High	**Commit**	fd208df
Type	Denial of Service	**Status**	Partially Fixed
File(s)	`Bridge.sol`		
Location(s)	_receiveMessage(), _rollbackMessage()		
Confirmed Fix At	6b28d8f		

The Bridge contract serves as a cross-chain messaging system that allows communication between the L1 and the L2 chains. It handles message passing in both directions:

1. L1 to L2 messaging via `sendMessage()` and `receiveMessage()`
2. L2 to L1 messaging via `sendMessage()` and `receiveMessageWithProof()`

The contract also includes a rollback mechanism for L1 to L2 messages through `rollbackMessageWithProof()` and `_rollbackMessage()`.

The core issue lies in the implementation of `_receiveMessage()` and `_rollbackMessage()` functions, which make external calls to arbitrary addresses without properly handling the potential for a return bomb attack:

```
1 (bool success, bytes memory data) = _to.call{value: _value}(_message);
```

Snippet 5.8: Code snippet from the `_receiveMessage()` function.

A return bomb attack occurs when a malicious contract returns an extremely large amount of data during a call, causing the transaction to fail by exceeding the block gas limit when the calling contract tries to copy this data to memory. This vulnerability is well-documented in the ExcessivelySafeCall library.

Impact The impact of this vulnerability is severe and affects both L1→L2 and L2→L1 messaging:

For L2→L1 messages:

- Messages must be processed in sequence (enforced by the nonce check), as explained in another issue: V-FLNT-VUL-007
- If a message triggers a return bomb, the transaction reverts and the nonce doesn't advance.
- All subsequent messages are permanently blocked.

For L1→L2 messages:

- While there is a rollback mechanism designed to handle stale messages, it suffers from the same vulnerability.
- If a message triggers a return bomb in `_receiveMessage()`, attempts to roll it back will also fail due to the same vulnerability in `_rollbackMessage()`.
- This prevents the dequeuing of the problematic message via the rollback mechanism.
- Consequently, `acceptNextBatch()` cannot proceed, halting the batch posting process.

Recommendation The ExcessivelySafeCall library mentioned in the above comments provides a solution for this exact problem. Replace the vulnerable calls with a pattern that limits the amount of returned data that is being copied.

Developer Response The developers implemented a custom version of `ExcessivelySafeCall`. However, the Veridise team believes that this implementation does not resolve the issue, as it still copies the full return data from the external call into `memory`.

Updated Developer Response The developers now ensure that only up to a defined maximum of 1024 bytes of return data is copied.

The Veridise team also recommends that the fix should also include a gas cap, as outlined in the `ExcessivelySafeCall` library.

5.1.9 V-FLNT-VUL-009: Deadlock scenario in rollup due to undeliverable deposits

Severity	High	**Commit**	fd208df
Type	Denial of Service	**Status**	Fixed
File(s)	`Rollup.sol, Bridge.sol`		
Location(s)	See description		
Confirmed Fix At	618e671		

The `Rollup` contract is responsible for managing batches of block commitments, verifying their validity, and handling challenges. As described in this issue, the contract can reach a **deadlock state** where no new batches can be accepted.

The `acceptNextBatch()` function allows the sequencer to post batches of `BlockCommitments`. When a `BlockCommitment`contains a non-zero `depositHash`, the function invokes `_checkDeposit()` to ensure that L1-to-L2 messages (deposits) were correctly processed. In `_checkDeposit()`, the system verifies the `depositHash` by popping messages from the Bridge queue, computing the hash of their IDs, and comparing it to the provided `depositHash`.

```
for (uint256 i = 0; i < depositInBlock.depositCount; ++i) {
     bytes32 depositId = Bridge(bridge).popSentMessage();
     depositIds[i] = depositId;
}

return keccak256(abi.encodePacked(depositIds)) == _commitmentBatch.depositHash;
```

Snippet 5.9: Code snippet from the `_checkDeposit()` function of the `Rollup.sol` contract.

To prevent censorship of L1-to-L2 messages, the `Rollup` contract tracks the last time a deposit was processed (`lastDepositAcceptedBlockNumber`). If there are deposits in the Bridge queue and none are processed within the `acceptDepositDeadline`, the contract **halts further batch submissions**. This ensures the sequencer cannot ignore pending deposits indefinitely.

The deadlock occurs due to interaction with `_receiveMessage()` in the `Bridge` contract. On the L2 side, this function delivers L1 messages. If a message exceeds the `receiveMessageDeadline`, the contract emits a `RollbackMessage` and exits without emitting `ReceivedMessage` (which is used to construct the `depositHash`):

```
if (_to == address(this)) revert ForbiddenSelfCall();
    if (
        receiveMessageDeadline != 0 &&
        _blockNumber + receiveMessageDeadline < block.number
    ) {
       emit RollbackMessage(_messageHash, block.number);
       return;
    }


    (bool success, bytes memory data) = _to.call{value: _value}(_message);

    receivedMessage[_messageHash] = success
        ? MessageStatus.Success
        : MessageStatus.Failed;

```

```
17 emit ReceivedMessage(_messageHash, success, data);
```

Snippet 5.10: Code snippet from the `_receiveMessage()` function of the `Bridge.sol` contract

Without the `ReceivedMessage` event, the sequencer **cannot construct a valid `BlockCommitment`** that satisfies `_checkDeposit()`, because the current head message in the queue was not delivered, and therefore it will remain in the Bridge queue.

While the system provides a `rollbackMessageWithProof()` function to clear the message from the queue, it requires a prior `BlockCommitment` containing the message in the`withdrawalHash`. But posting a batch is currently blocked due to the undeliverable deposit-creating a circular dependency.

Impact This creates a **deadlock** under the following conditions:

1. The sequencer must post a batch that includes a processed deposit (enforced by `acceptNextBatch()`).
2. The current head message in the Bridge queue cannot be delivered due to `receiveMessageDeadline` being exceeded which prevents the emission of the `RecievedMessage` event.
3. The message cannot be rolled back because rollback requires a batch containing a `withdrawalHash` to prove the rollback message against.
4. No new batches can be posted until the message is removed-returning us to point 1.

This scenario can be reached in cases of prolonged sequencer downtime or as consequence of bugs in the protocol such as the ones reported in:

- V-FLNT-VUL-007
- V-FLNT-VUL-008

Recommendation It is advisable to remove the rollover mechanism for the following reasons:

- The L1 to L2 messages deliveries are already enforced to be processed in order, for the sequencer to be able to keep posting batches.
- The deadline check is flawed as outlined in the issue: V-FLNT-VUL-010
- It is error-prone to implement which can lead to double spending issues as outlined in the issue: V-FLNT-VUL-004

Developer Response The developers now emit a `ReceivedMessage` event when receiving a message in the L2 even if the rollback mechanism is activated. This addresses the possible protocol deadlock condition number 2 mentioned above.

The developers also added a validation that when processing a rolled back message the chain id is the same as the chain that it is being processed on. This check was introduced to address another issue that was introduced when fixing the original issue which root cause was the removal of the following logic:

```
1 if (_messageHash != Queue.dequeue(sentMessageQueue))
```

Snippet 5.11: Code snippet from the `__rollbackMessage()` function of the `Bridge.sol` contract.

5.1.10 V-FLNT-VUL-010: Incompatible block number comparison breaks message delivery mechanism for L1 to L2 messages

Severity	High	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Bridge.sol`		
Location(s)	_receivemessage()		
Confirmed Fix At	5f840b7		

The Bridge contract implements a cross-chain messaging system between L1 and L2 networks, allowing messages to be sent in both directions. A component of this system is the message delivery deadline mechanism in the L2 side, which is intended to prevent the processing of stale messages and rolling them back to the L1 in case the delivery deadline is reached. In the `_receiveMessage()` function, the contract implements the following deadline check:

```
if (
    receiveMessageDeadline != 0 &&
    _blockNumber + receiveMessageDeadline < block.number
 ) {
    emit RollbackMessage(_messageHash, block.number);
    return;
}
```

Snippet 5.12: Code snippet from the `_receivemessage()` function.

This check is fundamentally flawed because it compares block numbers from different chains that operate on completely different scales and block production rates:

1. `_blockNumber` represents the L1 block number when the message was sent.
2. `block.number` in this context is the current L2 block number.
3. These values are incomparable due to vastly different scales and block production rates

For example, at the time of writing this report:

- Ethereum L1 last block is 22,540,642
- Optimism last block is 136,172,495

Impact The L2 block number will almost always be significantly higher than the L1 block number plus any reasonable deadline, causing the condition to evaluate to true for virtually all messages. This means that nearly all messages will be rolled back to L1 instead of being delivered as intended.

Recommendation The system should adopt an approach that doesn't rely on direct comparison of block numbers from the L1 and L2 chains.

Developer Response The developers now make use of a `blockDifference` variable which is used to keep track of the differences in block numbers between the L1 and the L2, and is updated by the owner. This block difference is also adjusted in the message deadline validation, to account for the differences in block numbers between the two chains when comparing them.

Veridise Response The Veridise team believes the issue remains unresolved. Specifically, relying on block numbers to determine timing between L1 and L2 is problematic, as the block production rates of the two chains are not synchronized. This makes any fixed block difference assumption unreliable.

We recommend exploring alternative mechanisms that utilize timestamps rather than block numbers to align timing logic more accurately across chains. Furthermore, we suggest thoroughly testing the proposed solution in a live or production-like environment to ensure correct behavior under real-world conditions.

Updated Developer Response The developers now make use of a block number oracle. They also intend to add a sequencer server that will update the actual l1 block number. This should allow making use of the actual l1 block number to compare block differences for the deadline as long as the owner is not malicious and the l1 block number is updated frequently.

5.1.11 V-FLNT-VUL-011: Missing decimals function override in the ERC20PeggedToken contract

Severity	High	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`ERC20PeggedToken.sol`		
Location(s)	_decimals		
Confirmed Fix At	d61871b		

The `ERC20PeggedToken` contract is designed to represent pegged versions of tokens, as a counterpart to the original tokens, which may be native to the L1 or the L2 chain. To maintain consistency in token properties such as name, symbol, and decimals these properties are communicated from the native chain of the original token and copied during the initialization of the pegged token, ensuring uniformity across chains.

However, while the contract stores the correct decimal value in the `_decimals` state variable during initialization, it fails to override the `decimals()` function inherited from the OpenZeppelin `ERC20` implementation. As a result, any call to `decimals()` will return the default value of 18, regardless of the actual decimals value stored in `_decimals`.

This discrepancy becomes particularly problematic for tokens with non-standard decimal places, such as USDC (6 decimals) or WBTC (8 decimals).

Impact While this issue doesn't directly risk token funds, it severely impacts the token's usability and could lead to significant user confusion and integration problems such as:

1. DeFi protocols and smart contracts commonly rely on the `decimals()` function to perform accurate token amount calculations.
2. Off-chain applications and interfaces use this value to display token amounts correctly.
3. Automated trading systems and oracles may use incorrect price calculations due to the decimal mismatch.

Recommendation Override the `decimals()` function in the `ERC20PeggedToken` contract to return the correct value stored in `_decimals`.

Developer Response The developers have implemented the suggested fix.

5.1.12 V-FLNT-VUL-012: Missing state updates in force revert batch function

Severity	Medium	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	forceRevertBatch()		
Confirmed Fix At	42eaa65		

The `Rollup` contract implements a mechanism to revert the rollup state to a previous batch through the `forcerRevertBatch()` function. This function is designed as a safety measure to handle situations where the rollup needs to be reverted to a previous valid state, typically in response to the rollup getting corrupted.

When a force revert occurs, the contract should properly clean up all state variables associated with the reverted batches to ensure the system can continue operating correctly. However, the current implementation of `forcerRevertBatch()` fails to update several important state variables such as `lastBlockHashAccepted` and several mappings that use batch indices as keys.

Impact

1. `lastBlockHashAccepted` - This variable tracks the last valid block hash and is essential for the sequencer to post new batches. Without updating this value during a force revert, the sequencer would be unable to continue operations after the revert.
2. Mappings that remain unchanged:
   a) `acceptedBatchHash` - This mapping stores the hash of posted batches. Failing to clear this mapping for reverted batches allows attackers to call `proofBatch()` again for previously valid batches that were reverted, even if the used index is smaller than `nextBatchIndex`. This effectively enables to approve invalid batches before they are posted using past data of valid batches that were reverted.
   b) `proofedBatch` - Similar to `acceptedBatchHash`, this mapping tracks which batches have been proven. Without proper cleanup, previously proven batches indices that were reverted could be considered valid without needing to call `proofBatch()` again even if the future batches occupying these indices are invalid.
   c) `challengeDeadline` - This mapping records the block number deadline for a challenged batch to resolve. If not cleared during a force revert, it could lead to confusion about whether a batch is under an ongoing challenge.

To illustrate the issue, consider the following scenario with the `proofedBatch` mapping:

1. The last valid batch has index `N` with a `blockHash` of `0x10`.
2. An invalid batch is posted at index `N+1` using the same `blockHash` of `0x10`.
3. A valid batch is posted at index `N+2` using `prevBlockHash` of `0x10` (building from the valid batch `N`).
4. The batch at index `N+2` is proven valid, and `proofedBatch[N+2]` is set to `true`.
5. The rollup is force reverted back to index `N`.
6. Now, new batches can be posted for indices `N+1` and `N+2`.
7. Critically, the batch at index `N+2` will still appear as proven (`proofedBatch[N+2] == true`) even if the newly posted batch is completely invalid.

Recommendation Update the `forcerRevertBatch()` function to properly clean up all the relevant state. The most critical updates needed are for `lastBlockHashAccepted`, `acceptedBatchHash`, `proofedBatch`, and `challengeDeadline` as these have direct security implications. While `acceptedBlock` doesn't have identified security impacts, it's advisable to also clean it for reverted indices to maintain state consistency.

Developer Response The developers changed the proof mechanism from a per batch proof verification to a per block proof verification. This introduced some changes into their `forcerRevertBatch()`, but all of the relevant state information is now correctly cleaned up.

5.1.13 V-FLNT-VUL-013: Challenger reimbursement flaw undermines the optimistic feature

Severity	Medium	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)		`Rollup.sol`	
Location(s)		proofBatch()	
Confirmed Fix At		54104bf	

The Rollup contract implements an optimistic rollup system where batches are accepted without immediate verification and can be challenged within a specific optimistic window after which the batch can be considered approved.

In the current implementation, when a batch is challenged, the challenger must deposit an amount of ETH (`challengeDepositAmount`) to prevent spam challenges. This deposit serves as a stake that should be:

1. Returned to the challenger if the challenge is valid (batch is invalid)
2. Slashed (taken from the challenger) if the challenge is invalid (batch is valid)

However, there's a flaw in the `proofBatch()` function. When a batch is successfully proven valid with, the challenger who incorrectly challenged the batch gets their deposit returned instead of having it slashed. This fundamentally breaks the economic incentives of the optimistic rollup design. An attacker could exploit this by:

1. Challenging every posted batch.
2. Forcing the rollup actors to provide proofs for all the challenged batches.
3. Receiving their challenge deposit back each time.

Impact By challenging all the posted batches with no financial risk, the attacker forces the rollup actors to provide proofs for all batches, negating the primary advantage of optimistic rollups.

Recommendation The challenge deposit should be slashed when a challenge is proven incorrect, with the funds transferred to the entity providing the proof.

Developer Response The developers have implemented the suggested fix.

5.1.14 V-FLNT-VUL-014: Counting of batch indices should start with index 1

Severity	Low	Commit	fd208df
Type	Logic Error	Status	Fixed
File(s)	Rollup.sol		
Location(s)	forceRevertBatch(), _cleanQueue()		
Confirmed Fix At	c4ae8b9		

The `Rollup` contract operates as an optimistic rollup, accepting new batches of data that can be challenged if deemed invalid. When a batch is challenged, it must be proved correct within a predefined challenge period. If the proof fails to materialize within this window, the rollup enters a corrupted state. To recover, the rollup is designed to revert to a previously known valid batch index using the `forceRevertBatch()` function.

Currently, batch indices are expected to start at `0`, since `nextBatchIndex` is left uninitialized and implicitly assumed to be `0`. However, this first batch cannot be forcefully reverted via `forceRevertBatch()`, as the function always reverts when called with index `0`. The snippet below demonstrates this limitation in `forceRevertBatch()`.

```
    function forceRevertBatch(
        uint256 _revertedBatchIndex
    ) external onlyOwner nonReentrant {
        if (!_acceptedBatch(_revertedBatchIndex)) {
            revert BatchNotAccepted(_revertedBatchIndex);
        }
        if (_revertedBatchIndex == 0) { // reverts if batch has index 0
            revert InvalidRevertIndex(_revertedBatchIndex);
        }
        /// Veridise: elided//
      }
```

Snippet 5.13: Snippet from function `forceRevertBatch()` in `Rollup.sol`

This becomes problematic if the proof for batch `0` is challenged and fails verification. In such a case, the rollup enters a corrupted state, and due to the inability to revert batch `0`, the system becomes permanently stuck. Since this affects only the first batch, the scope of damage in this case is somewhat limited, and it can be remedied by deploying a new instance of the rollup.

Additionally, there are inconsistencies in how index `0` is interpreted throughout the codebase. For instance, the `_cleanQueue()` function uses `0` to represent the absence of an active challenge at a given position. If the challenge queue contains `[1, 0]`, and the challenge for batch `1` is resolved, the function proceeds to delete the entire queue, under the assumption that `0` is a placeholder rather than an active challenge for batch `0`. This can be seen in the snippet below.

```
function _cleanQueue() internal {
while (
    challengeQueue.length != 0 &&
    challengeQueue[challengeQueueStart] == 0
) {
    ++challengeQueueStart;
    if (challengeQueueStart >= challengeQueue.length) {
        challengeQueueStart = 0;
```

```
9           delete challengeQueue;
10          return;
11      }
12  }
13  }
```

Snippet 5.14: Snippet from function `_cleanQueue()` in `Rollup.sol`

This behavior introduces the risk of unintentionally discarding a valid unresolved challenge, particularly one related to batch `0`. Therefore several problems can arise from having the batch indices begin with `0`.

Impact If the first batch is invalid and its proof fails, the system becomes permanently corrupted with no path to recovery, as batch `0` cannot be forcefully reverted. Additionally, the misinterpretation of index `0` in the challenge queue risks prematurely clearing valid challenges, which can lead to a compromise of correct dispute resolution.

Recommendation Start the batch indices from `1` instead of `0`. This will make the interpretation of index `0` consistent across the system, and ensure that all batches, including the first one, can be cleanly reverted if needed.

Developer Response The developers implemented the suggested fix.

5.1.15 V-FLNT-VUL-015: Free batch submission enables DoS and lacks challenge incentives

Severity	Low	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	acceptNextBatch()		
Confirmed Fix At	dab3543		

The Rollup contract implements a batch submission and challenge system where batches of L2 block commitments can be submitted and challenged. The system relies on challengers to identify and flag invalid batches by staking a `challengeDepositAmount` through the `challengeBatch()` function.

Two issues arise from the fact that batch submission through `acceptNextBatch()` has no associated bond or cost:

1. **Lack of Challenge Incentives**: When a challenger identifies an invalid batch, they must stake `challengeDepositAmount` to challenge it. However, even if the batch is proven invalid (by not being proven valid within the challenge period), the challenger only receives their deposit back without any additional reward. This creates a negative-sum game for challengers, who risk opportunity cost and gas fees with no upside, potentially leading to invalid batches going unchallenged.
2. **Owner DoS Vector**: The `forceRevertBatch()` function, which allows the owner to revert batches after the rollup gets corrupted, must iterate through all batches from the revert point to `nextBatchIndex`. A malicious sequencer can DoS this function by submitting a large number of batches at no cost, making the gas cost of reverting prohibitively expensive. This effectively prevents the owner from exercising their emergency controls.

Impact The impact is particularly severe because:

- Invalid state transitions could be accepted if challengers are not incentivized to monitor and challenge invalid batches.
- The owner loses the ability to perform emergency interventions if a malicious sequencer floods the system with invalid batches.

Recommendation Implement the following changes:

1. Require batch submitters to post a bond that:
   - Gets slashed and is partially awarded to successful challengers.
   - Gets returned once the batch is proven valid or after the optimistic window.
2. Allow the owner to pause the batch submission process.

Developer Response The developers only implemented the first recommendation by forcing the owner to pay incentives for challengers of invalid blocks. The second recommendation was implemented by introducing the ability for the owner to pause the batch submission process.

5.1.16 V-FLNT-VUL-016: Excess challenger deposits cannot be retrieved

Severity	Low	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	challengeBatch(), proofBatch()		
Confirmed Fix At	24b3fd6		

When initiating a challenge for a particular batch index, the challenger is required to deposit at least the `challengeDepositAmount`. However, the rollup contract does not enforce that the deposit matches this amount exactly-challengers can deposit more than the required value. In such cases, when the challenge is resolved, only the fixed `challengeDepositAmount` is marked for withdrawal. Any excess funds deposited beyond this amount are not tracked or made available for withdrawal.

Impact Challengers who deposit more than `challengeDepositAmount` risk having part of their funds permanently locked in the contract.

Recommendation Enforce that challenge deposits must be exactly equal to `challengeDepositAmount`. Additionally, consider declaring `challengeDepositAmount` as `immutable`, since it is intended to be assigned only once in the constructor and remains constant throughout the contract's lifecycle.

Developer Response The variable `challengeDepositAmount` has been declared as `immutable`, and the developers also validate that the challenger deposit is not greater than `challengeDepositAmount`.

5.1.17 V-FLNT-VUL-017: Incorrect update of tokenMapping may misclassify origin tokens as pegged tokens

Severity	Low	**Commit**	fd208df
Type	Logic Error	**Status**	Fixed
File(s)	`ERC20Gateway.sol`		
Location(s)	updateTokenMapping()		
Confirmed Fix At	7989a12		

The `tokenMapping` is intended to map pegged token addresses to their corresponding origin token addresses. This is for determining whether a token is native (origin) to the particular layer, or a pegged token which is pegged to a token originating from another layer.

However, in `updateTokenMapping()`, the mapping is updated incorrectly as `tokenMapping[_originToken] = _peggedToken`, reversing the intended direction of the mapping. This is inconsistent with the rest of the contract, where the mapping is used to resolve origin tokens from a given pegged token i.e. `tokenMapping[pegged] = origin`. See snippet below for context.

```
function updateTokenMapping(
  address _originToken,
  address _peggedToken
) external onlyOwner {
  require(_peggedToken != address(0), "token address cannot be 0");

  address _oldPeggedToken = tokenMapping[_originToken];
  tokenMapping[_originToken] = _peggedToken;

  emit UpdateTokenMapping(_originToken, _oldPeggedToken, _peggedToken);
}
```

Snippet 5.15: Snippet from function `updateTokenMapping()` in `ERC20Gateway.sol`

To illustrate the intended design of the token bridging flow: consider USDC as an origin token on L1. When bridging USDC to L2, the gateway checks that `tokenMapping[USDC]` is empty, confirming it is an origin token. It then deploys or references a pegged USDC on L2, and sets `tokenMapping[_peggedUSDC] = USDC`. This logic ensures the gateway correctly distinguishes between native and pegged tokens on each side.

If the mapping is mistakenly written as `tokenMapping[USDC] = _peggedUSDC`, it inverts this logic. The L1 will incorrectly treat USDC as a pegged token, causing it to attempt a transfer of the pegged version on L2 as if it were the native token-despite that token potentially not existing. As a result, any L1-to-L2 bridging attempts involving USDC will fail until the mapping is corrected.

Impact An incorrectly updated `tokenMapping` can cause persistent failure of bridge token transfers for affected tokens. Specifically, the origin token would be misidentified as a pegged token, leading the bridge to trigger invalid operations on the other side-such as attempting to transfer a non-existent pegged token. This will break the bridging functionality for the token in question, till it is observed and addressed by updating the token mapping correctly.

Recommendation Ensure consistency in how `tokenMapping` is used and updated throughout the contract. In `updateTokenMapping()`, it should follow the pattern `tokenMapping[_peggedToken] = _originToken`.

Developer Response The developers have made the suggested changes and now follow the correct pattern when updating the token mapping.

5.1.18 V-FLNT-VUL-018: Use of Solidity transfer is discouraged for gas reasons

Severity	Warning	**Commit**	fd208df
Type	Denial of Service	**Status**	Fixed
File(s)	`Rollup.sol`		
Location(s)	withdrawChallengeDeposit()		
Confirmed Fix At	ca80dc8		

The `withdrawChallengeDeposit()` function uses Solidity's `transfer()` to send ETH to challengers. The `transfer()` function is hardcoded to forward only 2,300 gas, which is enough for basic ETH transfers but can fail when the recipient is a smart contract that performs additional operations in its receive/fallback function.

Impact If a challenger is a smart contract that requires more than 2,300 gas to receive ETH, their withdrawal will revert and they will be unable to retrieve their challenge deposit. This could effectively lock their funds in the contract permanently.

Recommendation Replace `transfer()` with a low-level `call()` while still maintaining reentrancy protection.

Developer Response The developers implemented the suggested fix.

5.1.19 V-FLNT-VUL-019: Solidity best practices

Severity	Warning	**Commit**	fd208df
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	0c881f9, dfa513b, 14232c3, 1a12cc2, 2835ecf		

Consider implementing the following Solidity best practices:

1. **Two-phase ownership transfers.** The following contracts are `Ownable`, but transfer ownership in a single step. Using `Ownable2Step` will ensure ownership is not transferred to an account which is unable to accept ownership:
   a) `ERC20Gateway`
   b) `ERC20TokenFactory`
   c) `Rollup`
2. **Check result of token transfers or make use of safe transfer functions.** The following locations perform transfers on `ERC20` tokens, but do not check that the transfer returned `true`. Since the `ERC20Gateway` is designed to interact with a wide variety of ERC20 tokens-including those that do not strictly adhere to the ERC20 standard-it is strongly recommended to use OpenZeppelin's `SafeERC20` library. This library provides safe wrappers for token operations such as `transfer`, `transferFrom`, and `approve` that handle non-standard behavior gracefully and reverting on failure.
   a) `ERC20Gateway._receiveNativeTokens()`. Makes use of `ERC20.transfer()` to transfer tokens.
   b) `ERC20Gateway.sendTokensFrom()`. Makes use of `ERC20.transferFrom()` to transfer tokens from the sender to the gateway.
3. **Appropriate variable names.** The following location defines a variable name which can be better described with a more appropriate name, grammatically or contextually. ****
   a) `Rollup.proofedBatch` can be better described as `Rollup.provenBatch`.
4. **Missing Length Checks.** Some functions take as input two arrays which are expected to be of a comparable length, but do not perform verifications to ensure the same.
   a) `Rollup.acceptNextBatch()`. This function takes in as input two arrays, `_commitmentBatch` and `depositsInBlocks`. The length of `depositsInBlocks` is always expected to be less than or equal to `_commitmentBatch`, since not each block will have deposits, but this is not explicitly validated.
5. **Missing TODOs.** Some functions have remaining TODOs which should be implemented.
   a) `Rollup.acceptNextBatch()`
6. **Missing input validation on state updates.** Some functions perform important state updates but do not validate the provided inputs to be non-zero. This may result in incorrect or irrecoverable configurations for the contracts, which can greatly affect the smooth functioning of the protocol.
   a) `Rollup.updateVerifier()`. This function does not validate the new verifier address to be non-zero.

b) `Rollup.constructor()`. The constructor of the rollup initializes a lot of important state variables. All of these should be validated to be non-zero.

Impact Not following best practices may lead to projects with reduced "by default" security/usability, allowing simple errors to magnify into large mistakes.

Recommendation Follow the above Solidity best practices.

Developer Response The developers have implemented the suggested changes.

5.1.20 V-FLNT-VUL-020: Missing events

Severity	Warning	**Commit**	fd208df
Type	Missing/Incorrect Events	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	300a5a9		

Upon a state update, it is recommended that developers emit an event to indicate that a change was made. Doing so allows both external users and protocol administrators to monitor the protocol for a variety of reasons, including for potentially suspicious activity. It is therefore a good practice for significant changes to the protocol to be accompanied with events to enable this monitoring. The following functions make state updates but do not emit events recording the same.

1. `ERC20Gateway.setOtherSide()` updates `otherSide`, `otherSideTokenImplementation` and `otherSideFactory` which are the gateway, token implementation and the token factory addresses on the corresponding layer.
2. `Rollup.withdrawChallengeDeposit()` can be used by a challenger to withdraw their challenged deposit. But the function does not emit an event on challenger withdrawal.
3. `Rollup.setDaCheck()` can be used to toggle the data availability check within the rollup contract.
4. `Rollup.setBridge()` updates the bridge contract address in the rollup.

Impact If important state updates are made without administrators noticing, it can be seriously damaging to the protocol. For instance, in the `Rollup` if the bridge address is updated without the admins knowledge, then the `owner` address for the contract has been compromised. The attacker can forge the bridge message queue information that the rollup views when accepting new batches, without the owner being aware of it.

Recommendation Emit events to indicate when state changes occur in the functions listed above.

Developer Response The developers have added events for the mentioned state updates.

5.1.21 V-FLNT-VUL-021: Fetched gateway address labeled incorrectly

Severity	Warning	**Commit**	fd208df
Type	Maintainability	**Status**	Fixed
File(s)	ERC20Gateway.sol		
Location(s)	sendTokensFrom()		
Confirmed Fix At	ed49682		

In the function `sendTokensFrom()`, the gateway address and the origin token address are fetched in by calling `getOrigin()`. But, the current code mistakenly refers to the gateway address returned as the `originGateway`. This discrepancy becomes apparent when reviewing the `_deployL2Token()` function, where the pegged token is deployed using the current gateway's address. The following snippet illustrates the incorrect reference to the gateway address.

```
function sendTokensFrom(
  address _token,
  address _sender,
  address _from,
  address _to,
  uint256 _amount,
  uint256 _value
) internal {
  bytes memory _message;

  if (tokenMapping[_token] == address(0)) {
        // Veridise: elided///
  } else {
      (address originGateway, address originAddress) = ERC20PeggedToken(
          _token
      ).getOrigin();
      require(tokenMapping[_token] == originAddress);

      ERC20PeggedToken(_token).burn(_from, _amount);

      _message = abi.encodeCall(
          ERC20Gateway.receiveNativeTokens,
          (originAddress, _sender, _to, _amount)
      );
  }

  IBridge(bridgeContract).sendMessage{value: _value}(otherSide, _message);
}
```

Snippet 5.16: Snippet from the function `sendTokensFrom()` in `ERC20Gateway.sol`

Currently this does not lead to a serious issue, because the value of the returned `originGateway` is unused, and therefore it is not misused. If left unaddressed, it may be mistakenly used in the future under the false assumption that it represents the `originGateway`, potentially leading to severe issues.

Impact While this does not currently cause a direct issue, misuse of the gateway address

returned by `getOrigin()`, under the assumption that it represents the `originGateway`, could lead to serious problems in the token bridging logic.

Recommendation Correctly refer to the gateway address returned by `getOrigin()` as the `gateway` address and not the `originGateway`. Additionally, since the returned value is currently unused within `sendTokensFrom()`, consider removing it or clarify the purpose of fetching the gateway address in this context.

Developer Response The developers have removed the incorrect label, and do not cache the gateway value returned since it is unused in the function flow.

5.1.22 V-FLNT-VUL-022: Unused code

Severity	Info	**Commit**	fd208df
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	7aa1f87		

Description The following program constructs are unused:

1. `contracts/rollup/Rollup.sol`:
   a) `Interface IRollupVerifier`: This interface is unused in this contract.
   b) `error EthTransferFailed`: This error is unused.
2. `contracts/ERC20Gateway.sol`
   a) `address gatewayAuthority`: This address is unused.
3. `contracts/interfaces/IERC20Gateway.sol`
   a) `Interface IERC20`: This interface is unused in this contract.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers have removed the unused constructs.