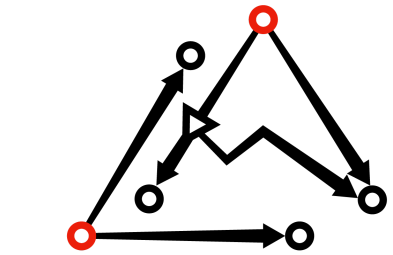

Least Authority

PRIVACY MATTERS

Ethrex

Security Audit Report

LambdaClass

Final Audit Report: 4 May 2026

Table of Contents

Overview

Background

LambdaClass has requested Least Authority perform a security audit of Ethrex L2, the L2 mode of Ethrex, an Ethereum Rust Execution L1 and L2 client.

Project Dates

- **January 12, 2026 - January 30, 2026:** Initial Code Review *(Completed)*
- **February 3, 2026:** Delivery of Initial Audit Report *(Completed)*
- **May 4, 2026:** Verification Review *(Completed)*
- **May 4, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Nikos Iliakis, Security Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Will Sklenars, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Ethrex L2 followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Main repo:
  https://github.com/lambdaclass/ethrex

- First Priority:
  - The prover and sequencer:
    - https://github.com/lambdaclass/ethrex/tree/main/crates/l2/prover
    - https://github.com/lambdaclass/ethrex/tree/main/crates/l2/sequencer

- Smart contracts:
  https://github.com/lambdaclass/ethrex/tree/main/crates/l2/contracts/src
  - Excluding:
    https://github.com/lambdaclass/ethrex/tree/main/crates/l2/contracts/src/l1/based
  - Including the dependency:
    https://github.com/lambdaclass/ethrex/tree/main/crates/vm

- Other related crates :
  - Second Priority:
    - https://github.com/lambdaclass/ethrex/tree/main/crates/networking/rpc
    - https://github.com/lambdaclass/ethrex/tree/main/crates/common/crypto

- Third Priority:
    - Remaining sections of the repo (except exclusions)

Specifically, we examined the Git revision for our initial review:

- `e88175e2d49f1192cc9f2fdeae6fde1392d0759d`

For the verification, we examined the following Git revision:

- `c63829a1bde522beb5f5e00904d979114f5b951f`

For the review, this repository was cloned for use during the audit and for reference in this report:

- `lambdaclass-ethrex`: https://github.com/LeastAuthority/lambdaclass-ethrex

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website: https://lambdaclass.com
- Audit details shared via HackMD on April 15, 2025: https://hackmd.io/@pmhS3JPZRjy0AJ06Ycq_vQ/S1FmYTqA1g
- Documentation for Ethrex L2 in the `/docs` directory

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Ethrex is a Rust implementation of the Ethereum protocol, designed to support ZK proofs and L2 execution. It includes a client with support for two different modes, L1 and L2, with the former working as a regular Ethereum client and the latter as a ZK-rollup prover with support for several different backends (`SP1`, `RISC0`, `OpenVM`, `ZisK`, and `TEEs`).

Smart Contracts & VM

The Ethrex client features a custom implementation of the Ethereum Virtual Machine, a serially executing stack machine, called Lambda EVM (LEVM) designed for the ZK-optimized data structures and stateless execution used in Ethrex. Our team reviewed this implementation for issues within the system and found several areas where atomicity guarantees are limited and certain types of failure could result in inconsistent state (Issue L, Issue B, and Issue C).

We also noted one particular case of potentially insufficient locking where a value is read both at the beginning and at the end of a transaction. Due to two distinct reads, we inferred that certain transactions could potentially lead to the two reads returning different values. We suggest that the value be cached rather than read multiple times (Issue Q).

Ethrex also supports `Aligned-Mode`, where proof aggregation and verification are outsourced to the Aligned Layer service. We found two inconsistencies relating to Aligned mode (Issue J and Issue K). We also noted that outages in Aligned Layer can lead to a breakdown of eventual consistency, and that Aligned Layer outages should be expected to occur on occasion (Issue A).

At the contract level, security considerations are evident in multiple contract-level controls, including `onlySelf/onlyBridge` access control, Pausable and `ReentrancyGuard` on L1 functions, `Ownable2Step` and UUPS upgradeability, Merkle proof verification for withdrawals, `claimedWithdrawalIDs` to prevent replays, deposits provenance tracking to prevent token mismatches, and address aliasing in `CommonBridge` to mitigate L1 contract impersonation.

We reviewed Ethrex's core bridge system, which is intended to move value and messages safely between L1 and L2. Users deposit on L1, the L1 bridge records the deposit, and a privileged L2 transaction mints funds on L2. For withdrawals, users burn on L2, the L2 bridge emits a message, and once the batch is verified on L1, the user can claim funds with a Merkle proof. The `OnChainProposer` coordinates these components by committing and verifying batches and publishing withdrawal roots, while the L2 Messenger is the simple event emitter that exposes messages to L1. Overall, the system is designed to support fast bridging on L2 while keeping finality and custody anchored on L1.

Prover

The prover folder includes the implementation of the prover component of the Ethrex client, which uses ZK-rollups to batch blocks of L2 transactions into a cryptographic proof, supporting multiple proving backends (for example, `SP1`, `RISC0`, `OpenVM`, and `ZisK`).

We reviewed the different zkVM backend implementations and found that the `ZisK` backend is vulnerable to a DoS when reading the compiled `ZisK` zkVM guest binary from the filesystem (Issue O). We also identified that both the `ZisK` and `OpenVM` entrypoints fail to perform domain separation when hashing the zkVM output, leading to potential replay attacks (Issue I).

Our team further observed that the guest program implementation does not check whether the `transactions` and `withdrawals` roots in the block header match the block body, enabling an attacker to submit a block where the body does not match the header (Issue G).

Cryptographic Sub-Components

We also reviewed the `Blake2` and `Keccak` Rust and assembly implementations for `x86_64` and `ARM v8`, as well as a KZG implementation in the `common/crypto` crate. The `Keccak` implementations are based on the cryptogams project and are adapted for LambdaClass. We identified two issues: one potential nearby-memory oracle attack within the `Blake2` implementation (Issue R) and a potential buffer overflow write within the `Keccak` implementation (Issue S). In addition, we recommend improvements for the `Blake2` implementation (Suggestion 8).

Dependencies

We examined all dependencies implemented in the codebase. For the smart contracts, well-known and audited `OpenZeppelin` libraries are used. However, dependency-driven risks remain in other components of the system. When Aligned Mode is enabled, the Ethrex system depends on Aligned Layer for proof aggregation and verification, and Aligned Layer outages can affect rollup performance. We also identified several vulnerable crates (Issue T).

Code Quality

We performed a manual review of the repositories in scope and found the code to be well organized and straightforward to navigate. The contracts are structured with clear L1/L2 separation, interfaces, and NatSpec comments, and they follow standard upgradeable and access-control patterns. Some deprecated state variables remain for compatibility but do not affect readability.

In addition, the VM code was well written and clear to reason about. To avoid potential future inconsistencies, we recommend caution when setting accumulators (Suggestion 3).

Tests

Our team found the test coverage of the repositories in scope to be insufficient. We recommend adding more tests (Suggestion 6).

Documentation and Code Comments

The documentation is detailed and describes the system clearly, including diagrams. Similar documentation is provided for the prover component. In addition, the smart contracts and interfaces include useful NatSpec and inline comments for critical behaviors. However, inline documentation in the prover code is limited, with few comments present (Suggestion 7).

Scope

The scope of this review was sufficient and included all security-critical components.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Aligned Layer Service Interruptions Can Result in Denial of Service	High	Resolved
Issue B: Intrinsic Gas Error Can Result in Loss of Funds	High	Resolved
Issue C: Nonatomic Finalization Can Lead to Inconsistent State	High	Resolved
Issue D: ERC-20 AssetDiffs Omitted in BalanceDiff Aggregation	High	Resolved
Issue E: Permissionless Privileged Message Parameters Can Stall Verification via Expired Privileged Queue Entries	High	Resolved
Issue F: Underflow and Off-by-One in regenerate_state Target Handling	High	Resolved
Issue G: Stateless L1 Validation Omits Transactions Root Check	High	Resolved
Issue H: Incomplete Gas Used Validation Allows Nonzero Gas in Empty Blocks	High	Resolved
Issue I: Missing Domain Separation on Guest Output Digest Enables Cross-Context Replay	High	Determined Non-Issue
Issue J: Privileged Transaction Failure Path Can Be Bypassed	Medium	Resolved
Issue K: Privileged Transaction Inclusion Not Guaranteed in ALIGNED_MODE	Medium	Resolved
Issue L: Fee-Token Fees Can Be Locked for Transactions That Fail Validation	Medium	Resolved
Issue M: Unvalidated Proof Persistence Halts Liveness	Medium	Determined Non-Issue
Issue N: L1 Watcher Cursor Advances Before Processing, Dropping Privileged Logs	Medium	Resolved
Issue O: Unbounded Read and Nonatomic Write in write_elf_file Enables Local DoS	Medium	Resolved
Issue P: Block Execution Pre-Rejects Transactions Based on Declared Gas Limit	Medium	Determined Non-Issue
Issue Q: Fee-Token Ratio Is Fetched in Both Prepare and Finalize, Allowing Inconsistent Lock vs. Settlement if the Ratio Changes During Transaction Execution	Low	Resolved
Issue R: Unsanitized Boolean Leads to Nearby-Memory Blake2b Oracle	Low	Resolved
Issue S: Zero Length Leads to Buffer Overflow Write in SHA-3 Squeeze	Low	Resolved

Issue T: Usage of Vulnerable Dependencies	Undetermined	Resolved
Suggestion 1: Verify That Tokens Match in WithdrawERC20	Informational	Resolved
Suggestion 2: Add Router-Only Access Control to receiveETHFromSharedBridge	Informational	Resolved
Suggestion 3: Accumulate the Substate Property Instead of Overwriting It	Informational	Resolved
Suggestion 4: Remove Deprecated “State Diffs” Feature	Informational	Resolved
Suggestion 5: Improve Prover Code Quality	Informational	Partially Resolved
Suggestion 6: Improve Test Coverage	Informational	Partially Resolved
Suggestion 7: Improve Code Comments	Informational	Partially Resolved
Suggestion 8: Improve BlakeE2b Implementation	Informational	Resolved

Issue A: Aligned Layer Service Interruptions Can Result in Denial of Service

Location
`l2/sequencer/l1_proof_sender.rs#L174`

`l2/sequencer/l1_proof_verifier.rs#L145`

Synopsis
When the L2 runs in `aligned-mode`, proof verification depends on an external aggregation pipeline. If Aligned accepts a proof but fails to aggregate it (or Aligned becomes unresponsive), `verifyBatchesAligned()` cannot advance, stalling L1 verification for all subsequent batches. Without a clear operator recovery path, prolonged finality outages can occur.

Impact
High.

Proof verification can be blocked indefinitely, preventing withdrawal finalization and cross-chain message settlement until operators intervene.

Feasibility
Medium.

Aligned Layer outages or partial failures in the aggregation pipeline can trigger the stall.

Severity
High.

Preconditions

For this issue to occur, the following must hold true:

- `OnChainProposer.sol` must be initialized with `ALIGNED_MODE = true`.
- Proofs must be routed through Aligned's batcher or aggregator.
- A submitted proof must be lost or never aggregated, or Aligned must be unreachable for an extended period.

Technical Details

When a proof is successfully submitted to Aligned Layer, the sequencer advances its local "last sent" cursor via `set_latest_sent_batch_proof` in `l1_proof_sender.rs`. On each `send` loop, it chooses the next batch using `get_latest_sent_batch_proof` and `lastVerifiedBatch`, such that a proof that was accepted but later lost by Aligned will not be resent automatically. The verifier (`l1_proof_verifier.rs`) only polls Aligned for aggregation and does not trigger resubmission. Accordingly, a missing aggregated proof blocks `verifyBatchesAligned` from advancing.

Remediation

We recommend implementing one or more of the following remediations:

- Maintain an operator runbook to facilitate fast recovery from Aligned Layer outages.
- Monitor for "not aggregated" proofs beyond a defined threshold and `alert` operators.
- Add an automated resubmission policy when aggregation is not observed after a configurable timeout.

Status

The LambdaClass team has introduced a dual-cursor architecture, which independently tracks dispatch to Aligned Layer and confirmation on-chain. This supports proof resubmission if on-chain verification does not advance within a configurable window. Additionally, operator documentation has been further developed to provide guidance on recovery procedures.

Verification

Resolved.

Issue B: Intrinsic Gas Error Can Result in Loss of Funds

Location

vm/levm/src/hooks/l2_hook.rs#L337

Synopsis

For privileged transactions where the sender is not `COMMON_BRIDGE_L2_ADDRESS`, the sender's balance can be debited for `msg_value` before all failure-triggering validation steps have completed. If `add_intrinsic_gas()` fails, the transaction is forced to fail via `tx_should_fail`, and `msg_value` is zeroed. Since the refund path (`undo_value_transfer`) depends on `vm.current_call_frame.msg_value`, the original value is no longer available to refund, and the sender's balance reduction is not reversed. This can result in permanent loss of `funds` for the transaction sender.

Impact

High.

A privileged transaction sender can permanently lose funds.

Feasibility
Medium.

If an actor or operator workflow is able to submit a non-bridge privileged transaction with nonzero `msg_value`, triggering the intrinsic gas failure condition is straightforward (for example, by setting the gas limit below the intrinsic gas requirements).

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The transaction must be treated as privileged by the VM.
- The sender or origin must not be `COMMON_BRIDGE_L2_ADDRESS`.
- The transaction must not have a nonzero `msg_value`.
- `add_intrinsic_gas()` must fail, causing `tx_should_fail` to be set.

Technical Details
The privileged execution preparation path debits the sender's balance for the transaction value.

```
vm.decrease_account_balance(sender_address, value)
```

However, this debit occurs before all checks that can force a failure have completed.

```
if vm.add_intrinsic_gas().is_err() {

    tx_should_fail = true;

}
```

The failure path results in `msg_value` being zeroed.

```
vm.current_call_frame.msg_value = U256::zero();
```

However, the refund path depends on `msg_value` to roll back this value transfer.

```
vm.decrease_account_balance(vm.current_call_frame.to,

vm.current_call_frame.msg_value)?;

vm.increase_account_balance(vm.env.origin, vm.current_call_frame.msg_value)?;
```

If `msg_value` is zeroed before the refund path runs, the refund becomes a no-op, even though the sender's balance was previously decreased. This can burn the sender's funds in the failure case.

Mitigation
Operators can reduce risk by:

- Avoiding non-bridge privileged transactions with nonzero `msg_value`, and/or
- Ensuring privileged transactions always satisfy intrinsic gas constraints.

Remediation
We recommend reordering the code so that `tx_should_fail` decision points occur before funds are debited.

Status
The LambdaClass team has deferred the sender balance debit for non-bridge privileged transactions until after all validation checks pass, so an intrinsic gas failure no longer leaves the sender debited with a zeroed refund value. A regression test confirms the sender's balance is fully preserved on revert.

Verification
Resolved.

Issue C: Nonatomic Finalization Can Lead to Inconsistent State

Location
vm/levm/src/hooks/l2_hook.rs#L96

Synopsis
`finalize_non_privileged_execution` applies multiple state mutations in sequence and propagates errors. If an earlier mutation succeeds and a later step returns an error, the function returns early without rolling back the earlier mutations. This breaks the atomicity of transaction finalization and can leave the `VM/database` in a partially-updated state even though execution did not successfully complete.

Impact
Medium.

If the surrounding execution environment treats a finalize-time `VMError` as a transaction failure but continues using the mutated `VM/database` state (for subsequent transactions in a block, for example), an attacker can potentially cause partial fee, refund, or `selfdestruct` effects to persist without the full intended set of finalization steps being applied. This can lead to inconsistent state transitions and, depending on which mutations persist, may enable underpayment of fees or unintended balance or storage changes.

Feasibility
High.

The L2 sequencer and L1 blockchain builders use a shared EVM instance across all transactions in a block, so any inconsistent state is built upon by subsequent transactions in the block.

Severity
High.

Technical Details
In `hooks/l2_hook.rs`, `finalize_non_privileged_execution` performs a series of mutations after execution completion:

1. `default_hook::delete_self_destruct_accounts(vm)?` (line 134) mutates account state (marks destroyed).
2. `get_fee_token_ratio(vm, fee_token)?` and a U256 to U64 conversion (lines 136-146) can error.
3. `pay_to_l1_fee_vault(...)` (lines 148-155) may mutate balances or fee-token storage.

4. `refund_sender_fee_token(...)` or `default_hook::refund_sender(...)` (lines 157-161) mutates balances or fee-token storage.
5. `pay_coinbase_l2(...)` (lines 163-168) mutates balances or fee-token storage.
6. `pay_base_fee_vault(...)` (lines 174-188) mutates balances or fee-token storage.
7. `pay_operator_fee(...)` (lines 190-197) mutates balances or fee-token storage.

Any error in steps 2-7 returns early, after some prior mutations may already have been applied.

Remediation
We recommend wrapping mutations within `finalize_non_privileged_execution` in a transactional guard so that all mutations are committed or rolled back atomically.

Status
The LambdaClass team has split `Finalize` into a fallible-computation phase followed by an atomic mutation phase that reverts all partial state changes on error.

Verification
Resolved.

Issue D: ERC-20 AssetDiffs Omitted in BalanceDiff Aggregation

Location
[l2/common/src/messages.rs#L169]

Synopsis
ERC-20 mint diffs are decoded in the function `get_balance_diffs` but never inserted into the per-chain `value_per_token` vector, resulting in empty `assetDiffs` even when ERC-20 messages exist.

Impact
Medium.

ERC-20 cross-chain mints may not be forwarded to the shared router, causing stuck bridging and inconsistent accounting while `message_hashes` advance.

Feasibility
High.

No privileges are required. The condition occurs whenever L2 messages include `crosschainMintERC20` and the committer derives balance diffs.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- `crosschainMintERC20` messages must be present in `L2Message.data`.
- The committer pipeline must use the `get_balance_diffs` function.

Technical Details
The function `get_balance_diffs` detects an ERC-20 mint when the first four bytes of the variable `message.data` match `CROSSCHAIN_MINT_ERC20_SELECTOR`, then decodes `token_l1`,

`token_src_l2`, `token_dst_l2`, and value into the variable `value_per_token_decoded`. The per-chain entry is created with an empty `value_per_token` vector, and the subsequent loop only updates an existing element but never inserts `value_per_token_decoded` when no match exists, leaving `entry.value_per_token` empty.

As a result, batches committed to L1 carry `BalanceDiff` with `message_hashes` including `get_l2_message_hash(message)` but with `value_per_token` empty. Therefore, the contract function `publishL2Messages` injects hashes without calling `sendERC20Message`, preventing ERC-20 transfers while progressing message queues.

Remediation

We recommend inserting the decoded `AssetDiff` into the vector when no match is found by pushing `value_per_token_decoded` to `entry.value_per_token` after the aggregation loop or by aggregating via a map keyed by (`token_l1`, `token_src_l2`, `token_dst_l2`).

Status

The LambdaClass team has resolved the issue by pushing the decoded `AssetDiff` into the vector when no match is found.

Verification

Resolved.

Issue E: Permissionless Privileged Message Parameters Can Stall Verification via Expired Privileged Queue Entries

Location

`l2/sequencer/l1_watcher.rs#L319`

Synopsis

A permissionless L1 entrypoint propagates user-controlled gas and calldata into the privileged queue, which can create non-includable privileged items that later expire and block verification.

Impact

High.

An unprivileged actor can cause `CommonBridge.hasExpiredPrivilegedTransactions()` to return `true` persistently, which can prevent verification of batches and effectively enable censorship of non-privileged transactions.

Feasibility

Medium.

An attacker with no onchain privileges can call the function `ICommonBridge.sendToL2` and select `SendValues.gasLimit` and `SendValues.data` values that exceed L2 inclusion constraints or cause repeated privileged admission failures.

Severity

High.

Preconditions

For this issue to occur, the following must hold true:

- The function `ICommonBridge.sendToL2` must be permissionless and callable when the contract is not paused.
- The L2 pipeline must reconstruct privileged transactions using user-controlled `gasLimit` and data from `PrivilegedTxSent`.
- Verification logic must gate progress on the function `CommonBridge.hasExpiredPrivilegedTransactions()`.
- The configured L2 constraints must be able to render some privileged transactions non-includable, for example, due to block gas limit or calldata size bounds.

Technical Details

The function `ICommonBridge.sendToL2` accepts `SendValues.gasLimit` and arbitrary-length `SendValues.data`, and the function `CommonBridge._sendToL2` emits `PrivilegedTxSent` with these fields while also pushing the derived hash into `pendingTxHashes` and setting `privilegedTxDeadline[hash]`. On the watcher side, the function `L1Watcher.process_l2_transactions` reconstructs a `PrivilegedL2Transaction` using `gas_limit: tx.gas_limit.as_u64()` and `data: tx.data.clone()`, treating these values as authoritative for L2 inclusion.

If an attacker chooses a `gasLimit` or data length that violates L2 inclusion limits or causes repeated rejection by privileged selection logic, the corresponding privileged hash can remain at the head of the pending queue until `privilegedTxDeadline` elapses. After the deadline, the function `CommonBridge.hasExpiredPrivilegedTransactions()` returns `true` when `pendingTxHashesLength() != 0` and the head entry is expired, which can block verification of subsequent batches that do not include the privileged item.

Remediation

We recommend bounding and validating `SendValues.gasLimit` and `SendValues.data` in the function `ICommonBridge.sendToL2` against the L2 inclusion constraints. Accordingly, parameters that would produce non-includable privileged transactions should be rejected or normalized.

Status

The LambdaClass team has introduced `l2GasLimit` within the `CommonBridge` contract, enabling privileged transactions with excessive gas limits to be rejected at the contract level.

Verification

Resolved.

Issue F: Underflow and Off-by-One in regenerate_state Target Handling

Location

`l2/sequencer/l1_committer.rs#L284`

Synopsis

Target handling in state regeneration uses a predecessor value that can underflow and produces checkpoints one block behind the requested target.

Impact

Medium.

Exploitation can cause a debug panic or a wrapped replay range and produce an off-by-one checkpoint, potentially delaying batch preparation or commitment.

Feasibility
High.

The vulnerable branch is executed during checkpoint rebuild with `Some(target)` and requires no special privileges beyond triggering that path.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The function `regenerate_state` must be called with `Some(target_block_number)`.
- The variable `target_block_number` must equal `0` for the underflow crash.
- Release builds must use wrapping semantics, and debug builds must panic on integer underflow.
- Any nonzero `target_block_number` must be sufficient for the off-by-one behavior.

Technical Details
The function `regenerate_state` rewrites the variable `target_block_number` using `target_block_number - 1` when `Some(target_block_number)` is provided, which underflows for `0` before any guard executes and always shifts the replay range by one block. The subsequent check if `target_block_number == 0 { return Ok(()); }` executes after the subtraction and therefore does not prevent the underflow case.

In the checkpoint rebuild path, the function `ensure_checkpoint_for_committed_batch` invokes the function `regenerate_state` with `Some(batch.last_block)`. If `last_block` equals `0`, the subtraction panics in debug or wraps to `u64::MAX` in release, leading to failures when fetching headers and halting checkpoint creation. With any nonzero target, the off-by-one replays up to target `- 1`, producing a checkpoint one block behind the caller's expectation and causing downstream state divergence or stalls.

Remediation
We recommend replacing the subtraction on the variable `target_block_number` in the function `regenerate_state` with an early return when it equals `0`. The function should use the provided target directly without decrementing, or use `checked_sub` only where a predecessor is required.

Status
The LambdaClass team has resolved the issue by using a match expression that returns `Ok(())` early, preventing the underflow.

Verification
Resolved.

Issue G: Stateless L1 Validation Omits Transactions Root Check

Location
`l2/prover/src/guest_program/src/execution.rs#L123`

Synopsis
The prover validates L1 block headers without verifying that the header's `transactions` root commits to the provided block body.

Impact
High.

A malicious prover input could produce a proof for an invalid Ethereum block header that does not commit to the executed transaction list, which could cause downstream consumers that rely on the proof to accept an invalid block transition.

Feasibility
Medium.

Exploitation requires control of the batch supplied to the prover along with the ability to construct a block whose header fields are consistent with execution outputs while the `transactions` root is inconsistent with the body.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The prover input for the function `stateless_validation_l1` must be attacker-controlled or independently unauthenticated.
- A downstream verifier must accept the prover output without independently recomputing and validating the header's `transactions` root against the body.

Technical Details
The function `stateless_validation_l1` performs pre-execution validation by calling the function `validate_block`, which explicitly excludes checking that the header's `transactions` root and withdrawals root match the block body, based on the assumption that the caller already performed that validation. In the prover call path, no additional check recomputes and compares the `transactions` root against `block.body.transactions`, so an internally inconsistent block can pass stateless validation as long as the remaining header fields satisfy `validate_block_header` and the post-execution checks `validate_gas_used`, `validate_receipts_root`, `validate_requests_hash`, and `validate_state_root`.

If a consumer uses the resulting proof and `last_block_hash` as evidence of validity for an L1 block header, an attacker can supply a block with a header that is locally consistent with execution outputs while setting an arbitrary `transactions` root that does not commit to the executed transaction list. Although this does not require cryptographic collisions because the `transactions` root is never verified, it does require the attacker to provide a header that matches the executed `state` root and `receipts` root for the supplied body.

Remediation
We recommend adding an explicit check on the `transactions` and `withdrawals` roots in the `stateless_validation_l1` function, or updating the `validate_block` function to optionally enforce these checks when used by the prover.

Status
The LambdaClass team has resolved the issue by introducing a check on the `transactions` and `withdrawals` roots.

Verification
Resolved.

Issue H: Incomplete Gas Used Validation Allows Nonzero Gas in Empty Blocks

Location
blockchain/blockchain.rs#L1911

Synopsis
When a block contains no transactions, the function `validate_gas_used` does not verify that the variable `block_header.gas_used` equals `0`, allowing acceptance of inconsistent gas accounting.

Impact
Medium.

A malicious producer may have an empty block accepted with nonzero gas accounting, which may skew fee calculations and downstream state such as base fee or L2 accounting.

Feasibility
High.

An attacker who can submit a block to this validator with a valid header and an empty body can exploit this without further privileges.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The block body must contain zero transactions. Accordingly, the `receipts` list must be empty, and the rest of the header fields and the `receipts` root must be valid.
- The attacker must be able to submit such a block to a node running this validation.

Technical Details
The function `validate_gas_used` compares only the last receipt's `cumulative_gas_used` to the variable `block_header.gas_used`. When the slice `receipts` is empty, `receipts.last()` is None and the function returns `Ok(())` without any comparison. As a result, a header that sets `gas_used > 0` passes. The bug arises from a failure to handle the `empty-receipts` case, where the correct expected value is `0`.

An attacker can craft a block with zero transactions, which leaves the vector `receipts` empty, and set the variable `block_header.gas_used` to an arbitrary nonzero value. The call to the function `validate_gas_used` succeeds, and subsequent validations accept the block, leading to incorrect gas accounting.

Remediation
We recommend extending the function `validate_gas_used` to explicitly check `receipts.is_empty()`. If `receipts` is empty, `block_header.gas_used` must equal `0`, and `InvalidBlockError::GasUsedMismatch` should be returned otherwise.

Status

The LambdaClass team has resolved the issue by sourcing the amount of gas used directly from the VM execution.

Verification

Resolved.

Issue I: Missing Domain Separation on Guest Output Digest Enables Cross-Context Replay

Location

`l2/prover/src/guest_program/src/openvm/src/main.rs#L18`

`l2/prover/src/guest_program/src/zisk/src/main.rs#L20`

Synopsis

The guest program hashes the encoded program output without domain separation when using the `OpenVM` or `ZisK` backends, which allows the same digest to be reused across different protocol contexts or program versions.

Impact

High.

An attacker may replay a valid output digest generated in one verification context as valid in another, which may cause incorrect result acceptance or bypass of context-specific invariants.

Feasibility

Medium.

The attack requires the ability to submit a valid proof or output to a verifier that does not bind the digest to a unique guest program identity or version.

Severity

High.

Preconditions

For this issue to occur, the following conditions must be met:

- A verifier or committer must accept an output digest without binding it to a unique guest program identifier or version.
- `ProgramOutput::encode()` semantics must be identical across multiple protocol contexts or program versions.
- An attacker must be able to submit a valid proof or output from one context into another.

Technical Details

Both guest entrypoints compute a digest over the raw bytes returned by the function `ProgramOutput::encode()` using the same hash backend per context, such as `Keccak-256` or `SHA-256`. The resulting digest commits only to the serialized output fields and omits any binding to the guest program identity, protocol context, or encoding version.

If multiple guest programs or protocol versions share the same output encoding and hash function, the same encoded output produces an identical digest across contexts. A verifier that relies solely on the

digest value without a pinned program identifier may therefore accept a replayed output that was generated under different assumptions or invariants.

Remediation
We recommend replacing direct hashing of `ProgramOutput::encode()` with a domain-separated construction that incorporates a fixed context tag, along with a program identifier and version, into the hash input.

Status
The LambdaClass team has clarified that the `vkey` already provides domain separation, as it represents a cryptographic commitment to a specific guest program binary. As a result, replayed outputs from a different program would be rejected because the `vkey` would differ. Our team agrees with this assessment and has determined the finding to be a non-issue.

Verification
Determined Non-Issue.

Issue J: Privileged Transaction Failure Path Can Be Bypassed

Location
`vm/levm/src/hooks/l2_hook.rs#L366`

Synopsis
Privileged transactions that encounter `tx_should_fail` and inject the `INVALID` opcode can still execute successfully when the transaction destination is a precompile. This occurs because precompiles bypass bytecode interpretation. Additionally, the precompile execution path casts the negative signed `gas_remaining` into a very large u64 via wrapping conversion, potentially causing incorrect gas accounting and unintended execution behavior.

Impact
Medium.

A privileged transaction that should be forced to fail can instead proceed into precompile execution with an effectively unbounded gas allowance.

Feasibility
Medium.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- The transaction must be a privileged transaction.
- The destination must be a precompile.
- The transaction must trigger the intrinsic gas failure path.

Technical Details

`Prepare_execution_privileged` performs intrinsic gas deductions.

```
// l2_hook.rs

if vm.add_intrinsic_gas().is_err() {

    tx_should_fail = true;

}
```

This causes `call_frame.gas_remaining` to be updated. If a negative value occurs, an error is returned.

```
// call_frame.rs
if self.gas_remaining < 0 {

    return Err(ExceptionalHalt::OutOfGas);

}
```

Due to the error check in `l2_hook.rs` printed above, `tx_should_fail` is set to `true`.
On this failure path, the `INVALID` opcode is pushed.

```
// l2_hook.rs
vm.current_call_frame.set_code(Code {

    bytecode: vec![Opcode::INVALID.into()].into()

})?;
```

However, precompiles do not run via the opcode interpreter. Instead, the VM dispatches directly into the precompile implementation. As a result, the injected `INVALID` bytecode is never executed.

Additionally, in the preparation for precompile execution, `run_execution` casts `gas_remaining` to `u64`. However, in this failure case, `gas_remaining` is currently a negative integer. Due to type conversion, `gas_remaining` becomes a very large positive integer.

Mitigation

We recommend the following measures to mitigate this issue:

- Ensure privileged transactions are never constructed with a destination in the precompile address range. Alternatively,
- ensure privileged transactions are always constructed with `gas_limit >= intrinsic_gas`.

Remediation

We recommend the following steps to remediate this issue:

- Short-circuit execution when `tx_should_fail` is set, returning a failed result without entering opcode or precompile execution paths.
- Guard against negative gas in the precompile dispatch path.

- Avoid wrapping casts.
- Ensure that `increase_consumed_gas` does not leave `gas_remaining` as a negative value.

Status

The LambdaClass team has updated execution to short-circuit with an `OutOfGas` revert that consumes the full gas limit whenever remaining gas is negative. This preserves the failure path regardless of whether the destination is a precompile or contract bytecode. Regression tests cover the precompile failure and success paths.

Verification

Resolved.

Issue K: Privileged Transaction Inclusion Not Guaranteed in ALIGNED_MODE

Location

l2/contracts/src/l1/OnChainProposer.sol#L502

Synopsis

When `ALIGNED_MODE` is enabled, batch verification uses `verifyBatchesAligned()` instead of `verifyBatch()`. Unlike `verifyBatch()`, the Aligned verification path does *not* enforce the privileged transaction inclusion deadline rule. As a result, in Aligned mode, it is possible to verify batches that include non-privileged transactions even while privileged transactions are past their inclusion deadline.

Impact

Medium.

Batches containing non-privileged transactions can be finalized while privileged transactions are left pending past their deadline, weakening guarantees of privileged transaction inclusion.

Feasibility

Medium.

Severity

Medium.

Preconditions

This issue can occur when `ALIGNED_MODE == true`.

Technical Details

When not in Aligned mode, `verifyBatch()` performs the following check:

```
if (
    ICommonBridge(BRIDGE).hasExpiredPrivilegedTransactions() &&
    batchCommitments[batchNumber].nonPrivilegedTransactions != 0
) {
```

```
    revert("00v"); // exceeded privileged transaction inclusion deadline,
can't include non-privileged transactions

}
```

This check guards the priority afforded to privileged transactions. `verifyBatchesAligned()` performs largely the same role as `verifyBatch` when the system runs in `ALIGNED_MODE`, but it does not perform this check.

Mitigation
We suggest continuously monitoring the bridge's "expired privileged" condition, then alerting or escalating if non-privileged batches continue to be proposed while expired privileged transactions are present.

Remediation
We recommend adding the same deadline enforcement to the Aligned verification path.

Status
The LambdaClass team has applied the same privileged-transaction inclusion deadline check enforced by the standard verification path to every batch processed under `ALIGNED_MODE`, restoring parity between the two verification paths.

Verification
Resolved.

Issue L: Fee-Token Fees Can Be Locked for Transactions That Fail Validation

Location
vm/levm/src/hooks/l2_hook.rs#L466

vm/levm/src/db/gen_db.rs#L88-L94

Synopsis
In the fee-token transaction preparation flow, fee-token deduction and locking is performed before several validation steps that can still fail. If a later validation fails, the transaction is rejected, but the earlier fee-token lock may not be rolled back. This occurs because the fee-token lock mutates fee-token contract storage via `GeneralizedDatabase::get_account_mut`, which bypasses the call-frame backup mechanism.

Impact
Medium.

A transaction that fails validation can still cause fee-token balances to be locked or deducted, resulting in unexpected fee loss or lock-up for the sender of the transaction.

Feasibility
Medium.

Triggering a later validation failure after fee locking is straightforward. For instance, submitting a transaction that will fail max-fee validation or intrinsic gas validation is sufficient.

Severity
Medium.

Technical Details
The fee-token preparation sequence performs fee-token deduction and locking and then executes validations that can still return an error.

Conceptually, the flow is as follows:

```
// 1) Compute upfront cost.

let gaslimit_price_product = env.gas_price * env.gas_limit;

// 2) Lock/deduct fees in fee-token.

deduct_caller_fee_token(vm, gaslimit_price_product * fee_token_ratio)?;

// 3) Perform additional checks that may still fail validation.

validate_sufficient_max_fee_per_gas(vm)?;

add_intrinsic_gas()?;
```

The fee-token lock mechanism performs a simulated call and then applies the resulting fee-token contract storage to the VM DB by directly mutating the account storage via `db.get_account_mut(...)`. The `GeneralizedDatabase::get_account_mut` API explicitly warns in a code comment that it bypasses call-frame backups.

As a result, if any subsequent validation fails and the transaction is rejected, the VM's standard rollback mechanism for prepare-time failures may not revert the fee-token storage mutation (because the mutation did not use backup-aware accessors). This creates a state where an invalid transaction can still lock or deduct fee tokens.

Remediation
We recommend reordering the preparation flow so that `deduct_caller_fee_token`, and any fee-token lock and deduction side effects, occur only after all validation steps that can fail have completed successfully. This prevents fee tokens from being locked or deducted for transactions that fail validation.

Status
The LambdaClass team has updated fee deduction so that fee-token storage mutations are recorded in the call-frame backup. As a result, any subsequent validation failure triggers a rollback that fully reverts the lock.

Verification
Resolved.

Issue M: Unvalidated Proof Persistence Halts Liveness

Location
[l2/sequencer/proof_coordinator.rs#L498]

l2/sequencer/proof_coordinator.rs#L563

Synopsis
The coordinator accepts and persists any submitted proof without validation, which renders a corrupted proof sticky and stops further regeneration.

Impact
Medium.

An attacker or fault may place a batch into an unusable state by causing a malformed `BatchProof` to be stored and preventing new proofs, which can halt rollup progression.

Feasibility
Medium.

A local process with access to `127.0.0.1` can submit an invalid `ProofData::ProofSubmit` without authentication.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- `Localhost` reachability to the coordinator port must be available.
- A malformed or corrupted `BatchProof` must be submitted for a target (`batch_number`, `prover_type`).
- The `store_proof_by_batch_and_type` function must persist entries without structural or semantic validation.
- No automated deletion or revalidation on read must be present.

Technical Details
The function `ConnectionHandler::handle_connection` routes `ProofData::ProofSubmit` to the function `ProofCoordinator::handle_submit`, which derives the prover type from the variable `batch_proof` and calls the function `store_proof_by_batch_and_type` without structural or semantic verification of the proof bytes. If a proof already exists for the same (`batch_number`, `prover_type`), the function `handle_submit` only logs and still returns `ProofSubmitACK` while refusing to overwrite.

During scheduling, the function `ProofCoordinator::handle_request` deems a batch complete when `get_proof_by_batch_and_type(...).is_some()` for all required types, then returns `ProofData::empty_batch_response`. Accordingly, the prover stops regenerating. A malicious client or transient corruption can submit an invalid but well-formed `BatchProof`. As a result, the system persists it, ACKs it, and never requests a replacement, while upstream verification fails repeatedly.

Remediation
We recommend replacing the existence-based completion check in the function `handle_request` with a verified status check. In addition, a validation step in the function `handle_submit` can perform structural parsing and fast proof verification before calling the function `store_proof_by_batch_and_type`. Under this approach, proofs that fail validation should be rejected and not ACKed.

Status

The LambdaClass team has clarified that on-chain verification prevents this issue and added documentation to explain the intended behavior.

Verification

Determined Non-Issue.

Issue N: L1 Watcher Cursor Advances Before Processing, Dropping Privileged Logs

Location

`l2/sequencer/l1_watcher.rs#L176`

Synopsis

The watcher advances its L1 cursor before processing and silently skips failed privileged mints, which causes permanent gaps in consumed L1 logs.

Impact

Medium.

The system may omit L1-to-L2 privileged mints, leading to inconsistent processing and reduced availability until manual replay.

Feasibility

Medium.

An unprivileged user can emit `PrivilegedTxSent`, but successful exploitation depends on timing and mempool rejection during processing, which is not consistently achievable.

Severity

Medium.

Preconditions

For this issue to occur, the following must hold true:

- The variable `last_block_fetched_l1` must be updated in the function `get_logs_l1` before logs are processed.
- The function `add_transaction_to_pool` must fail, and the function `process_privileged_transactions` must continue.
- A retry mechanism for the skipped block range must be absent.

Technical Details

The function `get_logs_l1` sets the variable `last_block_fetched_l1` to the newly computed block immediately after `get_privileged_transactions` returns, then outputs logs. The function `process_privileged_transactions` derives a `PrivilegedL2Transaction` and calls the function `add_transaction_to_pool`. On failure, it continues without recording or retrying, so the cursor advances past unprocessed logs.

An adversary can produce `PrivilegedTxSent` that derives to a transaction the mempool rejects, or can rely on transient resource pressure to induce the function `add_transaction_to_pool` to fail. Because

the code neither rewinds the variable `last_block_fetched_l1` nor persists failed offsets, those logs are never retried, and the bridge skips them permanently.

Remediation
We recommend replacing the advancement of the variable `last_block_fetched_l1` in the function `get_logs_l1` with a post-processing checkpoint updated only after successful completion of the function `process_privileged_transactions`. Instead, the implementation should rely on a persisted cursor or replay queue that retries failed ranges.

Status
The LambdaClass team has resolved the issue by allowing the processing cursor to advance only after a successful operation and generating an error instead of silently discarding it.

Verification
Resolved.

Issue O: Unbounded Read and Nonatomic Write in write_elf_file Enables Local DoS

Location
`l2/prover/src/backend/zisk.rs#L149`

Synopsis
The function `write_elf_file` reads the entire existing ELF file and rewrites it nonatomically, which can lead to local denial of service and file corruption.

Impact
Medium.

An attacker may cause process stalls, memory or I/O exhaustion, and disrupted proof generation by forcing large or blocking reads or by inducing torn writes.

Feasibility
Medium.

Exploitation requires local control of the path in the variable `ELF_PATH` or the ability to replace it with a large file, special file, or `symlink`.

Severity
Medium.

Preconditions
For this issue to occur, the following conditions must be met:

- The path in the variable `ELF_PATH` must be writable or replaceable by an untrusted local actor.
- Multiple invocations of the functions `execute`, `execute_timed`, `prove`, or `prove_timed` must run concurrently, or the process must crash mid-write.

Technical Details
The function `write_elf_file` calls `std::fs::read(ELF_PATH)` and performs a byte-wise equality check against the constant `ZKVM_ZISK_PROGRAM_ELF`. This performs an unbounded read, without

inspecting metadata. As a result, a very large file, a special device, or a pipe at `ELF_PATH` may cause excessive I/O, blocking, or memory pressure. It then uses `std::fs::write` to overwrite the file, which is not atomic and may leave a partially written file if interrupted.

A local attacker who can place a very large file, sparse file, or named pipe at `ELF_PATH` can trigger `write_elf_file` via the functions `execute`, `execute_timed`, `prove`, or `prove_timed`, causing resource exhaustion or blocking. Concurrent calls or interruption during `std::fs::write` may yield a torn ELF, causing subsequent executions to fail.

Remediation

We recommend replacing the full read and equality comparison in the `write_elf_file` function with a metadata size check and a streaming cryptographic hash comparison (for example, `SHA-256`) over `ZKVM_ZISK_PROGRAM_ELF`. Additionally, an atomic replace flow should write to a temporary file and call `std::fs::rename` only when the hash differs.

Status

The LambdaClass team has addressed the issue by performing a metadata check before reading the file to memory and writing to a temporary file, which is swapped in place of the old file after a successful `write` operation.

Verification

Resolved.

Issue P: Block Execution Pre-Rejects Transactions Based on Declared Gas Limit

Location

`vm/backends/levm/mod.rs#L64`

Synopsis

The executor rejects a transaction when `cumulative_gas_used + tx.gas_limit()` exceeds the block gas limit, even if actual gas usage would keep total gas within the limit, which introduces a non-Ethereum validity condition.

Impact

Medium.

The prover may fail to execute or prove Ethereum-valid blocks that include transactions with high-declared gas limits but low actual gas usage, which may reduce availability without accepting incorrect state transitions.

Feasibility

Medium.

Valid blocks can trigger the condition without special privileges, although such transaction patterns are uncommon.

Severity

Medium.

Preconditions

For this issue to occur, the following must hold true:

- The function `LEVM::execute_block` or the function `LEVM::execute_block_pipeline` must be used for block execution or stateless validation.
- A transaction must satisfy `cumulative_gas_used + tx.gas_limit() > block.header.gas_limit` while its actual execution would result in total `gas_used ≤ block.header.gas_limit`.

Technical Details

In `LEVM::execute_block` and `LEVM::execute_block_pipeline`, a pre-execution check of `cumulative_gas_used + tx.gas_limit() > block.header.gas_limit` rejects transactions that would be includable based on actual gas usage. This interprets the variable `tx.gas_limit()` as consumed gas rather than an upper bound. Under Ethereum semantics, block validity depends on the sum of actual gas used.

For example, with a block gas limit of `30,000,000`, two simple transfers each with `tx.gas_limit = 30,000,000` and actual `gas_used ≈ 21,000` are valid under consensus. The current check rejects the second transaction even though total `gas_used` remains below the block limit.

Remediation

We recommend replacing the pre-execution gate based on `cumulative_gas_used + tx.gas_limit()` in `LEVM::execute_block` and `LEVM::execute_block_pipeline`. A per-transaction check of `tx.gas_limit() ≤ block.header.gas_limit` should be used, with post-execution verification of total `gas_used ≤ block.header.gas_limit` via `validate_gas_used`.

Status

The LambdaClass team has pointed out that the implementation behavior matches Ethereum consensus rules. As a result, the finding has been determined to be a non-issue.

Verification

Determined Non-Issue.

Issue Q: Fee-Token Ratio Is Fetched in Both Prepare and Finalize, Allowing Inconsistent Lock vs. Settlement if the Ratio Changes During Transaction Execution

Location

`crates/vm/levm/src/hooks/l2_hook.rs#L438`

`crates/vm/levm/src/hooks/l2_hook.rs#L136`

Synopsis

`fee_token_ratio` is fetched during transaction preparation and then fetched again during finalization. If the underlying source of `fee_token_ratio` changes during the transaction's execution, the VM can "lock" fee-token amounts using one ratio and then pay or refund using another ratio. This can lead to over-locking, underpayment, incorrect refunds, or unexpected errors in fee accounting. If the ratio is intended to be constant per transaction, it should be cached once and reused.

Impact
Medium.

Inconsistent ratios could result in over-locking, underpayment and accounting errors.

Feasibility
Low.

Severity
Low.

Technical Details
The transaction flow reads the ratio twice:

- In `prepare`, the VM obtains `fee_token_ratio` and uses it to compute up-front lock and deduction amounts.
- In `finalize`, the VM obtains `fee_token_ratio` again and uses it to compute payments and refunds.

If the ratio changes between these points, the `prepare` and `finalize` stages disagree on how to convert "gas denominated amounts" into fee-token amounts, which could lead to unexpected outcomes. If `ratio_before != ratio_after`, the system could over-lock or underpay or overpay.

Remediation
We recommend caching `fee_token_ratio` once per transaction and reusing it.

Status
The LambdaClass team has updated the flow so that the fee-token ratio is fetched once during preparation and reused during finalization. As a result, lock and settlement amounts use the same value even if the underlying ratio source is mutated by the transaction.

Verification
Resolved.

Issue R: Unsanitized Boolean Leads to Nearby-Memory Blake2b Oracle

Location
`common/crypto/blake2f/x86_64.s#L45`

`common/crypto/blake2f/x86_64.s#L62-L67`

Synopsis
Due to an unsanitized boolean value (f), an attacker can potentially read from nearby memory.

Impact
Medium.

A potential memory disclosure within `Blake2b` risks the security guarantees defined by `Blake2b`.

Feasibility
Low.

Severity
Low.

Technical Details
Under the System V AMD64 ABI, for integer arguments smaller than 32 bits, the upper bits of the argument register are unspecified. Compilers often pass an `i8/_Bool/bool` in `r8b` and do not guarantee the rest of `r8` is zeroed.

If the caller does not clear `r8` fully, `r8` can contain garbage in the upper bits. Following the `add` and `shl` operations, this results in a large multiple of 32 offset from `blake2b_iv`, and 32 bytes will be read from the incorrect address.

If an attacker is able to influence the call ABI or call the function from a context where `r8` is not fully defined, the resulting read becomes a read-from-nearby-memory oracle. With `r==0` and chosen `h/t`, the final "merge" can cause the output to contain `ymm3`'s contents almost directly (for example, if `h[4..7]=0` and `t=0`, output words `4..7` become `v12..15`). As a result, the issue can leak 32 bytes at `[blake2b_iv + (garbage*32)]`.

Remediation
We recommend either masking `f` to `0` or `1` or zero-extending from byte:

- Masking `f` to `0/1`:

```
and     r8d, 1          # clear garbage upper bits; keep only bit0
add     r8d, 1
shl     r8d, 5
```

- Zero-extending from byte:

```
movzx   r8d, r8b
add     r8d, 1
shl     r8d, 5
```

Status
The LambdaClass team has resolved this issue by zero-extending the boolean parameter as recommended.

Verification
Resolved.

Issue S: Zero Length Leads to Buffer Overflow Write in SHA-3 Squeeze

Location
common/crypto/keccak/keccak1600-armv8-elf.s#L472-L541

common/crypto/keccak/keccak1600-armv8-elf.s#L791-L853

Synopsis
If the length passed is zero, the squeeze step in SHA-3 results in a buffer overflow write.

Impact
Medium.

If the passed length is attacker-controlled, the condition introduces a security risk within the SHA-3 calculation.

Feasibility
Low.

Severity
Low.

Technical Details
Both `SHA3_squeeze` and `SHA3_squeeze_cext` write out of bounds if the length passed is zero. The length is passed here for `SHA3_squeeze`.

Both functions enter the loop unconditionally and load one 64-bit word from state (for example, here for `SHA3_squeeze`), then check the length (for example, here for `SHA3_squeeze`) and branch to the byte-tail path if the length is smaller than 8. The byte-tail path always writes at least 1 byte before it checks whether length is zero. Accordingly, if the length is zero (in here), `subs x21, x21, #1` causes an underflow from `0 to 2^64-1`. (An integer underflow results in a buffer overflow.) There is no early exit and the full unrolled tail sequence of 7 bytes will be written (here, instead of the requested 0 bytes).

Even if the higher-level code prevents this, the behavior is a correctness bug and can become security-relevant if the attacker controls the length value.

Remediation
We recommend adding a zero-length early exit before the first load/store in each function (both `SHA3_squeeze` and `SHA3_squeeze_cext`), as follows:

```
cbz x21, Lsqueeze_done
```

Status
The LambdaClass team has resolved this issue by adding a zero-length early exit as recommended.

Verification
Resolved.

Issue T: Usage of Vulnerable Dependencies

Synopsis
Analyzing the project's dependencies with `cargo audit` reveals six vulnerable crates:

- `protobuf v2.28.0`: stack overflow when parsing unknown fields on user-supplied input, as described in this advisory.
- `ring v0.16.20`: panic on AES functions when overflow checking is enabled, as described in this advisory.
- `rkyv v0.8.12`: undefined behavior can occur due to null pointer use when calling safe deserialization APIs in OOM conditions, as described in this security advisory.

- `rsa v0.9.9`: RSA private key leakage through a timing side-channel, as described in this security advisory.
- `ruint v1.17.0`: out-of-bounds access may lead to memory corruption when compiling in release mode, as described in this security advisory.
- `tracing-subscriber v0.2.25`: logging user input may enable ANSI escape sequence injections, as described in this security advisory.

Impact
Consult the listed advisories on a case-by-case basis.

Feasibility
Consult the listed advisories on a case-by-case basis.

Severity
Consult the listed advisories on a case-by-case basis.

Technical Details
The Ethrex client implementation includes dependencies with known security issues that have since been resolved in updated versions.

Remediation
We recommend updating these crates and following a process that emphasizes secure crate usage to avoid introducing vulnerabilities into the Ethrex client implementation and to mitigate supply-chain attacks. This process includes:

- Manually reviewing and assessing currently used crates;
- Upgrading crates with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained crates with secure and battle-tested alternatives, if possible;
- Pinning crates to specific versions, including pinning build-level crates in the `Cargo.toml` file to a specific version;
- Only upgrading crates upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including automated dependency auditing reports in the project's CI/CD workflow.

Status
The LambdaClass team has addressed the issue by updating all vulnerable crates, except for `tracing-subscriber` v0.2.25, which is a transitive dependency of `ark-relations` v0.5.1. However, the LambdaClass team has clarified that the vulnerable logging functionality in `tracing-subscriber` is only used with static, compile-time metadata and is therefore never attacker-controlled or exploitable.

Verification
Resolved.

Suggestions

Suggestion 1: Verify That Tokens Match in WithdrawERC20

Location
`l2/contracts/src/l2/CommonBridgeL2.sol#L82`

Synopsis
`withdrawERC20` accepts both `tokenL1` and `tokenL2` but does not verify that they correspond, allowing a mismatch that can burn L2 tokens and emit a withdrawal proof that can never be claimed on L1. L1 performs the check and prevents exploitation, but users may lock their funds accidentally.

Mitigation
We recommend checking that the L1 address of `token2` matches the address of `token1` that is passed.

Status
The LambdaClass team has added a check in `withdrawERC20` requiring `token.l1Address()` to match the passed `tokenL1` address.

Verification
Resolved.

Suggestion 2: Add Router-Only Access Control to receiveETHFromSharedBridge

Location
[l2/contracts/src/l1/CommonBridge.sol#L511]

Synopsis
`receiveETHFromSharedBridge` accepts ETH from any caller, while `receiveERC20FromSharedBridge` restricts calls to SHARED_BRIDGE_ROUTER. Although direct calls only donate ETH (and thus do not present an attack vector), the inconsistency complicates bridge accounting and analysis while also weakening assumptions about the source of funds. Aligning access control removes ambiguity and maintains consistent ETH/ ERC20 flows.

Mitigation
We recommend adding the `require(msg.sender == SHARED_BRIDGE_ROUTER)` check to `receiveETHFromSharedBridge` to mirror the access control used by `receiveERC20FromSharedBridge`, thereby allowing only the router to submit ETH deposits.

Status
The LambdaClass team has added a `msg.sender == SHARED_BRIDGE_ROUTER` check to `receiveETHFromSharedBridge`.

Verification
Resolved.

Suggestion 3: Accumulate the Substate Property Instead of Overwriting It

Location
[vm/levm/src/utils.rs#L434]

Synopsis
`eip7702_set_access_code` currently sets `substate.refunded_gas` rather than modifying the value through addition. This is currently acceptable because it runs only when `substate.refunded_gas` is guaranteed to be `0`. However, a future refactor (for example, introducing a new hook in the execution pipeline) could silently overwrite the value, leading to accounting errors.

Mitigation
We recommend either asserting `substate.refunded_gas == 0` before assignment, or changing the assignment to an addition operation.

Status
The LambdaClass team has replaced the direct assignment to `substate.refunded_gas` with checked accumulation.

Verification
Resolved.

Suggestion 4: Remove Deprecated "State Diffs" Feature

Location
`guest_program/src/execution.rs#L181`

`guest_program/src/execution.rs#L400`

Synopsis
Currently, the prover maintains a `HashMap` of the accounts modified by each transaction block. This map is updated after each transaction is executed in both the `stateless_validation_l1` function and the `execute_stateless` function. As this is part of the "state diffs" feature that has since been deprecated, the values are unused and unnecessary.

Mitigation
We recommend removing the loops used to update the `HashMap` to improve performance.

Status
The LambdaClass team has removed the `StateDiff` module and the loops responsible for updating the `acc_account_updates` HashMaps.

Verification
Resolved.

Suggestion 5: Improve Prover Code Quality

Location
Inconsistent error messages:

- `prover/src/prover.rs#L112`

Unnecessary runtime checks:

- `prover/src/guest_program/src/execution.rs#L107`

Comment mismatch:

- `prover/src/backend/risc0.rs#L137-L140`

Synopsis

During our extensive review of the codebase, our team identified practices that impact its quality, readability, and maintainability. To illustrate, the following is a non-exhaustive list of recommended remediations:

- Correct the error message in `request_new_input` to accurately reflect the expected return type.
- Remove the redundant runtime check of the `l2` feature flag in `execution_program`.
- Update the comment in `to_calldata` by removing the `imageId` field from the description to accurately reflect the data stored in `calldata`.

Mitigation

We recommend addressing the items listed above to improve overall code quality, and using them as a baseline for identifying and remediating similar issues across the codebase.

Status

The LambdaClass team has corrected the error messages and the mismatched comments. Regarding the redundant runtime checks, the team stated that these checks are intentional and are intended to prevent misconfiguration.

Verification

Partially Resolved.

Suggestion 6: Improve Test Coverage

Synopsis

VM and contract logic rely mostly on a small set of unit tests and broad L2 integration tests. This leaves opcode semantics, precompile edge cases, and critical contract invariants largely unverified at the unit level.

Mitigation

We recommend improving test coverage across the codebase. In the VM, negative test cases should be added for scenarios that may result in inconsistent state. This can help confirm that rollbacks are atomic and do not result in partially committed transactions.

Status

The LambdaClass team acknowledged the suggestion and stated that coverage is being continuously expanded. We note that each remediation PR for this audit also included targeted regression tests.

Verification

Partially Resolved.

Suggestion 7: Improve Code Comments

Synopsis

There are insufficient code comments explaining the rationale behind certain critical lines of code. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation helps provide reviewers of the code with a better understanding and ability to reason about the system design.

Mitigation

We recommend expanding and improving the code comments to facilitate reasoning about the security properties of the system.

Status

The LambdaClass team has acknowledged the suggestion and has begun implementing documentation improvements.

Verification

Partially Resolved.

Suggestion 8: Improve Blake2b Implementation

Location

`common/crypto/blake2f/x86_64.s`

`common/crypto/blake2f/aarch64.rs`

Synopsis

The following items can be improved within the `Blake2b` implementation:

- The range of the rounds `r` should be clamped or validated since it is intended to be `0..12`, either by the caller function or within the `x86_64` implementation. An arbitrary `r` should not be supported.
- In addition, `r` should be sanitized here. The edge case of `r=0` will not correctly produce a meaningful result because while the rounds will be skipped, the final merge will still be performed, which is not intended.
- `rsp` is allocated and filled with `0x0500` bytes of shuffled message vectors here. Then, it is restored without clearing here. For example, in keyed `Blake2b` mode, the first block contains the key material (padded). As a result, the introduced shuffle buffer will contain that key-derived block in a recoverable form in stack memory until overwritten. Therefore, `[rsp..rsp+0x4ff]` should be zeroized before return.

Mitigation

We recommend addressing the items listed above to improve overall code quality.

Status

The LambdaClass team has addressed the aforementioned bullet points and clarified the usage of `Blake2b`:

- While the intended number of rounds mentioned is correct and should be clamped for correctness, the LambdaClass team decided to adhere to the Ethereum precompile specification (EIP-512), which allows an arbitrary number of rounds. Although our team would recommend stronger measures in this case, we acknowledge the reasoning underlying the LambdaClass team's decision.
- The edge case of `r=0` is correctly handled within the `Blake2b` computation assembly code. Initialization and finalization are still computed. The LambdaClass team stated that the existing mitigation in the assembly code adequately addresses this issue, and we concur that it is sufficient for this bullet point.
- The LambdaClass team noted that `Blake2b` is never used in keyed mode, so this comment is nonapplicable in this use case.

Our team therefore determined that this suggestion does not constitute an issue.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.