

Arcus Labs Rootchain v5 Smart Contracts

Security Assessment

June 26, 2026

Prepared for:
Arcus Labs

Prepared by: **Elvis Skoždopolj and Denys Pakizh**

Table of Contents

Table of Contents 1
Project Summary 2
Executive Summary 3
Project Goals 6
Project Targets 7
Project Coverage 8
Automated Testing 10
Codebase Maturity Evaluation 12
Summary of Findings 15
Detailed Findings 16
1. Mandatory gas burn can be bypassed 16
2. Missing inbox bounds check lets the inbox head advance past the tail 18
3. Missing version validation in the getOpposed function 20
A. Vulnerability Categories 21
B. Code Maturity Categories 23
C. Code Quality Issues 25
D. Mutation Testing 28
Interpreting the Results 29
Mutation Testing Analysis 29
Mutation Testing Conclusions 31
E. Fix Review Results 32
Detailed Fix Review Results 33
F. Fix Review Status Categories 34
About Trail of Bits 35
Notices and Remarks 36

Project Summary

Contact Information

The following project manager was associated with this project:

Emily Doucette, Project Manager
emily.doucette@trailofbits.com

The following engineering director was associated with this project:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Denys Pakizh, Consultant
denys.pakizh@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 8, 2026	Pre-project kickoff call
June 18, 2026	Delivery of report draft
June 18, 2026	Report readout meeting
June 26, 2026	Delivery of final comprehensive report with fix review

Executive Summary

Engagement Overview

Arcus Labs engaged Trail of Bits to review the security of their v5 smart contracts.

The contracts are the rootchain side of a hybrid perpetuals exchange. They custody user funds in a bridge (`BridgeVault`), gate state transitions behind an on-chain validator consensus and strictly in-order batch finalization (`ValidatorConsensus` and `CheckpointManager`), provide an emergency-settlement escape hatch through which users can recover funds if the validator set stalls, and wrap and unwrap deposits and withdrawals between the underlying stablecoin and a Paxos vault token (`PaxosDepositProxy` and `PaxosWithdrawalUnwrapper`).

A team of two consultants conducted the review from June 10 to June 16, 2026, for a total of two engineer-weeks of effort. Our testing efforts focused on the custody and accounting invariants of the bridge, the integrity of validator consensus and batch finalization, the soundness of the emergency-settlement path, the signature and replay protections for validator voting, and the access-control and upgradeability model. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. The off-chain components, the `TimelockController`, and the external Paxos `Teller` and vault contracts were out of scope.

Observations and Impact

The core value-bearing logic of the system, namely bridge custody and accounting, validator consensus, batch finalization, and the emergency-settlement exit, is implemented carefully and defensively, and our review did not uncover a way to move or strand user funds through it. The most consequential issue we identified lies in the anti-spam control that guards the permissionless raw-request entry point: a flaw in its gas-burn routine allows any unprivileged caller to skip the mandatory burn by tuning the gas supplied to the call (TOB-ARC-1). Because the entry point is permissionless and the bypass is inexpensive to reproduce, an attacker can enqueue requests far more cheaply than the design intends, inflating the off-chain processing and data-availability burden and potentially degrading the availability of the validator pipeline that must handle every request. We rated this issue medium severity.

The remaining two issues are informational. A missing bounds check can desynchronize the request-inbox accounting, but triggering it requires the trusted validator quorum (TOB-ARC-2). Additionally, an input-validation inconsistency in a read-only signaling function can mislead off-chain consumers without affecting on-chain state (TOB-ARC-3).

The codebase reflects a high level of quality and maturity. It has a thorough testing suite with high coverage, although a few coverage gaps were discovered using mutation testing

(see appendix D). The contracts are extensively documented, capturing design decisions and trust assumptions that make it easier to review and maintain the code.

The remaining risks in the system are concentrated at its integration points. The off-chain components and the validator quorum are fully trusted by the contracts; a malfunction or compromise in that layer could have a large impact on the system. The set of raw requests that users can enqueue, and how those requests are encoded, decoded, and executed, is not documented in the codebase, and we could not assess whether a crafted payload could be abused. The contracts also make assumptions about the Paxos integration, most notably a fixed 1:1 token exchange rate in the `Teller`, that are not currently guaranteed and should be verified and thoroughly tested.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Arcus Labs take the following steps before deploying the system:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.
- **Review the external and off-chain dependencies that the contracts trust.** This review has focused on the security of the smart contracts in isolation, working off of documented assumptions for integration points. The contracts trust the off-chain components and validators to behave correctly, and make assumptions about the behavior of the Paxos `Teller` contract (e.g., requiring a fixed exchange rate between the underlying and the wrapped token). These elements were out of scope for this review but are load-bearing for the on-chain system.
- **Close the test-coverage gaps identified by mutation testing.** Add tests for the surviving-mutant paths in `BridgeVault`, `CheckpointManager`, and `BlockExecutor`, then rerun the campaign to confirm the gaps are closed.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	1
Denial of Service	1
Error Reporting	1

Project Goals

The engagement was scoped to provide a security assessment of the Arcus Labs v5 smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an unauthorized actor move, mint, or withdraw funds held in `BridgeVault` custody, or otherwise violate its deposit, on-exchange, and pending-withdrawal accounting invariants?
- Are deposits, confirmations, cancellations, and withdrawals handled correctly across the full `BridgeVault` and `CheckpointManager` lifecycle, including expiry, pausing, and refund paths?
- Can the validator consensus mechanism be driven to finalize a block or execute a governance action that the validator set did not approve?
- Is quorum correctly enforced against the current validator set?
- Are the EIP-712 signature and keyed-nonce schemes used for validator voting resistant to forgery, replay, and cross-chain reuse?
- Is the emergency-settlement escape hatch sound, and can it be triggered improperly, blocked, or used to over- or under-pay margin and collateral claims?
- Are the system's access controls, role assignments, and UUPS upgradeability configured to prevent privilege escalation and unauthorized upgrades?
- Can a valid side effect be forced to revert when it is submitted via `finalizeBatch`, in order to perform a denial-of-service attack on batch finalization?

Project Targets

The engagement involved reviewing and testing the following target.

v5 contracts

Repository	https://github.com/arcus-xyz/rootchain-contracts
Version	`f7173ca64a6228073cd0e72581b301f92f613510`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- We performed a manual review of the in-scope contracts, concentrating on `BridgeVault` custody and deposit and withdrawal accounting, the `CheckpointManager` inbox, batch-finalization, and emergency-settlement flows, and the `ValidatorConsensus` block-finalization and governance-voting paths.
- We reviewed the Paxos deposit and withdrawal proxies, `PaxosDepositProxy` and `PaxosWithdrawalUnwrapper`, focusing on the wrap and unwrap flows between the underlying token and the Paxos vault token, the Permit2-based gasless deposit path and its witness binding, and the cancellation and refund logic.
- We reviewed the `EmergencySettlement` contract and the `CheckpointManager` emergency-settlement flow, including the activation gate, the settlement-summary proof, the per-account margin and collateral claim paths, the pro-rata haircut computation, and the per-account nullifiers that prevent double-claims.
- We reviewed the `ValidatorBeacon` signaling contract, including its version lifecycle, the oppose and unoppose paths, and the version-existence validation applied across its functions.
- We reviewed the helper libraries `BalanceAssertedTransfer` and `InboxWriter`, confirming that the balance-delta assertions enforced on token transfers and the inbox-write payload encoding behave as the calling contracts assume.
- We ran a `mewt` mutation-testing campaign driven by the project's existing Foundry suite to identify gaps in test coverage; results are detailed in appendix D.
- We used Slither static analysis to surface common vulnerability patterns and corroborate manual review. Slither did not surface any meaningful issues.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not review the off-chain components of the system. The contracts assume an honest validator quorum, and the soundness of that assumption depends on those out-of-scope components.
- The Paxos `Teller` and vault contracts are external to the repository under review and were not reviewed. The deposit and withdrawal proxies depend on the Teller

guaranteeing a 1:1 conversion between underlying token and the wrapped vault token, and the `bulkWithdraw` function being permissionless; if that guarantee does not hold, the wrap and unwrap assertions revert, and the deposit and withdrawal paths cease to function. This integration warrants dedicated review.

- We did not review the `TimelockController` configuration and the timelock-based access-control path; confirming whether the timelock is the sole upgrade authority, rather than being reachable through the `ValidatorConsensus` and `CheckpointManager` admin grants, warrants further review.
- We did not verify if any malicious action can be executed by submitting an arbitrary payload to the `enqueueRawRequest` function of the `BridgeVault` contract, as this would require access to the off-chain system.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tool in the automated testing phase of this project:

Tool	Description
mewt	`mewt` is a tool for running mutation testing campaigns against smart contracts and other code written in a variety of languages. At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change, uncovering gaps in test coverage. More information and results of the campaign are available in appendix D.

We ran a mutation testing campaign with mewt against all ten production contracts, driving the project's existing Foundry test suite. No vulnerability-class issues were uncovered by the tool that are not already covered under separate findings; the results below therefore focus on test coverage gaps and recommendations.

Test Results

The results of this focused testing are detailed below.

Mutation Testing Results

The summarized results of the mutation testing campaign are shown in the table below. Per-contract breakdowns and a detailed analysis of the surviving mutants are provided in appendix D.

Component	Mutants Generated	Mutants Caught	Rate
BlockExecutor.sol	75	70	**93.3%**
BridgeVault.sol	466	440	**94.4%**
CheckpointManager.sol	421	394	**93.6%**
EmergencySettlement.sol	181	181	**100%**

PaxosDepositProxy.sol	259	259	**100%**
PaxosWithdrawalUnwrapper.sol	28	28	**100%**
ValidatorBeacon.sol	163	163	**100%**
ValidatorConsensus.sol	484	484	**100%**
lib/BalanceAssertedTransfer.sol	43	43	**100%**
lib/InboxWriter.sol	11	11	**100%**
Total	**2,113**	**2,073**	**~98%**

The test suite achieves a high overall kill rate, with seven of the ten in-scope contracts reaching a perfect score. The surviving mutants are concentrated in the `BlockExecutor`, `BridgeVault`, and `CheckpointManager` contracts. After excluding equivalent and inert mutants (unsigned zero-checks, monotonic loop conditions, and OpenZeppelin base-initializer calls), the genuine gaps cluster on the emergency-settlement entry gate and getters in `CheckpointManager`, the pending-deposit accounting and approval-reset paths in `BridgeVault`, and the role grants performed in the initializers. The Arcus Labs team should add tests for these paths and rerun the campaign before production deployment; the specific recommendations are listed in appendix D.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Arithmetic is minimal and uses Solidity 0.8.35 with no unchecked blocks; the only non-trivial formula, the pro-rata settlement haircut, floors via `Math.mulDiv`, is bounded and documented, and the 1:1 wrap and unwrap paths are enforced with strict equality checks and covered by invariant tests.	**Strong**
Auditing	State-changing and administrative functions emit indexed events after their state changes, but the off-chain monitoring and incident-response procedures that the events are meant to support were not provided for review.	**Satisfactory**
Authentication / Access Controls	Privileged functions are gated through OpenZeppelin `AccessControlEnumerable` with eight distinct roles; the actors and their privileges are documented in access control documentation; the superadmins are contracts and the upgrade proposer is a Safe multisig; and roles are revocable. However, the three superadmins (`TC`, `VC`, and `CM`) all hold `DEFAULT_ADMIN_ROLE` across the governed contracts. This is required by contract design and is explicitly documented; however, the compromise of any single superadmin confers full upgrade and configuration authority. Separately, the governance voting model accepts votes for action numbers beyond the next unexecuted one, letting validators pre-cast votes and pre-sign approvals for future actions, contrary to the documented sequential-approval guarantee.	**Moderate**
Complexity Management	The contracts are simple and well written, with the largest in-scope contract at 631 lines. They use ERC-7201	**Satisfactory**

	namespaced storage separation, and carry extensive inline documentation that records assumptions and design decisions, making the code straightforward to reason about.	
Cryptography and Key Management	Voting uses EIP-712 typed data with `SignatureChecker`, keyed nonces, deadlines, and chain-id domain separation for replay protection, with no custom cryptography. Emergency settlement claims additionally rely on a sparse Merkle proof that reconstructs to a committed state root and separates leaf and internal-node preimages by length. We did not discover an issue in either the signature based or Merkle tree-based verification.	**Satisfactory**
Decentralization	Centralization is disclosed, and an emergency-settlement exit exists. However, the admin can prevent withdrawal claims by pausing the withdrawals in the `BridgeVault` contract, making the emergency settlement path not fully permissionless. Most of the in-scope contracts are upgradeable and users cannot opt out of an upgrade, and per the project documentation, `ValidatorConsensus` and `CheckpointManager` are direct superadmins that can perform upgrades without the `TimelockController` timelock, so the timelock is not the sole upgrade authority. The default admin can set the deposit and withdrawal configuration at any point, including updating the token whitelist, updating the withdrawal unwrapper contract, pausing withdrawals during an emergency settlement, and setting the `rawRequestGasBurn` and `rawRequestMaxPayloadBytes` in order to turn off the raw request paths. The off-chain system, validator configuration and incentives, multisignature wallets, and the timelock have not been reviewed.	**Further Investigation Required**
Documentation	NatSpec is extensive and supported by protocol and access-control documents. Assumptions and design decisions are clearly documented in the NatSpec, though one NatSpec security claim was found to be inaccurate, describing a governance-voting freshness guarantee (that votes for future actions cannot be pre-signed) that the	**Satisfactory**

	code does not enforce. The trust model, access controls, flow diagrams, and example flows were included in the review onboarding documentation. Documenting the off-chain system, the types of raw requests that may be enqueued and how they are encoded and executed, and the constraints the external Paxos `Teller` must satisfy (such as the assumed fixed 1:1 conversion) would further improve the documentation.	
Low-Level Manipulation	Assembly is limited and commented, but the gas-burn assembly contains an exploitable defect (TOB-ARC-1) and is not covered by a property test asserting that it consumes at least the configured amount across a range of supplied gas.	**Moderate**
Testing and Verification	The codebase contains an extensive test suite with high coverage, including a certain number of fuzz tests. We ran a mutation testing campaign that caught 98% of all mutants; seven contracts reached 100%, while the rest have some testing gaps, which we describe in appendix D. Additionally, we discovered a configuration defect in the `PaxosDepositProxyInvariant.t.sol` test file and no CI/CD pipeline exists in the shared repository.	**Satisfactory**
Transaction Ordering	The ordering surface is limited with no oracles or swaps and relies on in-order finalization, deadlines, and nonces. We did not discover any impactful transaction ordering issues.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Mandatory gas burn can be bypassed	Denial of Service	**Medium**
2	Missing inbox bounds check lets the inbox head advance past the tail	Data Validation	**Informational**
3	Missing version validation in the getOpposed function	Error Reporting	**Informational**

Detailed Findings

1. Mandatory gas burn can be bypassed	
Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-ARC-1
Target: `src/BridgeVault.sol`	

Description

The mandatory gas burn can be bypassed due to the `_burnGas` function using two invocations of the GAS opcode.

The `_burnGas` function is the anti-spam control for the `enqueueRawRequest` function, a permissionless entrypoint whose NatSpec states that "spam is bounded by an admin-set payload-size cap and a mandatory gas burn, so the inbox cannot be flooded cheaply."

However, any unprivileged caller can defeat that guarantee by providing a specific amount of gas to the call. This is because the `_burnGas` function reads the GAS opcode twice: once in the `if` condition and once in the `sub` inside the body. Because a few opcodes execute between the two reads, the second value is slightly smaller than the first. If the stipend is tuned so the first read is just above `gasToBurn` while the second is just below it, `sub(gas(), gasToBurn)` underflows to a large value. The loop condition `gt(gas(), target)` is then immediately false, so the loop never runs and the configured burn is skipped.

```
function _burnGas(uint64 gasToBurn) private view {
    assembly ("memory-safe") {
        let target := 0
        if gt(gas(), gasToBurn) { target := sub(gas(), gasToBurn) }
        for {} gt(gas(), target) {} {}
    }
}
```

Figure 1.1: The two separate `gas` reads create the underflow window. (`src/BridgeVault.sol#L383–L389`)

Exploit Scenario

An attacker deploys a thin wrapper contract that calls `enqueueRawRequest` with a gas stipend tuned to the underflow window. Each call enqueues a raw request into `CheckpointManager`'s inbox while skipping the fixed gas burn, so the attacker pays only

the residual execution and calldata cost. By repeating this across many transactions, the attacker inflates the inbox and the off-chain processing and data-availability burden at a fraction of the intended cost, degrading the availability of the validator pipeline that must process every enqueued request.

Recommendations

Short term, read `gas()` once into a single local variable and use that value for both the comparison against `gasToBurn` and the subtraction, eliminating the window between the two reads. This prevents the issue because the target can no longer underflow, so the loop always burns the intended gas.

Long term, add invariant tests that assert `enqueueRawRequest` consumes at least the configured burn across a sweep of supplied gas values, including values near `gasToBurn`. This prevents the issue because any future regression that lets a caller skip the burn is caught before deployment.

2. Missing inbox bounds check lets the inbox head advance past the tail

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARC-2
Target: `src/CheckpointManager.sol`	

Description

The `_verifyAndConsumeProcessedRequests` function advances `inboxHead` over a caller-supplied list of request hashes without checking that the consumed indices stay below `inboxTail`. The loop runs `processedRequestHashes.length` times and never bounds `inboxIdx`:

```
uint64 inboxIdx = $.inboxHead;
bytes32 rolling = bytes32(0);
for (uint256 i = 0; i < count; ++i) {
    bytes32 reqHash = processedRequestHashes[i];
    if ($.inbox[inboxIdx].hash != reqHash) revert
ProcessedRequestHashMismatch(inboxIdx);
    rolling = keccak256(abi.encodePacked(rolling, reqHash));
    ++inboxIdx;
}
if (rolling != expectedHash) revert ProcessedRequestsRollingHashMismatch();
$.inboxHead = inboxIdx;
```

Figure 2.1: The consumption loop has no `inboxIdx < inboxTail` bound. (`src/CheckpointManager.sol#L272–L282`)

Unwritten inbox slots return the default `InboxEntry`, whose `hash` is `bytes32(0)`, while real entries written by `enqueueRequest` store `keccak256(payload)`, which is never zero. A batch that commits more request hashes than the inbox holds, padding the surplus with `bytes32(0)`, passes both the per-entry check (against the empty slots) and the rolling-hash check, and writes back an `inboxHead` greater than `inboxTail`. Once that invariant is broken, requests later enqueued at indices below the new `inboxHead` are never consumed by future batches, so the associated deposits are never confirmed on the sidechain (the depositor recovers only through the expiry refund path), and `activateEmergencySettlement` reverts with `NoUnconsumedRequest` until the tail climbs back past the head.

The function is reachable only through `finalizeBatch`, restricted to `FINALIZER_ROLE`. In production, that role is held by the `ValidatorConsensus` contract, which forwards a quorum-committed `processedRequestsHash` from `executeBlock`; producing a phantom-padded set therefore requires the trusted validator quorum, which already holds

greater capability through the arbitrary side-effect dispatch in the same function. The missing check is thus a defensive-invariant gap rather than an exploitable boundary, and its value is to preserve the FIFO invariant and to catch an honest validator-software defect before it corrupts inbox accounting.

Recommendations

Short term, require `inboxHead + count <= inboxTail` before the loop (or `inboxIdx < inboxTail` within it) and revert otherwise. This prevents the issue because the head can no longer advance over inbox positions that were never written, so the `inboxHead <= inboxTail` invariant always holds and no request can be skipped.

Long term, create a list of function and system properties that must always hold, and add invariant tests for each property.

3. Missing version validation in the getOpposed function

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-ARC-3
Target: `src/ValidatorBeacon.sol`	

Description

The `getOpposed` function is the only version-parameterized function in the `ValidatorBeacon` contract that does not validate its `version_` argument. It reads the nested `opposed` mapping directly, so for a `version_` of `0` or one greater than `latestVersion`, it returns `false` for every queried address instead of reverting. Every sibling version-indexed function calls `_requireExistingVersion`, which reverts with `InvalidVersion` for those same inputs.

```
function getOpposed(uint64 version_, address[] calldata addrs) external view returns
(bool[] memory opposed) {
    ValidatorBeaconStorage storage $ = _getValidatorBeaconStorage();
    opposed = new bool[](addrs.length);
    for (uint256 i = 0; i < addrs.length; ++i) {
        opposed[i] = $.opposed[version_][addrs[i]];
    }
}
```

Figure 3.1: `getOpposed` omits the `_requireExistingVersion` check used by every other version-indexed function. (`src/ValidatorBeacon.sol#L203–L209`)

Because the function returns the zero value for a nonexistent version, a caller cannot distinguish "this version does not exist" from "this version exists and no one opposes it" using the response alone. `ValidatorBeacon` is signaling-only, so the consequence is limited to API inconsistency and off-chain misinterpretation rather than on-chain state corruption.

Recommendations

Short term, call `_requireExistingVersion` at the start of `getOpposed` so it matches the behavior of every other version-indexed function in the contract. This prevents the issue because queries for nonexistent versions revert rather than returning an ambiguous result.

Long term, add API-consistency tests that exercise invalid-version handling across all beacon view and mutation paths. This prevents the issue because any function that diverges from the uniform validation behavior is caught before deployment.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **The `initialize` function of the `ValidatorConsensus` contract does not enforce that the `validators` input array contains only unique addresses.** The `threshold` cannot be set lower than the number of unique validators due to the `_validateThreshold` check; however, this could lead to the threshold silently becoming a larger percentage of the total validator number.

```
for (uint256 i = 0; i < validators.length; ++i) {
    _grantRole(VALIDATOR_ROLE, validators[i]);
}
_validateThreshold(threshold_);
```

Figure C.1: The return value of `_grantRole` is ignored. (src/ValidatorConsensus.sol#L149–L152)

- **The `PaxosDepositProxy` deposit entrypoints accept an `IERC20 token` argument that is only ever used to revert with `WrongToken` when it does not equal the configured `underlyingToken`.** This parameter should be removed and the stored `underlyingToken` used directly, eliminating redundant caller input and the risk of an integrator passing an arbitrary token where the underlying token is required.

```
function initiateDeposit(address owner, uint16 accountIndex, IERC20 token,
uint256 amount)
    external
    nonReentrant
    returns (uint64 inboxIndex)
{
    // ...
}

// ...

function _initiateDeposit(InitiateParams memory p) private returns (uint64
inboxIndex) {
    PaxosDepositProxyStorage storage $ = _getPaxosDepositProxyStorage();
    if (address(p.token) != address($.underlyingToken)) revert
WrongToken(address(p.token));
    // ...
    uint256 wrappedAmount = _wrap($, p.payer, p.amount); // _wrap reads
$.underlyingToken from storage
}
```

Figure C.2: The caller-supplied `token` is always validated against the stored `underlyingToken`. (src/PaxosDepositProxy.sol#L150, #L282–L288)

- **`initiateDeposit` function accepts an owner parameter but immediately reverts if it differs from `msg.sender`.** The parameter is redundant; `msg.sender` could be used directly.

```
function initiateDeposit(address owner, uint16 accountIndex, IERC20 token,
uint256 amount)
    external
    nonReentrant
    returns (uint64 inboxIndex)
{
    if (owner != msg.sender) revert NotOwner(msg.sender, owner);
    // ...
}
```

Figure C.3: `initiateDeposit` accepts `owner` parameter. (src/PaxosDepositProxy.sol#L150–L155)

- **`_haircutPayout` function recomputes `netEquity` and `distributable` variables on every `proveSettlementMargin` call,** but both values are fixed once `proveSettlementSummary` runs. They could be computed once there and cached in storage.

```
uint256 p = $.settlementTotalPositiveMargin;
if (p == 0) return 0;
uint256 n = $.settlementTotalBadDebt;
uint256 netEquity = p > n ? p - n : 0;
uint256 distributable = Math.min($.settlementMarginPool, netEquity);
return Math.mulDiv(positiveMargin, distributable, p);
```

Figure C.4: `netEquity` and `distributable` variables are recomputed on every call. (src/CheckpointManager.sol#L414–L419)

- **`cancelBlockVoteSignatures` function is callable by any address,** despite being semantically intended only for validators to revoke `voteBlockBySig` signatures.

```
function cancelBlockVoteSignatures(uint64 blockNumber) external returns
(uint256 usedNonce) {
    return _useNonce(msg.sender, uint192(blockNumber));
}
```

Figure C.5: `cancelBlockVoteSignatures` has no role restriction. (src/ValidatorConsensus.sol#L265–L267)

- **`__NoncesKeyed_init` function is not called in `ValidatorConsensus` initializer.**

```
__UUPSUpgradeable_init();
__AccessControlEnumerable_init();
```

```
__ReentrancyGuardTransient_init();
__EIP712_init("ValidatorConsensus", "1");
```

Figure C.6: Parent initializer calls in `initialize` function where `__NoncesKeyed_init` is absent. (src/ValidatorConsensus.sol#L138–L141)

- **The `hashGovernanceAction` NatSpec incorrectly states that `parentActionHash` prevents validators from pre-signing votes for future governance actions.** The action hash is deterministic once a planned action sequence is known, and EIP-712 votes can always be signed off-chain before they are relayed. Additionally, `_castGovernanceVote` allows submitting votes for future actions; this could be updated to require `actionNumber == executedGovernanceCount` in order to prevent pre-submission of votes.

```
/// @dev Each action commits to `parentActionHash` — the hash of the
previously
/// executed governance action, like a block's `parentHash`. The parent is
only
/// known once the prior action executes, so validators cannot pre-sign votes
for
/// future actions; this limits signature harvesting.
```

Figure C.7: The NatSpec claims future governance votes cannot be pre-signed. (src/ValidatorConsensus.sol#L470–L473)

D. Mutation Testing

This appendix outlines our mutation testing process for the Arcus Labs smart contracts and highlights the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in test coverage. Although mutation testing is a slow process, it enables auditors to focus their review on areas of the codebase most likely to contain latent bugs, and it enables developers to identify and add missing tests.

We used our tool mewt to run our mutation testing campaign. It provided valuable insights into the test suites implemented across the contracts. This tool can be installed by running the command shown in figure D.1 or by following the alternative instructions in the README file.

```
curl --proto '=https' --tlsv1.2 -LsSf
https://github.com/trailofbits/mewt/releases/latest/download/mewt-installer.sh | sh
```

Figure D.1: The command used to install `mewt`

Once installed, a mutation campaign can be run against all source and test files in a repository by running the commands shown in figure D.2.

```
mewt mutate .
mewt run --test.cmd "forge test"
```

Figure D.2: The `mewt` commands used to run a full mutation testing campaign against all source files in a repository

Consider the following notes about the mutation testing campaign:

- We ran the campaign against all ten production contracts under `src/`, using the provided Foundry test suite. The `src/test/` mocks, `src/interfaces/`, and third-party dependencies were excluded.
- The `--test.cmd` argument given to the `mewt` executable designates the test command. Given the long runtime, the tool estimates the test suite's runtime and sets a conservative timeout to prevent any single test from stalling the campaign.

Interpreting the Results

An example output line is shown in figure D.3. This illustrates the general output format of the tool after running the `mewt results` command.

```
Target: src/SampleContract.sol
  Uncaught  | [CR 555] Line 248: 'original_line(example);' -> '/*
original_line(example) */;'
```

Figure D.3: Example output from the mutation testing campaign

The first line specifies the file being mutated. The next line contains an uncaught mutant sample, following this format:

```
[Type ID] Line number: 'original line' -> 'mutated line'
```

The type specifies which mutator was applied; for example, CR, which stands for "code removal," means that the whole line was commented out. The line number and original line show where the mutation was applied, and the mutated line shows the result of applying the mutator. The full list of mutators can be shown by running `mewt print mutations`.

A number of surviving mutations are equivalent or inert and have been excluded from the analysis below; no action is required for them:

- Replacing `x == 0` with `x <= 0` on an unsigned integer or an `address` is equivalent, since neither can be less than zero.
- Replacing `i < len` with `i != len` in a for-loop over a `uint256` index starting at zero is equivalent.
- Commenting out the OpenZeppelin base-initializer calls (`__UUPSUpgradeable_init`, `__AccessControlEnumerable_init`, `__ReentrancyGuardTransient_init`) survives because the unit tests do not exercise the upgrade, re-initialization, or reentrancy machinery in isolation. These establish properties enforced by the framework rather than by unit assertions.

Mutation Testing Analysis

A ~98% kill rate indicates the test suite detects the large majority of injected faults. Seven of the ten contracts reach a 100% kill rate. After excluding the equivalent and inert mutants described above, the genuine coverage gaps are concentrated in three contracts: `BlockExecutor`, `BridgeVault`, and `CheckpointManager`.

- **`CheckpointManager.sol`.** The emergency-settlement entry gate `if ($.settlementActivatedAt == 0) revert` (`src/CheckpointManager.sol#L341`) survives both code-removal and if-false mutation: no test verifies that `proveSettlementSummary` reverts when called

outside emergency settlement. The external proof call (src/CheckpointManager.sol#L345-L348) and the `settlementTotalBadDebt` storage write (src/CheckpointManager.sol#L352) survive code removal, and the `settlementTotalPositiveMargin` / `settlementTotalBadDebt` getters survive `require(false)` replacement. No test asserts these return values or that the storage is written. The `INBOX_WRITER_ROLE` grant in the initializer (src/CheckpointManager.sol#L194) also survives.

- **`BridgeVault.sol`.** The `pendingDeposits -= d.amount` accounting in `cancelDeposit` and `confirmDeposit` (src/BridgeVault.sol#L338, src/BridgeVault.sol#L354) survives operator replacement, and the `delete $.pendingDepositByIndex[inboxIndex]` after cancellation (src/BridgeVault.sol#L339) survives removal: no test asserts the running balance or that a cancelled entry cannot be reprocessed. The residual-approval reset `token.forceApprove(unwrapper, 0)` (src/BridgeVault.sol#L466) survives removal, the deposit-expiry boundary `nowTs >= d.expiresAt` (src/BridgeVault.sol#L352) is never exercised strictly past expiry, the depositor-vs-owner argument order in `writeDeposit` (src/BridgeVault.sol#L298) is not distinguished, and `rawRequestGasBurn` is never read after initialization.

- **`BlockExecutor.sol`.** The `_grantRole(DEFAULT_ADMIN_ROLE, admin)` call in the constructor (src/BlockExecutor.sol#L69) survives removal: no test confirms the admin role is granted.

A small number of additional low-priority survivors reflect the absence of adversarial or boundary inputs to access-control and length-check paths (for example, `msg.sender != checkpointManager` direction shuffles), but they do not indicate logic errors in the production code.

Additionally, re-testing surfaced a mis-targeted invariant test. Because `PaxosDepositProxyInvariant.t.sol` calls `targetSelector` function without also calling `targetContract(address(handler))`, the invariant campaign is not restricted to the handler; setup-deployed contracts remain eligible fuzz targets, including the mocks. The fuzzer can then call `MockPaxosVault.mint` directly with the proxy as recipient, planting funds the proxy never received and tripping `invariant_proxyNeverRetainsFunds` on correct code for some seeds. This can only inflate the caught count, never the survivor count, so every survivor reported here is a real survivor under this campaign, though some may still be equivalent or low-value mutants. Adding `targetContract(address(handler))` and rerunning would remove the flaky failure and produce a precise rate.

```
// ...
bytes4[] memory selectors = new bytes4[](4);
```

```
selectors[0] = PaxosDepositProxyHandler.initiateDeposit.selector;
selectors[1] = PaxosDepositProxyHandler.initiateDepositReturnToOwner.selector;
selectors[2] = PaxosDepositProxyHandler.initiateDepositReturnToSender.selector;
selectors[3] = PaxosDepositProxyHandler.cancelDeposit.selector;

targetSelector(FuzzSelector({addr: address(handler), selectors: selectors}));
```

Figure D.4: The `SetUp` function does not call `targetContract` (`test/paxos/PaxosDepositProxyInvariant.t.sol#L191-L197`)

Mutation Testing Conclusions

We recommend that the Arcus Labs team thoroughly review and evaluate the mutation test results, add tests that close the genuine coverage gaps above, and then rerun a mutation testing campaign to confirm the added tests provide adequate coverage. Equivalent and inert mutants (unsigned zero-checks, monotonic loop conditions, base-initializer calls) can be left in the surviving set without action. The highest-impact additions are negative tests on the `proveSettlementSummary` entry gate and getters, assertions on `pendingDeposits` accounting and pending-deposit deletion in BridgeVault, and assertions on the role grants performed in the initializers.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On June 25, 2026, Trail of Bits reviewed the fixes implemented by the Arcus Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, all three issues described in this report have been resolved by the Arcus Labs team. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Mandatory gas burn can be bypassed	Medium	Resolved
2	Missing inbox bounds check lets the inbox head advance past the tail	Informational	Resolved
3	Missing version validation in the getOpposed function	Informational	Resolved

Detailed Fix Review Results

TOB-ARC-1: Mandatory gas burn can be bypassed
Resolved in PR #16. The `_burnGas` function in `BridgeVault.sol` was updated to snapshot the current gas value into a local variable once and use that single snapshot for both the threshold comparison and the subtraction. This eliminates the two-opcode window between the two `GAS` reads that previously allowed an attacker to tune a call's gas stipend so that the first read exceeded `gasToBurn` while the second fell below it, causing the burn loop to never execute.

TOB-ARC-2: Missing inbox bounds check lets the inbox head advance past the tail
Resolved in PR #17. A bounds check was added to `_verifyAndConsumeProcessedRequests` in `CheckpointManager.sol` that requires the computed new inbox head (current head plus the number of provided hashes) to not exceed the current tail before the consumption loop runs. Supplying more request hashes than the inbox holds now reverts, preventing `inboxHead` from advancing past `inboxTail`.

TOB-ARC-3: Missing version validation in the getOpposed function
Resolved in PR #15. A call to `_requireExistingVersion` was added at the start of `getOpposed` in `ValidatorBeacon.sol`, making it consistent with every other version-indexed function in the contract. Queries for version `0` or a version greater than `latestVersion` now revert with `InvalidVersion` instead of silently returning `false`.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2026 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

Trail of Bits considers this report public information; it is licensed to Arcus Labs under the terms of the project statement of work and has been made public at Arcus Labs' request. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.