GUARDIAN

Olympus

Migration Review

Security Assessment

February 11th, 2026

Summary

Audit Firm Guardian

Prepared By Felipe Gomez, Robert Regeida

Client Firm Olympus

Final Report Date February 11, 2026

Audit Summary

Olympus engaged Guardian to review the security of their Migration Review of Olympus. From the 29th of January 2026 to the 2nd of February 2026, a team of 2 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Confidence Ranking

Given the lack of High and Critical issues detected during the main review, Guardian assigns a Confidence Ranking of 5 to the protocol. Guardian advises the protocol to consider periodic review with future changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

Note: Fixes to the findings uncovered in the remediation review have not been reviewed by Guardian. Guardian recommends the client either perform an amended follow-up review period to focus on the fixes to remediations findings or seek a follow-up audit from another team.

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

Guardian Confidence Ranking

Confidence Ranking	Definition and Recommendation	Risk Profile
5: Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. **Recommendation:** Code is highly secure at time of audit. Low risk of latent critical issues.	0 High/Critical findings and few Low/Medium severity findings.
4: High Confidence	Code is clean, well-structured, and adheres to best practices. Only 1 Significant issue was uncovered per week. Design patterns are sound, and test coverage is strong. **Recommendation:** Suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity issues. Varied Low severity findings.
3: Moderate Confidence	Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases. **Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1-2 High/Critical findings per engagement week.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities. Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. **Recommendation:** Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week. Or additional High/Critical findings uncovered in remediation review which have not been resolved and confirmed by Guardian.
1: Very Low Confidence	Code has systemic issues. Multiple High/Critical findings (≥5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured. **Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Overview

Project Summary

Project Name	Olympus
Language	Solidity
Codebase	https://github.com/OlympusDAO/olympus-v3
Commit(s)	Main Review commit: bfd7446bcb936fbf34c368d9664f09f5a33cf65a Remediation Review commit: 3627a72b65859c881ed1f1388e1c49b2f77b85c0

Audit Summary

Delivery Date	February 11, 2026
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
● Critical	0	0	0	0	0	0
● High	0	0	0	0	0	0
● Medium	0	0	0	0	0	0
● Low	3	0	0	0	1	2
● Info	12	0	0	3	0	9

Audit Scope & Methodology

https://github.com/OlympusDAO/olympus-v3/pull/196

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	● Critical	● High	● Medium
Likelihood: *Medium*	● High	● Medium	● Low
Likelihood: *Low*	● Medium	● Low	● Low

Impact

High	Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.
Medium	A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.
Low	Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High	The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.
Medium	An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.
Low	Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Findings & Resolutions

ID	Title	Category	Severity	Status
L-01	Cannot Update Root Or Cap When Paused	Unexpected Behavior	● Low	Resolved
L-02	Proposal Activation Fails With Legacy Migrations	Logical Error	● Low	Partially Resolved
L-03	Migration Proposal Griefed Via Timelock Donation	DoS	● Low	Resolved
I-01	Helper Burns All tempOHM It Pulls	Unexpected Behavior	● Info	Resolved
I-02	Activation Fails If Category Already Exists	Unexpected Behavior	● Info	Resolved
I-03	Tenderly Sim Uses daoMS	Unexpected Behavior	● Info	Resolved
I-04	Helper Lacks Token Rescue Mechanism	Warning	● Info	Resolved
I-05	Migration Cap Is Remaining Approval	Warning	● Info	Resolved
I-06	Partial Migrations Can Lose Dust	Rounding	● Info	Resolved
I-07	incurDebt Fails To Handle Fee On Transfers	Logical Error	● Info	Acknowledged
I-08	Bridge Deactivation Does Not Block Inbound Mints	Logical Error	● Info	Acknowledged
I-09	retryMessage Ignores Trusted-Remote Checks	Validation	● Info	Acknowledged
I-10	setMerkleRoot Lacks Same-Root Guard	Validation	● Info	Resolved

L-01 | Cannot Update Root Or Cap When Paused

Category	Severity	Location	Status
Unexpected Behavior	● Low	V1Migrator.sol	Resolved

Description

The administrative setters for the Merkle root and migration cap are gated by onlyEnabled. If the migrator is disabled for incident response, governance cannot update these values without first re enabling the policy. That creates a window where the old configuration is live or forces operators to re enable before fixing configuration.

```
function setMerkleRoot(bytes32 merkleRoot_) external onlyEnabled
onlyAdminOrLegacyMigrationAdmin {
_currentMerkleNonce++;
merkleRoot = merkleRoot_;
emit MerkleRootUpdated(merkleRoot_, msg.sender);
}
function setMigrationCap(uint256 cap_) external onlyEnabled onlyAdminRole {
_setMigrationCap(cap_);
}
```

The impact is reduced operational flexibility during a pause scenario.

Example: a bad merkle root is published that excludes a large set of eligible holders. The team disables the migrator to stop further migrations and wants to update the merkle root (or reduce the cap). Because both setters are gated by onlyEnabled, they must re-enable the migrator to apply the fix, re-opening migrations under the broken configuration.

Recommendation

Allow these admin setters to run while disabled, or provide separate admin only functions that can update configuration without re enabling the migrator. Keep migrate itself gated by onlyEnabled.

Resolution

Olympus Team: The issue was resolved in PR#205.

L-02 | Proposal Activation Fails With Legacy Migrations

Category	Severity	Location	Status
Logical Error	● Low	MigrationProposalHelper.sol: 147	Partially Resolved

Description PoC

MigrationProposalHelper always migrates the full OHMv1ToMigrate amount via the legacy TokenMigrator. If this configured amount exceeds the migrator's actual capacity (bounded by its gOHM balance and oldSupply headroom), the legacy migrator reverts and the helper activation fails.

Even if the amount was calibrated at planning time, a gOHM balance decrease before activation (e.g., legacy users migrating) creates the same failure mode. This can cause the governance proposal to fail if OHMv1ToMigrate is mis-calibrated or capacity drifts.

Example flow

1. Ops compute capacity off-chain and set OHMv1ToMigrate via the setup batch.

2. Ops mint tempOHM based on that stored amount.

3. A normal user migrates through the legacy TokenMigrator, reducing its gOHM balance.

4. Proposal executes activate(), which calls the legacy migrator for the full (now-stale) OHMv1ToMigrate and reverts due to insufficient capacity.

Recommendation

Compute and cap OHMv1ToMigrate to the migrator's capacity before execution (e.g., min(gOHM balance → OHMv1, oldSupply - currentSupply) with a small buffer), and include this check in the runbook or setup scripts.

Resolution

Olympus Team: Partially Resolved.

L-03 | Migration Proposal Griefed Via Timelock Donation

Category	Severity	Location	Status
DoS	● Low	MigrationProposal.sol: 227	Resolved

Description

MigrationProposal._validate requires the time lock to hold zero OHMv2, OHMv1, gOHM, and tempOHM balances after execution:

```
require(IERC20(OHMv2).balanceOf(timelock) == 0, "There should be no OHMv2 left in the Timelock");
require(IERC20(GOHM).balanceOf(timelock) == 0, "There should be no gOHM left in the Timelock");
```

The _validate function runs on-chain as part of proposal execution (not just simulation). An attacker can grief the migration proposal by:

1. Observing the migration proposal in the governance queue

2. Donating 1 wei of OHMv2 (or gOHM) to the timelock address

3. When the proposal executes, _validate() reverts because balanceOf(timelock) != 0

4. The entire governance proposal fails

Impact: DoS of migration governance proposal, requiring resubmission and new governance vote. The validation is overly strict—it checks tokens that are not part of the migration flow (OHMv2/gOHM can exist in the time lock for unrelated reasons). Only tempOHM is migration-specific and should be zero after activation.

Recommendation

1. Remove OHMv2 and gOHM balance checks from _validate() (they are not migration-specific)
2. Or use delta-based assertions: record pre-execution balances and assert no net increase
3. Or add a sweep mechanism to handle unexpected donations before validation

Resolution

Olympus Team: The issue was resolved in PR#203.

I-01 | Helper Burns All tempOHM It Pulls

Category	Severity	Location	Status
Unexpected Behavior	● Info	src/proposals/MigrationProposalHelper.sol	Resolved

Description

The helper transfers the entire tempOHM balance from the timelock, deposits only the computed amount into the legacy treasury, and then burns any remaining tempOHM. If the timelock holds extra tempOHM for any reason, that surplus is destroyed during activation.

```
ERC20(TEMPOHM).safeTransferFrom(owner, address(this), ERC20(TEMPOHM).balanceOf(owner));
uint256 tempOHMToDeposit = getTempOHMToDeposit();
ERC20(TEMPOHM).safeApprove(TREASURY, tempOHMToDeposit);
ohmV1Minted = IOlympusTreasury(TREASURY).deposit(tempOHMToDeposit, TEMPOHM, 0);
```

```
uint256 excessTempOHM = ERC20(TEMPOHM).balanceOf(address(this));
if (excessTempOHM > 0) {
ERC20Burnable(TEMPOHM).burn(excessTempOHM);
}
```

The impact is potential destruction of tempOHM balances beyond the intended migration amount.

Recommendation

Transfer only the required tempOHM amount, or explicitly document that any extra tempOHM in timelock will be burned and ensure operational processes keep the balance tight.

Resolution

Olympus Team: The issue was resolved in PR#202.

I-02 | Activation Fails If Category Already Exists

Category	Severity	Location	Status
Unexpected Behavior	● Info	MigrationProposalHelper.sol	Resolved

Description

The helper always calls `addCategory("migration")` before burning. The Burner policy reverts if the category is already approved.

If the category was added earlier by any operational test or previous attempt, the helper activation will revert and the governance execution will fail even though all other steps are correct.

```
Burner(BURNER).addCategory(MIGRATION_CATEGORY);
```

```
if (categoryApproved[category_]) revert Burner_CategoryApproved();
```

Example: the DAO runs a dry-run proposal that adds the "migration" category in `Burner`. Later, when the real proposal executes on mainnet (or when the helper is re-deployed for a second attempt after a failed run), `addCategory("migration")` reverts because the category already exists and the activation transaction fails even though everything else is correct.

Recommendation

Make the category addition idempotent. Check whether the category is already approved before adding it, or use a unique category name for this specific migration to avoid collisions.

Resolution

Olympus Team: The issue was resolved in PR#204.

I-03 | Tenderly Sim Uses daoMS

Category	Severity	Location	Status
Unexpected Behavior	● Info	OlyBatch.sol	Resolved

Description

The Tenderly simulation path in OlyBatch hardcodes the from address to daoMS, even when the batch is intended to be executed by the policy or emergency safe. In OlyBatch, the real sender is safe, which is set by the modifier used (isDaoBatch, isPolicyBatch, isEmergencyBatch).

By forcing daoMS into the Tenderly payload, the simulation can succeed under DAO permissions even though the actual sender would fail on-chain.

This is an operational/simulation mismatch. It can mislead reviewers into thinking a batch is safe to execute when the real sender lacks the required roles or balances.

Example: a batch guarded by isPolicyBatch is supposed to be executed by the policy multisig.

Tenderly simulation runs with daoMS as the sender, passes due to DAO permissions, and the team believes the batch is safe. When executed from the policy multisig, the transaction can revert due to missing roles, causing an avoidable deployment or governance delay.

Note: the newer BatchScriptV2 uses _owner as the sender in its Tenderly payload, which matches the real sender. This issue is specific to OlyBatch and was updated during this PR review (commit e255afe).

Recommendation

Use the actual safe address (the batch sender) for the Tenderly payload from field instead of always daoMS.

Resolution

Olympus Team: The issue was resolved in PR#208.

I-04 | Helper Lacks Token Rescue Mechanism

Category	Severity	Location	Status
Warning	● Info	MigrationProposalHelper.sol	Resolved

Description

The migration helper is a permanent contract with no method to recover tokens sent to it accidentally. Any `ERC20` transferred to the helper outside of the activation flow will remain stuck forever.

Recommendation

Add an `onlyOwner` sweep function for arbitrary tokens, or explicitly document that the helper must never receive tokens outside the activation sequence.

Resolution

Olympus Team: The issue was resolved in PR#209.

I-05 | Migration Cap Is Remaining Approval

Category	Severity	Location	Status
Warning	● Info	V1Migrator.sol	Resolved

Description

The migration cap logic is implemented by synchronizing `MINTR` `mintApproval` to a target value. Because `MINTR` approvals are consumed when minting, the value represents remaining allowance rather than a lifetime total.

The function name and comments can be interpreted as setting an absolute cap, which can lead to operational mistakes if governance resets the approval without accounting for already minted amounts.

The impact is governance confusion and potential misconfiguration of the migration allowance.

Example: 600 OHM v2 has already been minted and the remaining approval is 400. Governance calls `setMigrationCap(1000)` expecting the total cap to be 1000 overall.

The code sets the remaining approval to 1000, allowing another 1000 to be minted, for a total of 1600. The system behaved correctly, but the operator expectation was wrong because the cap is “remaining allowance,” not “lifetime total.”

Recommendation

Clarify semantics in naming and documentation, or track a separate total cap in `V1Migrator` and enforce minted total against it. If using remaining approvals, rename to something like `setRemainingMintApproval` and update `NatSpec` accordingly.

Resolution

Olympus Team: The issue was resolved in PR#210.

I-06 | Partial Migrations Can Lose Dust

Category	Severity	Location	Status
Rounding	● Info	V1Migrator.sol: 281	Resolved

Description

Migration converts OHM v1 to OHM v2 by calling gOHM.balanceTo and then gOHM.balanceFrom. Both conversions round down. If a user migrates in multiple small transactions, dust can be lost on each conversion rather than once on the total amount.

```
uint256 gohmAmount = _GOHM.balanceTo(amount_);
uint256 ohmV2Amount = _GOHM.balanceFrom(gohmAmount);
if (ohmV2Amount == 0) revert ZeroAmount();
```

```
function balanceFrom(uint256 _amount) public view override returns (uint256) {
return _amount.mul(index()).div(10**decimals());
}
function balanceTo(uint256 _amount) public view override returns (uint256) {
return _amount.mul(10**decimals()).div(index());
}
```

The impact is minor value loss for users who migrate in many small calls.

Recommendation

Add a previewMigrate view helper and document that users should migrate in a single transaction when possible. Consider adding a minimum amount threshold or UI guidance to reduce dust loss.

Resolution

Olympus Team: The issue was resolved in PR#207.

I-07 | incurDebt Fails To Handle Fee On Transfers

Category	Severity	Location	Status
Logical Error	● Info	OlympusTreasury.sol: 114	Acknowledged

Description

The `incurDebt` function increases `reserveDebt` and `totalDebt` by `amount_` before transferring tokens to the debtor. If the system supports fee-on-transfer or deflationary tokens, the debtor receives less than `amount_`, but debt is recorded for the full amount.

This overcharges the debtor and causes `getReserveBalance` (balance + `totalDebt`) to overstate total reserves by the amount lost to fees.

Recommendation

Either reject fee-on-transfer/deflationary tokens for debt operations, or measure the actual amount sent to the debtor (e.g., recipient balance delta) and record debt accordingly. Alternatively, perform a pre/post balance check on the treasury and adjust debt to the net amount that leaves the treasury.

Resolution

Olympus Team: Acknowledged.

I-08 | Bridge Deactivation Does Not Block Inbound Mints

Category	Severity	Location	Status
Logical Error	● Info	CrossChainBridge.sol: 204	Acknowledged

Description

`setBridgeStatus(false)` only affects `sendOhm` (outbound burns). Inbound minting via `lzReceive`/`retryMessage` continues even when the bridge is "deactivated," because neither path checks `bridgeActive`.

This makes the kill-switch asymmetric: operators may believe a shutdown halts minting, but messages from trusted remotes will still mint OHM on the destination chain.

Recommendation

If shutdown should stop all bridge activity, add `bridgeActive` checks to `lzReceive` and `retryMessage` (or explicitly document that deactivation is outbound-only and use trusted-remote removal to halt inbound).

Resolution

Olympus Team: Acknowledged.

I-09 | retryMessage Ignores Trusted-Remote Checks

Category	Severity	Location	Status
Validation	● Info	CrossChainBridge.sol: 216	Acknowledged

Description

`retryMessage` only verifies the stored payload hash and then calls `_receiveMessage` directly. It does not re-check whether the trusted-remote entry is still configured. As a result, a previously failed payload can be replayed to mint OHM even after operators clear trusted remotes.

Recommendation

Re-validate trusted remote/endpoint constraints (or explicitly document that retries bypass these controls). Consider invalidating stored payloads when removing trusted remotes.

Resolution

Olympus Team: Acknowledged.

I-10 | setMerkleRoot Lacks Same-Root Guard

Category	Severity	Location	Status
Validation	● Info	V1Migrator.sol: 319	Resolved

Description

`setMerkleRoot` increments the migration nonce and replaces the root, without validating that the new root differs from the current one. Existing proofs for the previous root become invalid, and per-user migrated amounts reset for the new nonce.

A mid-campaign root update can block users until new proofs are generated and distributed. If the same root is accidentally set again, the nonce still increments and migrated tracking resets, allowing a full re-migration under the same tree.

Recommendation

Add a guard to reject `setMerkleRoot` when `merkleRoot_ == merkleRoot` (or otherwise require an explicit “reset” flag), and document that root updates are campaign resets. If keeping the current behavior, update the runbook to warn that same-root updates re-open migrations.

Resolution

Olympus Team: The issue was resolved in PR#206.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
I-01	TempOHM Allowance Drift Can Revert	Warning	● Info	Resolved
I-02	V1Migrator.rescue() Blocked When Disabled	Access Control	● Info	Resolved

I-01 | TempOHM Allowance Drift Can Revert

Category	Severity	Location	Status
Warning	● Info	MigrationProposal.sol, MigrationProposalHelper.sol	Resolved

Description

The governance proposal approves a fixed tempOHM allowance based on the timelock balance at proposal build time, but the helper later transfers the timelock's full balance at execution time. If the timelock balance changes between those two moments, the transferFrom can exceed the approved amount and revert, causing the entire proposal execution to fail.

Operationally, the migration plan expects the timelock to hold only the intended amount of tempOHM and to have that balance remain unchanged until execution. That expectation is not enforced in code. tempOHM is a standard ERC20 and is transferable, so any holder can donate dust to the timelock and increase the balance.

The same effect occurs if the tempOHM owner mints additional tokens to the timelock after the proposal is queued. In either case the helper tries to pull the new, higher balance even though the allowance was set to the old balance, and the proposal reverts.

This does not imply a loss of protocol funds and may never happen if the team strictly ensures that the timelock is the only tempOHM holder and no additional minting occurs. The point is that this safety relies on an off-chain assumption rather than an on-chain invariant and the failure mode is a governance execution revert that requires a re-submission.

Recommendation

Make the transfer deterministic relative to the approved amount. The simplest approach is to approve type(uint256).max and revoke to zero after execution. Alternatively, transfer only the expected amount (for example, getTempOHMToDeposit or min(balance, allowance)) and handle any excess in a separate, explicit burn step. This removes the dependency on timelock balance stability and makes execution robust even if the balance drifts.

Resolution

Olympus Team: The issue was resolved in PR#214.

I-02 | V1Migrator.rescue() Blocked When Disabled

Category	Severity	Location	Status
Access Control	● Info	src/policies/V1Migrator.sol: 352	Resolved

Description

The newly added `rescue()` function has the `onlyEnabled` modifier, meaning tokens cannot be rescued when the contract is disabled/paused. This is counterintuitive for a rescue function — emergencies requiring pause are exactly when stuck tokens might need recovery.

Scenario:

1. Emergency discovered → admin calls `disable()` to pause migrations

2. Tokens need to be rescued (e.g., accidentally sent or from exploit recovery)

3. `rescue()` reverts due to `onlyEnabled`

4. Admin must re-enable (potentially exposing users to the emergency) just to rescue tokens

Contrast: `setMerkleRoot()` and `setRemainingMintApproval()` correctly removed `onlyEnabled` to allow admin updates while paused.

Recommendation

Remove `onlyEnabled` from `rescue()`. The access control (`onlyAdminOrLegacyMigrationAdmin`) is sufficient protection. This aligns with the pattern used for `setMerkleRoot()` and `setRemainingMintApproval()` which were updated to allow calls while disabled.

Resolution

Olympus Team: The issue was resolved in PR#213.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits