[HashEye]security

SMART CONTRACT SECURITY AUDIT REPORT

E2E TEST — DELETE

Chain: ETHEREUM

Completed: N/A

Report Version: v1.1

PASSED WITH NOTES

hasheye.io | Confidential | 2026

Table of Contents

1 Executive Summary

2 Scope

3 Methodology

4 Risk Overview

5 Findings

6 Summary Table

7 Disclaimer

8 Appendix

Executive Summary

E2E TEST audit — automated Sprint 8.1 verification. One HIGH reentrancy finding in withdraw(). Not a real engagement; safe to delete.

Scope

Project	E2E TEST — DELETE
Chain	ETHEREUM
Contracts Audited	1
Lines of Code	11
Repository / Commit	`N/A @ e2e0000000000000000000000000000000000000`
Files in Scope	test-reentrancy.sol
Audit Start	June 18, 2026
Audit Completion	N/A
Auditors	Admin
Report Version	v1.1

Methodology

HashEye employs a hybrid audit methodology combining automated static analysis with manual expert review.

Automated Analysis

- **Slither:** Static analysis framework for Solidity, detecting common vulnerability patterns
- **Mythril:** Symbolic execution engine for detecting security vulnerabilities
- **Semgrep:** Pattern-matching analysis for language-agnostic vulnerability detection

Manual Review

Our senior security researchers manually review all code in scope with focus on:

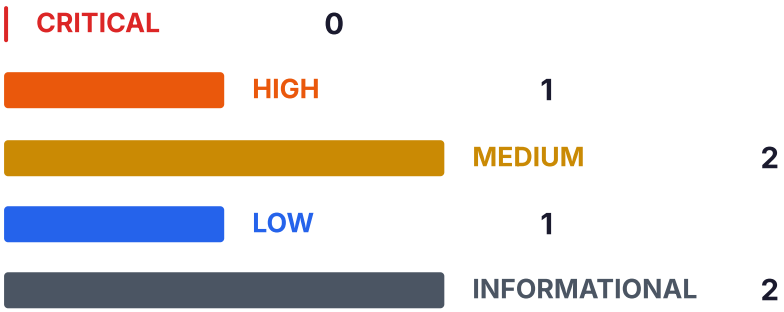
- Business logic correctness
- Access control integrity
- Economic attack vectors (flash loans, oracle manipulation, MEV)
- Upgrade and proxy patterns
- Cross-contract interactions

All automated findings are reviewed and verified by a human auditor before inclusion in this report. AI-assisted analysis is used to generate initial finding descriptions, which are then edited and approved by a senior auditor.

Severity Classification

- **Critical:** Direct loss of funds or complete protocol compromise. Immediate remediation required.
- **High:** Significant risk of fund loss or protocol disruption. Must be fixed before deployment.
- **Medium:** Limited financial risk or operational disruption. Remediation strongly recommended.
- **Low:** Minor issues with negligible financial impact. Improvement recommended.
- **Informational:** Best-practice violations, code quality issues, and non-security observations.

Risk Overview



6 total issues identified.

Findings

H-01 HIGH REENTRANCY OPEN

Reentrancy Eth

DESCRIPTION

A reentrancy vulnerability exists where an external call sending ETH is made before the contract's internal state (such as balances or flags) is updated. This allows a malicious contract receiving ETH to call back into the vulnerable function repeatedly, draining funds before the state reflects the initial withdrawal.

RECOMMENDATION

Apply the checks-effects-interactions pattern by updating all internal state variables before making any external ETH transfers, or use a reentrancy guard modifier (e.g., OpenZeppelin's ReentrancyGuard) to prevent recursive calls.

CLIENT RESPONSE

No response provided.

M-01 MEDIUM VALIDATION OPEN

Transaction Order Dependence

DESCRIPTION

The contract contains operations whose outcomes can be manipulated by the order in which transactions are included in a block. An attacker (or a miner) can observe a pending transaction in the mempool and front-run it by submitting their own transaction with a higher gas price, allowing them to exploit the state change before the original transaction executes.

RECOMMENDATION

Implement a commit-reveal scheme or use a time-lock mechanism to prevent front-running. For token approvals specifically, replace direct `approve()` calls with `increaseAllowance()`/`decreaseAllowance()` patterns, and consider adding slippage protection or minimum/maximum bounds checks to sensitive state-changing functions.

CLIENT RESPONSE

No response provided.

M-02 MEDIUM REENTRANCY OPEN

State access after external call

DESCRIPTION

The contract reads or writes to state variables after making an external call to another contract. This ordering is dangerous because the external contract could call back into this contract (reentrancy) before the state is updated, potentially allowing an attacker to exploit the inconsistent state to drain funds or manipulate logic in unintended ways.

RECOMMENDATION

Apply the checks-effects-interactions pattern by updating all state variables before making any external calls. Additionally, consider adding a reentrancy guard (e.g., OpenZeppelin's ReentrancyGuard) to prevent recursive calls from executing sensitive functions multiple times within the same transaction.

CLIENT RESPONSE

No response provided.

L-01 LOW REENTRANCY OPEN

External Call To User-Supplied Address

DESCRIPTION

The contract makes an external call to an address that is supplied by the user, meaning an attacker could pass in a malicious contract address to hijack the call flow. This creates a reentrancy risk where the malicious contract could call back into the original contract before the first execution completes, potentially exploiting inconsistent state.

RECOMMENDATION

Apply the checks-effects-interactions pattern by updating all state variables before making any external calls, and consider adding a reentrancy guard (e.g., OpenZeppelin's ReentrancyGuard) to critical functions that perform external calls to user-supplied addresses.

CLIENT RESPONSE

No response provided.

I-01 INFORMATIONAL LOGIC OPEN

Solc Version

DESCRIPTION

The contract uses an outdated or unpinned Solidity compiler version, which may expose it to known bugs or inconsistencies across different compiler releases. Using a floating pragma (e.g., ^0.8.x) means the contract could be compiled with unintended versions that introduce breaking changes or unpatched vulnerabilities.

RECOMMENDATION

Pin the Solidity compiler version to a specific, recent stable release (e.g., pragma solidity 0.8.20;) to ensure deterministic compilation and reduce exposure to compiler-related bugs.

CLIENT RESPONSE

No response provided.

I-02 INFORMATIONAL LOGIC OPEN

Low Level Calls

DESCRIPTION

The contract uses low-level calls (e.g., `call`, `delegatecall`, or `staticcall`) which bypass Solidity's type checking and do not automatically revert on failure. If the return value is not properly checked, failed calls may go unnoticed, leading to silent errors or unexpected behavior.

RECOMMENDATION

Replace low-level calls with high-level Solidity function calls wherever possible. If low-level calls are necessary, always check the returned success boolean and handle failure cases explicitly, for example: `(bool success, ) = target.call{...}(...); require(success, 'Call failed');`.

CLIENT RESPONSE

No response provided.

Summary Table

ID	Severity	Category	Title	Status
H-01	HIGH	REENTRANCY	Reentrancy Eth	OPEN
M-01	MEDIUM	VALIDATION	Transaction Order Dependence	OPEN
M-02	MEDIUM	REENTRANCY	State access after external call	OPEN
L-01	LOW	REENTRANCY	External Call To User-Supplied Address	OPEN
I-01	INFORMATIONAL	LOGIC	Solc Version	OPEN
I-02	INFORMATIONAL	LOGIC	Low Level Calls	OPEN
	CRITICAL: 0 | HIGH: 1 | MEDIUM: 2 | LOW: 1 | INFO: 2			**Total: 6**

Disclaimer

This security audit report has been prepared by HashEye for informational purposes only. The audit is not a guarantee that the audited code is free from security vulnerabilities. The audit reflects the state of the code at the specific commit hash identified in the Scope section. Any changes to the code after this commit hash are not covered by this report.

Security audits are inherently limited in scope and cannot guarantee the discovery of all vulnerabilities. This report should not be construed as investment advice, financial advice, or an endorsement of the audited project.

HashEye shall not be held liable for any losses, damages, or claims arising from the use of this report or from the deployment of the audited code. Deployment of smart contracts involves significant financial risk.

This report is confidential and intended solely for the named client. Redistribution is permitted only with prior written consent from HashEye or by mutual agreement for public disclosure.

© 2026 HashEye. All rights reserved.

Appendix

Tool run summaries and raw scanner output are available upon request.