XORS Auditing Report

FOR

$O token contract

https://o1.exchange/

BY

XORS Software

XORS Software
May 2, 2026

► **Prepared For:**

$O token contract
https://o1.exchange/

► **Prepared By:**

Xiangan He & Ahmed Panju

► **Contact Us:** x@xors.xyz

► **Version History:**

Apr. 30, 2026	First Draft
May 2, 2026	Final Draft

Contents

1 Executive Summary

XORS conducted a security assessment of $O token contract combining Slopless automated analysis with manual review and Foundry proof-of-concept exploits.

Summary of issues detected. The assessment uncovered **12 issues, 2 of which are assessed to be of high severity or above**. The severity distribution is as follows:

- **0 Critical-severity issues**
- **2 High-severity issues**
- **2 Medium-severity issues**
- **8 Low-severity issues**

All findings have been remediated and are marked Resolved.

Disclaimer. This report is generated by automated tooling and provides no warranty of any kind, explicit or implied. The contents should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall XORS or any of its employees be liable for any claim, damages or other liability arising from, out of or in connection with the results reported here.

About Slopless. Slopless is XORS's automated security analysis engine for smart contract codebases. It runs a multi-perspective adversarial cognitive loop, surfacing candidate vulnerabilities across reentrancy, access control, arithmetic, business logic, oracle, flash-loan, and upgradeability dimensions. Each candidate is then cross-validated by a prosecutor and defense pair before it is confirmed in the report. Manual review and Foundry proof-of-concept exploits supplement the automated output where deeper protocol-specific reasoning is required.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
$O token contract	HEAD - HEAD	Solidity 0.8.28	EVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engage	Level of Effort
Apr. 29 - May 2, 2026	Slopless + Manual	2	2 person-day

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	2	2
Low-Severity Issues	8	8
Warning-Severity Issues	0	0
Informational-Severity Issues	0	0
TOTAL	12	12

Table 2.4: Category Breakdown.

Name	Number
Security	12

3 Audit Goals and Scope

3.1 Audit Goals

The scan was scoped to provide a comprehensive automated security assessment of $O token contract. The analysis sought to identify security vulnerabilities, code quality issues, and reliability concerns using Slopless static analysis.

3.2 Classification of Vulnerabilities

Findings are classified by severity: Critical, High, Medium, and Low, based on the potential impact and likelihood of exploitation.

Table 3.1: Severity Breakdown.

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

4 Vulnerability Report

In this section, we describe the vulnerabilities found during the Slopless scan. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
XORS-H1	Deployer EOA holds DEFAULT_ADMIN_ROLE plus dual proposer/executor on EnumerableTimelock — full timelock bypass after one round-trip	High	Resolved
XORS-H2	AirdropPool claim accounting (totalClaimed, claimed[]) survives withdraw and updateMerkleRoot — strands tokens for downstream beneficiaries	High	Resolved
XORS-M1	EnumerableTimelock._recordSchedule does not refresh r.delay on cancel + reschedule — stale delay surfaces in opRecord view	Medium	Resolved
XORS-M2	Factory DEFAULT_ADMIN_ROLE granted to timelock — single round-trip permanent bypass via factory.grantRole	Medium	Resolved
XORS-L1	Pauser can permanently freeze any AirdropPool or VestingPool with no recovery path for users	Low	Resolved
XORS-L2	CREATOR_ROLE can deploy a VestingPool with arbitrary beneficiary and schedule — fund-loss contingent on operational funding flow	Low	Resolved
XORS-L3	AirdropPool and VestingPool have no balance check — underfunded pools cause failed-tx UX and wasted gas (atomic revert, no fund loss)	Low	Resolved
XORS-L4	AirdropPool.claim accepts amount = 0 — burns the user's claim slot if a zero-amount leaf exists in the merkle tree	Low	Resolved
XORS-L5	StakePool constructor accepts unlockDelay near type(uint64).max — traps stakers in STAKED state forever	Low	Resolved
XORS-L6	AirdropPool / VestingPool withdraw does not autopause — claimers waste gas on generic SafeERC20 reverts	Low	Resolved
XORS-L7	VestingPool._unlocked() linear scan is unbounded — long schedules can DoS the beneficiary	Low	Resolved
XORS-L8	AirdropPool.updateMerkleRoot accepts bytes32(0) — silently bricks a pool without pausing it	Low	Resolved

4.1 Detailed Description of Issues

4.1.1 XORS-H1: Deployer EOA holds DEFAULT_ADMIN_ROLE plus dual proposer/executor on EnumerableTimelock — full timelock bypass after one round-trip

Severity	High	**Type**	Security
Status	Resolved	**Location**	14
File(s)	ignition/modules/AirdropStack.ts:14		
CWE	CWE-269		

Description. Two compounding misconfigurations on the deployment side collapse the timelock's safety guarantees. First, AirdropStack.ts:26-27 passes [deployer] as both proposers and executors, and parameters.baseSepolia.json sets minDelay to 30 seconds. Second, AirdropStack.ts:14 defaults timelockAdmin to deployer, granting the deployer DEFAULT_ADMIN_ROLE on TimelockController. Because DEFAULT_ADMIN_ROLE is the role-admin of every other timelock role, the deployer can call timelock.grantRole(...) directly with no schedule/execute round-trip. From there, one minDelay cycle is enough to schedule timelock.updateDelay(0) and collapse the delay permanently — every subsequent withdrawPool, updateMerkleRoot, updateSchedule, updateBeneficiary becomes single-block.

Recommendation. Default timelockAdmin to address(0) in AirdropStack.ts so the timelock is self-administered. Operators who want a recovery hatch can override at deploy time. Separately, set minDelay to a production-realistic value (24h+) for non-testnet deployments and keep proposer / executor roles distinct.

4.1.2 XORS-H2: AirdropPool claim accounting (totalClaimed, claimed[]) survives withdraw and updateMerkleRoot — strands tokens for downstream beneficiaries

Severity	High	**Type**	Security
Status	Resolved	**Location**	40
File(s)	contracts/AirdropPool.sol:40		
CWE	CWE-840		

Description. totalClaimed and the claimed[address] flag persist across every admin operation. Two consequences in the docs-recommended operational flow: (1) re-funding the pool against the same root re-uses the prior cap; new beneficiaries hit 'exceeds total allocation' before consuming the new tokens, and the leftovers are stranded inside the pool. (2) Rotating the merkle root leaves every prior claimant flagged, so a user listed in two consecutive trees can only ever claim once, even when totalAllocation has headroom. Distinct from the no-balance-check issue (XORS-L3): even when the pool IS funded, stale accounting blocks valid claims.

Recommendation. Adopt per-epoch claim accounting. Add uint256 public epoch, key claims as _claimedByEpoch[epoch][addr] and _totalClaimedByEpoch[epoch], advance epoch in updateMerkleRoot. Public claimed(addr) and totalClaimed() getters can be preserved for ABI compatibility.

4.1.3 XORS-M1: EnumerableTimelock._recordSchedule does not refresh r.delay on cancel + reschedule — stale delay surfaces in opRecord view

Severity	Medium	**Type**	Security
Status	Resolved	**Location**	133
File(s)	contracts/EnumerableTimelock.sol:133		
CWE	CWE-372		

Description. _recordSchedule writes r.delay only on first sight of an id (inside the if (_idIndex[id] == 0) block). Operation IDs are produced by hashOperation(target, value, data, predecessor, salt) — delay is NOT part of the hash. So a cancelled operation re-scheduled with a different delay updates r.scheduledAt and r.state but leaves r.delay stale. On-chain execution is unaffected because TimelockController._timestamps[id] uses the new delay; only the enumerable view (opRecord(id).delay) returns the wrong value. Frontends and indexers computing 'execute window = scheduledAt + delay' will mis-display the actual ETA.

Recommendation. Move the r.delay = delay assignment outside the if (_idIndex[id] == 0) block so it executes on every schedule call. Storage layout unchanged.

4.1.4 XORS-M2: Factory DEFAULT_ADMIN_ROLE granted to timelock — single round-trip permanent bypass via factory.grantRole

Severity	Medium	**Type**	Security
Status	Resolved	**Location**	31
File(s)	contracts/AirdropFactory.sol:31		
CWE	CWE-269		

Description. Each factory's constructor grants DEFAULT_ADMIN_ROLE to admin_ (the timelock). Even with the timelock self-administered, a proposer can schedule factory.grantRole(ADMIN_ROLE, attackerEOA) via the timelock, wait minDelay, execute. Past that point attackerEOA calls withdrawPool or updateMerkleRoot directly with no further delay — the timelock is permanently bypassed for that factory. The marginal damage over what a compromised proposer can already do via the regular timelock path is saving minDelay on subsequent operations and staying out of the timelock event log.

Recommendation. Operational fix (no code change): after deploy, schedule factory.renounceRole(DEFAULT_ADMIN_ROLE, timelock) via the timelock. This locks role assignments forever — acceptable for fixed-role-set contracts. Add this step to the post-deploy runbook.

4.1.5 XORS-L1: Pauser can permanently freeze any AirdropPool or VestingPool with no recovery path for users

Severity	Low	**Type**	Security
Status	Resolved	**Location**	68
File(s)	contracts/AirdropFactory.sol:68		
CWE	CWE-269		

Description. PAUSER_ROLE (held by an EOA operator) can call pausePool() on any pool instantly. Unpause requires ADMIN_ROLE which is held by the TimelockController, meaning unpause requires scheduling a timelocked operation and waiting for minDelay. A malicious or compromised pauser can re-pause the pool every block, permanently preventing claims or vesting withdrawals. Same asymmetry applies to VestingFactory.sol:69.

Recommendation. Either (a) require a cooldown between pause operations for the same pool, or (b) hold PAUSER_ROLE on a multisig rather than a single EOA. Option (b) is the lower-risk operational change.

4.1.6 XORS-L2: CREATOR_ROLE can deploy a VestingPool with arbitrary beneficiary and schedule — fund-loss contingent on operational funding flow

Severity	Low	**Type**	Security
Status	Resolved	**Location**	39
File(s)	contracts/VestingFactory.sol:39		
CWE	CWE-269		

Description. VestingFactory.createVestingPool accepts arbitrary beneficiary and schedule from the caller. CREATOR_ROLE can therefore deploy a pool with an attacker-controlled beneficiary and an instant-unlock schedule. Important caveat: the constructor does not transfer tokens, so the deployed pool starts with zero balance. Actual fund loss requires a separate funding step by another party. Real risk only if the operations process auto-funds creator-deployed pools without manual review.

Recommendation. Document the funding-flow assumption explicitly in the deployment runbook: newly created pools must be reviewed before any token transfer. Alternatively, gate funding to a separate role that performs schedule and beneficiary checks at the time of funding.

4.1.7 XORS-L3: AirdropPool and VestingPool have no balance check — underfunded pools cause failed-tx UX and wasted gas (atomic revert, no fund loss)

Severity	Low	**Type**	Security
Status	Resolved	**Location**	40
File(s)	contracts/AirdropPool.sol:40		
CWE	CWE-754		

Description. AirdropPool.claim() checks totalClaimed + amount <= totalAllocation but never verifies that the contract's actual token balance is sufficient. VestingPool.claim() has the same gap. Factory createPool functions deploy without funding the pool — funding is entirely off-chain. If a pool is underfunded, claims that pass the allocation check will revert on safeTransfer, but because the revert is atomic, claimed[] and totalClaimed are rolled back together with the failed transfer; no slot is 'burnt'. Impact is therefore wasted gas and unclear failure mode for users, not lost tokens.

Recommendation. Either add require(token.balanceOf(address(this)) >= amount, "insufficient balance") at the top of claim(), or require pools to be fully funded before allowing claims (track expectedBalance at createPool, gate claim() on it). Coupled with the auto-pause-on-withdraw fix below, the failure mode becomes self-explaining.

4.1.8 XORS-L4: AirdropPool.claim accepts amount = 0 — burns the user's claim slot if a zero-amount leaf exists in the merkle tree

Severity	Low	**Type**	Security
Status	Resolved	**Location**	40
File(s)	contracts/AirdropPool.sol:40		
CWE	CWE-754		

Description. AirdropPool.claim() does not require amount > 0. If a merkle tree contains a leaf for (user, 0), the user can call claim(0, proof) which sets claimed[msg.sender] = true and transfers zero tokens. The user's claim slot is permanently consumed. If the tree is later updated to include a non-zero allocation for the same address, they cannot claim because the flag persists across root updates. Limited exploit surface: the leaf hash binds msg.sender, so an attacker cannot burn another user's slot. Realistic exposure requires either (a) a tree generator bug producing zero-amount leaves, or (b) a malicious tree creator targeting specific addresses.

Recommendation. Add require(amount > 0, "amount zero") at the top of claim(). Trivial one-line change with no downstream impact.

4.1.9 XORS-L5: StakePool constructor accepts unlockDelay near type(uint64).max — traps stakers in STAKED state forever

Severity	Low	**Type**	Security
Status	Resolved	**Location**	32
File(s)	contracts/StakePool.sol:32		
CWE	CWE-190		

Description. StakePool.unstake() emits Unstaked(..., nowTs + unlockDelay) where both operands are uint64. If a deployer sets unlockDelay near type(uint64).max, the addition overflows in checked arithmetic and unstake() reverts forever — stakers are stuck in STAKED state with no path to UNSTAKING and therefore no path to withdraw. Because StakePool has no admin, pause, or rescue, the lockup is permanent.

Recommendation. Cap unlockDelay in the constructor. A reasonable bound is e.g. MAX_UNLOCK_DELAY = 365 days * 10 — well above realistic unbonding, well below the overflow threshold.

4.1.10 XORS-L6: AirdropPool / VestingPool withdraw does not auto-pause — claimers waste gas on generic SafeERC20 reverts

Severity	Low	**Type**	Security
Status	Resolved	**Location**	68
File(s)	contracts/AirdropPool.sol:68		
CWE	CWE-754		

Description. Admin can call withdrawPool to drain a pool and the underlying paused flag stays false. A user whose claim() lands immediately after gets a generic 'ERC20: transfer amount exceeds balance' revert — they can't tell whether the pool was drained, mid-migration, or has a token bug, and retries keep burning gas on a guaranteed-revert path. Same issue in VestingPool.sol:105.

Recommendation. Set paused = true (and emit Paused) inside withdraw(). Re-opening claims requires explicit unpause — confirms intent before re-opening and gives users a clean signal. Manual PoC at test/foundry/poc/L02_WithdrawAutoPause.t.sol.

4.1.11 XORS-L7: VestingPool._unlocked() linear scan is unbounded — long schedules can DoS the beneficiary

Severity	Low	**Type**	Security
Status	Resolved	**Location**	56
File(s)	contracts/VestingPool.sol:56		
CWE	CWE-1284		

Description. _unlocked() walks the entire schedule[] array on every claim() and claimable() call. _validateSchedule enforces no length cap. A factory creator who supplies a 50,000-point schedule (whether maliciously or from a buggy off-chain process) makes every claim and claimable call exhaust the block gas limit. Each iteration costs roughly 4,200 gas (two cold SLOADs); a schedule of 7,000-plus points exhausts a 30M gas budget.

Recommendation. Either cap schedule length in _validateSchedule (e.g. <= 256), or replace the linear scan with a binary search over schedule[]. The cap is the simpler remediation.

4.1.12 XORS-L8: AirdropPool.updateMerkleRoot accepts bytes32(0) — silently bricks a pool without pausing it

Severity	Low	**Type**	Security
Status	Resolved	**Location**	74
File(s)	contracts/AirdropPool.sol:74		
CWE	CWE-20		

Description. Setting merkleRoot to bytes32(0) makes every proof verification fail with 'invalid proof', but the paused flag stays false. Users keep submitting claims, paying gas to revert, and have no clean signal that the pool is decommissioned.

Recommendation. Add require(newRoot != bytes32(0), "root zero") in updateMerkleRoot. If a true decommission is intended, prefer setting paused = true instead.