HackenProof

Expert crowdsourced security provider

REPORT

Private Security Audit

for **Algoxen**

PREPARED BY

HackenProof

REPORT CREATED

11.05.2026

AUDIT DURATION

29.04-09.05.2026

AUDIT TYPE

Solidity

TABLE OF CONTENTS

HackenProof | ALGOXEN

INTRODUCTION

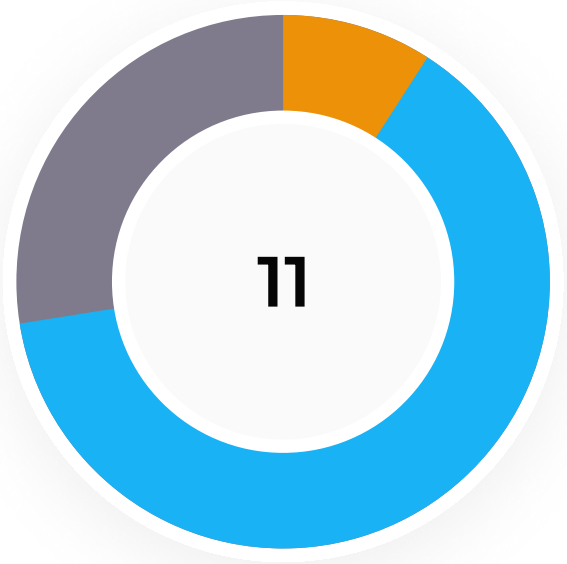
HackenProof Audit is your solution to maximize cybersecurity through the involvement of the industry's most skilled and experienced community security researchers.

Name	Algoxen
Website	https://algoxen.com/
Type	Private Security Audit
Date	29.04-09.05.2026
Changelog	11.05.2026 - Final Report
Scope	https://github.com/hackenproof-public/algoxen-m...
Approved by	HackenProof team
Audited by	 @shealtielanz  @0xBugSlayer  @0xaudron

HackenProof | ALGOXEN

AUDIT SUMMARY

11	8	3
Total Findings	Resolved	Informative



Critical	0
High	0
Medium	1
Low	7
None	3

FINDINGS LIST

MEDIUM

ALGOXEN-15 TokenVesting `burnExcess` callable without a schedule destroys all pre-funded AGX	Resolved

LOW

ALGOXEN-18 Missing event on Crowdsale tier advance	Resolved
ALGOXEN-17 RewardClaims `setWindowActive` rejects post-sweep reactivation against spec	Resolved
ALGOXEN-16 TokenVesting `setSchedule` accepts tiers with `bps == 0`	Resolved
ALGOXEN-14 Floating Pragma	Resolved

ALGOXEN-12	Ownable single-step transfer risks permanent ownership loss	Resolved
ALGOXEN-8	`quoteAgx()` returns zero where `buy()` reverts	Resolved
ALGOXEN-6	Tier 0 can start at deployment timestamp despite spec requiring strictly-after-deployment	Resolved

- NONE

ALGOXEN-5	Buyers cannot protect against execution in a later, worse-priced tier	Informative
ALGOXEN-9	Hardening Notes.	Informative
ALGOXEN-7	Duplicate same-token merkle roots can create accidental duplicate reward obligations	Informative

VULNERABILITY DETAILS

ALGOXEN-15 TokenVesting `burnExcess` callable without a schedule destroys all pre-funded AGX

• MEDIUM

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • MEDIUM

Vulnerability details

As per vesting-spec.md, the deployment ordering is funding-then-schedule then ICO. The spec states "Before the ICO starts, the owner must fund the vesting contract with AGX tokens and set the vesting schedule" and "If the vesting schedule is not set, the first purchase will revert." The vesting spec also constrains burnExcess only by ICO end timestamp; no schedule precondition.

The implementation gates only on block.timestamp > icoEndTimestamp and ! excessBurnExecuted. It does not require a schedule to be set.

```
// TokenVesting.sol
function burnExcess() external {
    if (block.timestamp <= icoEndTimestamp) revert IcoNotEnded();
    if (excessBurnExecuted) revert ExcessBurnAlreadyExecuted();
    excessBurnExecuted = true;

    uint256 owed = totalEntitled - totalClaimed;
    uint256 balance = token.balanceOf(address(this));
    uint256 amount = balance > owed ? balance - owed : 0;
    if (amount > 0) token.burn(amount);

    emit ExcessBurned(amount);
}
```

If the operator funds the vesting contract but setSchedule has not been called before the natural ICO end, totalEntitled is zero and owed is zero. Any unprivileged caller invoking burnExcess after icoEndTimestamp burns the entire balance.

The destructive action is taken by an unprivileged caller. The operator's setup error becomes irreversible the moment a third party fires burnExcess. excessBurnExecuted is then set, permanently closing entitlement registration through the RegistrationClosed guard. The pre-funded treasury AGX is destroyed and the contract is bricked.

The same risk class is already guarded in TokenVesting::updateIcoEndTimestamp and it rejects calls when no schedule is set:

```
if (_schedule.length == 0) revert ScheduleNotSet();
```

The author's own NatSpec in TokenVesting.sol:222-223 explains why: "Reverts if no schedule has been set, which would otherwise let a misfired emergency stop brick the contract." TokenVesting::burnExcess is destructive in exactly the same state, but the equivalent guard is missing — that omission is the bug.

Validation steps

1. Crowdsale is deployed which auto-deploys TokenVesting.
2. Operator transfers AGX directly to the TokenVesting address as funding.
3. The schedule has not been set.
4. block.timestamp > icoEndTimestamp arrives.
5. Any address calls burnExcess(). The full pre-funded balance is burned.

PoC:

```
function test_M01_burnExcess_without_schedule_burns_treasury() public {
    TokenVesting vesting = crowdsale.vesting();
    agx.mint(address(vesting), 1_000_000e18);

    vm.warp(uint256(crowdsale.icoEndTimestamp()) + 1);

    address randomCaller = address(0xBEEF);
    vm.prank(randomCaller);
    vesting.burnExcess();

    assertEq(agx.balanceOf(address(vesting)), 0);
    assertTrue(vesting.excessBurnExecuted());
}
```

Recommendation:

```
  function burnExcess() external {
+     if (_schedule.length == 0) revert ScheduleNotSet();
      if (block.timestamp <= icoEndTimestamp) revert IcoNotEnded();
      if (excessBurnExecuted) revert ExcessBurnAlreadyExecuted();
      ...
  }
```

Comments history

ALGOXEN This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-18 Missing event on Crowdsale tier advance

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

Crowdsale::buy advances the next tier's start to block.timestamp on sell-out. The mutation has no event. TokensPurchased carries no tier information. Off-chain indexers cannot detect the tier transition from
logs. TokenVesting::updateIcoEndTimestamp emits IcoEndUpdated for an equivalent timestamp mutation. The Crowdsale::buy advance is the only state write in the contract that goes unlogged.

Validation steps

Recommendation:

Emit a dedicated event on the mutation branch.

```
+ event TierAdvanced(uint8 indexed soldOutTier, uint8 indexed activatedTier, uint64 newStart);
```

Comments history

ALGOXEN This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-17 RewardClaims `setWindowActive` rejects post-sweep reactivation against spec

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

As per the RewardClaims spec, it permits setWindowActive on swept windows. However there is a divergence as the contract rejects it.

The spec explicitly authorises post-sweep calls:

- "setWindowActive is permitted after expiresAt and after a sweep; it only updates the active flag."

And fixes the rejection set:

- "Checks run in the order owner → configured → no-op."

Three checks, in order: owner, configured, no-op short-circuit. Nothing else.

The contract adds a fourth check:

```
function setWindowActive(bytes32 claimWindowId, bool active) external onlyOwner
{
    Window storage w = _windows[claimWindowId];
    if (w.token == address(0)) revert WindowNotConfigured();
    if (active && w.swept) revert WindowAlreadySwept();   // not in spec
    if (w.active == active) return;
    w.active = active;
    emit WindowActiveSet(claimWindowId, active);
}
```

Calling setWindowActive(id, true) on a swept window reverts. But the spec authorises that call.

Validation steps

Recommendation:

Add consistency by dropping the extra swept check so the function matches the spec :

```
 function setWindowActive(bytes32 claimWindowId, bool active) external
onlyOwner {
    Window storage w = _windows[claimWindowId];
    if (w.token == address(0)) revert WindowNotConfigured();
-   if (active && w.swept) revert WindowAlreadySwept();
    if (w.active == active) return;
    w.active = active;
    emit WindowActiveSet(claimWindowId, active);
 }
```

Comments history

ALGOXEN This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-16 TokenVesting `setSchedule` accepts tiers with `bps == 0`

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

The vesting spec defines a tier's percentage as "Percentage of entitlement released" and its timestamp as "Date/time at which this tranche becomes claimable." Each tier is a release event. However a bps == 0 tier releases nothing and is accepted, diverging with the spec.

The implementation validates only the cumulative sum:

```
// contracts/src/TokenVesting.sol:122-133
sum += t.bps;
_schedule.push(t);
...
if (sum != 10_000) revert BpsSumInvalid();
```

Two consequences of accepting bps == 0:

1. The tier counts toward _schedule.length despite having no release.
2. The tier's timestamp can shift claimDeadline() (which reads the last tier's timestamp). A zero-bps tier placed near type(uint64).max - 365 days overflows claimDeadline(), bricking burnUnclaimed().

Both require an owner mistake on setSchedule. The check makes the spec's per-tier release semantics enforced at the contract layer rather than left implicit.

Validation steps

Recommendation:

Add the following validation :

```
if (t.bps == 0) revert ZeroBps();
```

Comments history

ALGOXEN	This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-14 Floating Pragma

• LOW

Target	https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src
Status:	Resolved
Severity:	• LOW

Vulnerability details

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., ^0.8.28) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The CrowdSale, RewardClaims, Staking, Token, TokenVesting contracts currently use floating pragma declarations ^0.8.28. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Validation steps

It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using ^0.8.xx, explicitly define the version with pragma solidity 0.8.xx.

Comments history

ALGOXEN	This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-12 Ownable single-step transfer risks permanent ownership loss

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

The contracts (Crowdsale, TokenVesting, Staking, RewardClaims) inherit OZ Ownable. So the function transferOwnership(newOwner) instantly assigns ownership without recipient confirmation, so a typo, frozen wallet, or non-callable contract permanently bricks every owner-only entry point. The contract is non-upgradeable, so recovery is impossible.

Validation steps

Recommendation:

Switch every owned contract to Ownable2Step for enhanced security. Apply identically to TokenVesting, Staking, RewardClaims.

Comments history

ALGOXEN This has now been fixed in the latest GitHub commit.

VULNERABILITY DETAILS

ALGOXEN-8 `quoteAgx()` returns zero where `buy()` reverts

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

Summary: quoteAgx(paymentAmount) claims to return what buy(paymentAmount) would deliver, but for very small payments it returns 0 while buy() reverts with ZeroAgxAmount.

Affected Code:

- contracts/src/Crowdsale.sol:161-172
- contracts/src/Crowdsale.sol:185-191
- contracts/docs/ico-spec.md:31

Code Details: The quote function returns the truncated value:

```
// contracts/src/Crowdsale.sol:166-172
function quoteAgx(uint256 paymentAmount) external view returns (uint256) {
    (uint8 index, bool found) = _activeTier();
    if (!found) revert NoActiveTier();
    SaleTier storage t = _tiers[index];
    uint256 tokenAmount = paymentAmount * 1e18 / t.price;
    uint256 remaining = uint256(t.allocation - t.sold);
    return tokenAmount > remaining ? remaining : tokenAmount;
}
```

The actual purchase path rejects zero output:

```
// contracts/src/Crowdsale.sol:185-191
function buy(uint256 paymentAmount) external whenNotPaused {
    (uint8 index, bool found) = _activeTier();
    if (!found) revert NoActiveTier();
    SaleTier storage tier = _tiers[index];

    uint256 tokenAmount = paymentAmount * 1e18 / tier.price;
    if (tokenAmount == 0) revert ZeroAgxAmount();
```

The ICO spec agrees with the runtime behavior:

```
contracts/docs/ico-spec.md:31
A purchase is rejected if the resulting token amount would be zero.
```

Failure Scenario: Assume price = 2e18 micro-USDC per AGX. A caller queries:

```
quoteAgx(1) = 1 * 1e18 / 2e18 = 0
```

The view returns 0. If the same caller submits buy(1), the transaction reverts with ZeroAgxAmount.

Impact: No funds are at risk. The issue affects frontend and integration reliability because a view function that claims to mirror buy() does not mirror the revert behavior.

Recommendation

Make the view match runtime behavior:

```
uint256 tokenAmount = paymentAmount * 1e18 / t.price;
if (tokenAmount == 0) revert ZeroAgxAmount();
```

Alternatively, update NatSpec to say that quoteAgx() returns the raw output amount and callers must reject zero themselves. Matching runtime is cleaner.

Validation steps

1. Deploy a Crowdsale with an active tier whose price is greater than paymentAmount * 1e18. For example:

```
paymentAmount = 1
price = 2e18
```

2. Call quoteAgx(1).
3. Confirm it returns 0 because contracts/src/Crowdsale.sol:170 computes:

```
1 * 1e18 / 2e18 = 0
```

4. Call buy(1).
5. Confirm it reverts with ZeroAgxAmount at contracts/src/Crowdsale.sol:191.
6. This validates that the view and runtime paths diverge for the same paymentAmount.

Comments history

ALGOXEN Thanks! Valid find.

VULNERABILITY DETAILS

ALGOXEN-6 Tier 0 can start at deployment timestamp despite spec requiring strictly-after-deployment

• LOW

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Resolved

Severity: • LOW

Vulnerability details

Summary: The Crowdsale spec says tier 0 must start strictly after deployment. The constructor only rejects tiers_[0].start < block.timestamp, allowing tiers_[0].start == block.timestamp. This allows the sale to be live in the deployment block.

Affected Code:
- contracts/src/Crowdsale.sol:82
- contracts/docs/ico-spec.md:50

Code Details: Implementation:

```
// contracts/src/Crowdsale.sol:82
if (tiers_[0].start < block.timestamp) revert IcoStartInPast();
```

Spec:

```
contracts/docs/ico-spec.md:50
Tier 0's start time must be strictly after deployment — the constructor rejects a tier 0 that is already live or in the past.
```

The implementation allows equality, but equality is "already live" because _activeTier() accepts block.timestamp >= t.start:

```
// contracts/src/Crowdsale.sol:265-266
if (block.timestamp >= t.end) continue;
if (block.timestamp >= t.start && t.sold < t.allocation) return (i, true);
```

Failure Scenario:
1. Deployer deploys Crowdsale at timestamp T.
2. tiers_[0].start is also T.
3. Constructor accepts the tier because T < T is false.
4. In the same block, _activeTier() treats tier 0 as active because block.timestamp >= t.start.

Impact: This is a clean spec mismatch and integration risk. A sale can become active immediately at deployment even though the documented lifecycle says tier 0 starts strictly after deployment. Watchers, frontends, and operational runbooks may assume a post-deployment setup/announcement window that the contract does not enforce.

Recommendation

Change the constructor check to reject equality:

```
if (tiers_[0].start <= block.timestamp) revert IcoStartInPast();
```

Add a boundary test:

```
function test_constructor_revertsWhenTier0StartEqualsBlockTimestamp() public;
```

Validation steps

1. In a Foundry test, set block.timestamp = T.
2. Construct tiers_[0].start = T and tiers_[0].end = T + 1 days.
3. Deploy Crowdsale.
4. Confirm deployment succeeds because contracts/src/Crowdsale.sol:82 only checks < block.timestamp, and equality does not revert.
5. Immediately call currentTier().
6. Confirm it returns tier 0 because _activeTier() uses block.timestamp >= t.start at contracts/src/Crowdsale.sol:266.
7. After applying the fix (<=), repeat the same test and confirm deployment reverts with IcoStartInPast.

Comments history

ALGOXEN Thanks! Valid find.

VULNERABILITY DETAILS

ALGOXEN-5 Buyers cannot protect against execution in a later, worse-priced tier

• NONE

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Informative

Severity: • NONE

Vulnerability details

Summary: Crowdsale.buy(paymentAmount) always executes against the tier active at transaction execution time. The buyer cannot specify minAgxOut, expectedTier, or maxPrice. If the active tier changes between transaction signing and execution, the buyer can receive materially fewer AGX than expected while still spending the same USDC amount.

Affected Code:

- contracts/src/Crowdsale.sol:185-190 resolves the tier at execution time.
- contracts/src/Crowdsale.sol:204-207 advances the next tier immediately when the current tier sells out.
- contracts/docs/ico-spec.md:8 documents immediate next-tier activation on early sell-out.

Code Details: The active tier is selected inside buy() at execution time:

```
// contracts/src/Crowdsale.sol:185-190
function buy(uint256 paymentAmount) external whenNotPaused {
    (uint8 index, bool found) = _activeTier();
    if (!found) revert NoActiveTier();
    SaleTier storage tier = _tiers[index];

    uint256 tokenAmount = paymentAmount * 1e18 / tier.price;
```

When a tier sells out, the next tier can start immediately:

```
// contracts/src/Crowdsale.sol:204-207
if (tier.sold == tier.allocation && index + 1 < _tiers.length) {
    SaleTier storage next = _tiers[index + 1];
    if (next.start > block.timestamp) next.start = uint64(block.timestamp);
}
```

There is no buyer-provided guard after the tier is resolved:

- no minAgxOut;
- no expectedTier;
- no maxPrice;
- no deadline.

Failure Scenario: Assume:

- Tier 0 price: 100,000 micro-USDC per AGX.
- Tier 1 price: 150,000 micro-USDC per AGX.
- Bob signs buy(1,000e6) while tier 0 is active.

Expected by Bob:

```
1,000 USDC / 0.10 USDC = 10,000 AGX
```

Actual ordering:

1. Alice's transaction executes first and sells out tier 0.
2. Crowdsale advances tier 1 start to block.timestamp.
3. Bob's transaction executes after Alice's transaction.
4. _activeTier() now returns tier 1.
5. Bob receives:

```
1,000 USDC / 0.15 USDC = 6,666.666... AGX
```

Bob receives roughly 33.33% fewer AGX than expected, and the transaction does not revert.

Impact: This is not direct theft and does not corrupt accounting. However, it is a meaningful buyer-protection issue. Normal mempool ordering, same-block sequencing, or deliberate tier-fill transactions can cause users to execute in a worse tier than the one they intended.

The issue is more important because tier transitions can happen by allocation sell-out, not only by time.

Recommendation

Add buyer-side execution guards. This changes the ABI unless implemented as an overload.

```
function buy(
    uint256 paymentAmount,
    uint256 minAgxOut,
    uint8 expectedTier
) external;
```

Then:

```
(uint8 index, bool found) = _activeTier();
if (!found) revert NoActiveTier();
if (index != expectedTier) revert UnexpectedTier();

...

if (tokenAmount < minAgxOut) revert SlippageExceeded();
```

ABI-preserving alternative:

```
function buyWithGuard(
    uint256 paymentAmount,
    uint256 minAgxOut,
    uint8 expectedTier
) external;
```

Keep buy(uint256) for existing integrations but have frontends use the guarded version.

Validation steps

1. Deploy a Crowdsale with two contiguous active/future tiers:

- tier 0 price: 100_000 micro-USDC per AGX;
- tier 1 price: 150_000 micro-USDC per AGX;
- tier 0 has only a small remaining allocation.

2. Have Bob compute or observe the expected tier 0 output for buy(1_000e6):

```
1,000e6 * 1e18 / 100,000 = 10,000e18 AGX
```

3. Before Bob's transaction executes, have Alice call buy() with enough USDC to fill tier 0.

4. Confirm contracts/src/Crowdsale.sol:204-207 advances tier 1's start to block.timestamp.

5. Execute Bob's original buy(1_000e6).

6. Confirm Bob's entitlement is computed using tier 1:

```
1,000e6 * 1e18 / 150,000 = 6,666.666...e18 AGX
```

7. Confirm there is no revert path based on Bob's expected tier or minimum output, because buy() has only the paymentAmount argument at contracts/src/Crowdsale.sol:185.

Executable PoC

An executable Foundry PoC is included at contracts/test/audit/AuditPoC.t.sol:358-430:

```
AuditPoC.test_FINDING7_no_slippage_buyer_lands_in_worse_tier
```

The PoC sets tier 0 to 0.10 USDC / AGX with only 100 AGX available, and tier 1 to 0.20 USDC / AGX. A frontrunner first drains tier 0, which pulls tier 1's start to the current block. Alice then executes buy(10e6) that would have delivered 100 AGX in tier 0, but because the active tier changed, she receives only 50 AGX and the transaction succeeds.

PoC code:

```
function test_FINDING7_no_slippage_buyer_lands_in_worse_tier() public {
    MockERC20 usdc = new MockERC20("USDC", "USDC");
    MockAGX agx = new MockAGX();

    Crowdsale.SaleTier[4] memory tiers;
    uint64 t0 = uint64(block.timestamp + 1);

    // Tier 0: 0.10 USDC / AGX, only 100 AGX available.
    tiers[0] = Crowdsale.SaleTier({
        start: t0,
        end: t0 + 1 days,
        price: uint128(0.1e6),
        allocation: uint128(100e18),
        sold: 0
    });

    // Tier 1: 0.20 USDC / AGX.
    tiers[1] = Crowdsale.SaleTier({
        start: t0 + 1 days,
        end: t0 + 2 days,
        price: uint128(0.2e6),
        allocation: uint128(1_000e18),
        sold: 0
    });

    tiers[2] = Crowdsale.SaleTier({
        start: t0 + 2 days,
        end: t0 + 3 days,
        price: uint128(0.3e6),
        allocation: uint128(1_000e18),
        sold: 0
    });

    tiers[3] = Crowdsale.SaleTier({
        start: t0 + 3 days,
        end: t0 + 4 days,
        price: uint128(0.4e6),
        allocation: uint128(1_000e18),
        sold: 0
    });

    Crowdsale cs = new Crowdsale(
        owner,
        IERC20(address(usdc)),
        IERC20(address(agx)),
        owner,
        tiers
    );

    // Fund vesting and set a schedule so `buy()` can register entitlements.
    TokenVesting v = cs.vesting();
    TokenVesting.Tier[] memory sched = new TokenVesting.Tier[](1);
    sched[0] = TokenVesting.Tier({timestamp: t0 + 5 days, bps: 10_000});
    vm.prank(owner);
    v.setSchedule(sched);
    agx.mint(address(v), 100_000e18);
```

```
    vm.warp(t0 + 1);

    // Frontrunner drains tier 0. This pulls tier 1's start forward to now.
    address frontrunner = address(0xF20);
    usdc.mint(frontrunner, 100e6);
    vm.startPrank(frontrunner);
    usdc.approve(address(cs), type(uint256).max);
    cs.buy(cs.maxPurchaseUsdc());
    vm.stopPrank();

    assertEq(uint256(cs.currentTier()), 1, "tier 1 active after frontrun");

    // Alice expected tier 0 pricing:
    // 10 USDC / 0.10 USDC per AGX = 100 AGX.
    //
    // But she has no minAgxOut/expectedTier guard, so her transaction
    // executes in tier 1:
    // 10 USDC / 0.20 USDC per AGX = 50 AGX.
    usdc.mint(alice, 10e6);
    vm.startPrank(alice);
    usdc.approve(address(cs), 10e6);
    cs.buy(10e6);
    vm.stopPrank();

    uint256 expectedAtTier0 = 100e18;
    uint256 deliveredAtTier1 = 50e18;

    assertEq(v.totalEntitlement(alice), deliveredAtTier1);
    assertLt(v.totalEntitlement(alice), expectedAtTier0);
}
```

Run from contracts/:

```
RPC_URL=https://ethereum-rpc.publicnode.com forge test \
  --match-path test/audit/AuditPoC.t.sol \
  --match-test test_FINDING7_no_slippage_buyer_lands_in_worse_tier \
  -vv
```

Expected result:

```
Ran 1 test for test/audit/AuditPoC.t.sol:AuditPoC
[PASS] test_FINDING7_no_slippage_buyer_lands_in_worse_tier()
```

Comments history

ALGOXEN Thanks for the finding! Will need to consider whether we want to add such protection functionality.

VULNERABILITY DETAILS

ALGOXEN-9 Hardening Notes.

• NONE

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Informative

Severity: • NONE

Vulnerability details

H-1: **Add a maximum staking lock duration**

Affected: contracts/src/Staking.sol:78-84, contracts/src/Staking.sol:139-148

The constructor accepts any nonzero uint64 lock duration:

```
// contracts/src/Staking.sol:81-83
for (uint256 i = 0; i < lockDurations_.length; ++i) {
    if (lockDurations_[i] == 0) revert ZeroDuration();
    _lockDurations.push(lockDurations_[i]);
}
```

stake() later adds that duration to uint64(block.timestamp):

```
// contracts/src/Staking.sol:148
uint64 unlockTs = uint64(block.timestamp) + _lockDurations[lockIndex];
```

A near-type(uint64).max duration makes staking with that lock index revert due to checked arithmetic. This is owner/deployer misconfiguration, not an attacker path.

Validation: Deploy Staking with lockDurations_[0] = type(uint64).max. Then call stake(amount, 0) at any realistic timestamp. The addition at contracts/src/Staking.sol:148 overflows and reverts under Solidity checked arithmetic.

Fix: Add a sanity bound such as:

```
uint64 constant MAX_LOCK = 10 * 365 days;
if (lockDurations_[i] > MAX_LOCK) revert LockDurationTooLong();
```

H-2: **Disable renounceOwnership()**

Affected: all Ownable contracts:

- contracts/src/Crowdsale.sol
- contracts/src/TokenVesting.sol
- contracts/src/Staking.sol
- contracts/src/RewardClaims.sol

OZ v5 Ownable exposes renounceOwnership(). If the multisig accidentally renounces before lifecycle completion, owner-only controls become permanently unreachable:

- Crowdsale.emergencyStop(), setBeneficiary(), recoverToken()
- TokenVesting.setSchedule(), recoverToken()
- Staking.pause(), unpause(), recoverToken()
- RewardClaims.configureWindow(), setWindowActive(), sweepExpiredWindow(), recoverToken()

Validation: In a test, call inherited renounceOwnership() from the current owner, then assert owner() == address(0) and that any representative onlyOwner function reverts for the previous owner. Repeat for each contract inheriting Ownable.

Fix:

```
error RenounceDisabled();

function renounceOwnership() public view override onlyOwner {
    revert RenounceDisabled();
}
```

H-3: Add deployment preflights for token shape

Affected: contracts/src/Crowdsale.sol:71-101

The sale assumes:

- usdc_ is a 6-decimal USDC-shaped token;
- agx_ is the canonical burnable AGX token.

The constructor only checks nonzero addresses:

```
// contracts/src/Crowdsale.sol:78-80
if (address(usdc_) == address(0)) revert ZeroAddress();
if (address(agx_) == address(0)) revert ZeroAddress();
if (beneficiary_ == address(0)) revert ZeroAddress();
```

The vesting deployment force-casts agx_ to ERC20Burnable:

```
// contracts/src/Crowdsale.sol:101
vesting = new TokenVesting(owner_, ERC20Burnable(address(agx_)), address(this),
tiers_[tiers_.length - 1].end);
```

Under the trust assumptions this is fine, but a bad deployment address can break pricing or the burn lifecycle.

Validation: Deploy Crowdsale with a mock 18-decimal payment token and observe that pricing still treats paymentAmount as micro-USDC. Separately, deploy with a non-burnable ERC20 as agx_; deployment succeeds because of the address cast, but vesting burn calls later revert when token.burn(amount) is attempted.

Fix: In deployment scripts, verify canonical Arbitrum USDC, canonical AGX, decimals, and a burn lifecycle smoke test. Constructor type changes improve clarity but cannot fully prove runtime token behavior.

H-4: **Add RewardClaims operator preflights**

Affected: contracts/src/RewardClaims.sol:124-151, contracts/src/RewardClaims.sol:195-204

The spec deliberately allows configure-before-fund:

> contracts/docs/reward-claims-spec.md:11
> configureWindow does not check the contract's balance against totalAmount... this ordering is deliberate.

This is spec-compliant, but operators should preflight:

- balanceOf(rewardClaims) >= _tokenReserved[token] + totalAmount
- totalAmount == sum(leaves.amount)
- no duplicate leaf indices
- no accidental duplicate (token, merkleRoot)

Validation: Configure a window with totalAmount greater than the contract's token balance. The call succeeds by design at contracts/src/RewardClaims.sol:124-151; later claim() can fail on token transfer if unfunded, and recoverToken() reverts if balance < _tokenReserved[token] at contracts/src/RewardClaims.sol:198-201.

This prevents most operational mistakes without changing contract semantics.

H-5: **Add staking pagination or a frontend/indexer policy**

Affected: contracts/src/Staking.sol:125-132

positionsOf(account) returns every position ever opened by an account:

```
// contracts/src/Staking.sol:125-132
function positionsOf(address account) external view returns (uint256[] memory) {
    return _userPositions[account];
}
```

The list is append-only by design. A heavy user can make their own reads large.

Validation: In a test, call stake(1, lockIndex) repeatedly from the same account, withdrawing matured positions if desired. Then call positionsOf(account) and confirm the returned array length equals every position ever opened, including withdrawn positions, because IDs are pushed at contracts/src/Staking.sol:157 and never removed.

Fix: Keep the pinned ABI, but add optional helper views:

```
function positionCount(address account) external view returns (uint256);
function positionOfAt(address account, uint256 index) external view returns (uint256);
```

H-6: **Document post-burnExcess AGX transfers more loudly**

Affected: contracts/src/TokenVesting.sol:235-246, contracts/src/TokenVesting.sol:271-275, contracts/docs/vesting-spec.md:92-94
recoverToken() rejects the vesting token:

```
// contracts/src/TokenVesting.sol:271-275
function recoverToken(IERC20 other, address to, uint256 amount) external
onlyOwner {
    if (address(other) == address(token)) revert CannotRecoverVestingToken();
    if (to == address(0)) revert ZeroAddress();
    if (amount == 0) revert ZeroAmount();
    other.safeTransfer(to, amount);
```

AGX sent after burnExcess() cannot be recovered and remains until burnUnclaimed(). This is by design, but it deserves loud operational documentation.

Validation: After burnExcess() has executed, transfer additional AGX directly to TokenVesting. Confirm recoverToken(AGX, to, amount) reverts with CannotRecoverVestingToken, and burnExcess() cannot be called again because excessBurnExecuted is already true. The AGX remains until burnUnclaimed() becomes callable.

Comments history

ALGOXEN

Notes:

- H1: As noted, this is not an attack vector. We will consider whether it's worth it to add this guard.
- H2: I feel this is not reasonable to change. The team may choose to renounce ownership to indicate that no changes can be made to the contracts.
- H3: I feel this is not reasonable to change: verifying deployment script's parameter on-the-fly to make sure the chosen token is burnable is not trivial and makes the deployment more complicated.
- H4: Good stuff.
- H5: Good stuff.
- H6: Ok

VULNERABILITY DETAILS

ALGOXEN-7 Duplicate same-token merkle roots can create accidental duplicate reward obligations

• NONE

Target https://github.com/hackenproof-public/algoxen-main/tree/752408eaf05ef63d4281d7111a80907ec4e059f1/contracts/src

Status: Informative

Severity: • NONE

Vulnerability details

Summary

RewardClaims allows multiple windows to reuse the same merkleRoot. Because claims are tracked per (claimWindowId, index), the same leaf can be claimed once in each configured window. If the operator accidentally configures two same-token windows with the same root, the contract treats them as two independent reward obligations and pays both.

This is real behavior, but it should not be framed as attacker theft or a reserve bypass. _tokenReserved[token] is credited for both windows. The contract pays exactly what the operator configured. The sponsor value is an operator safety guard against accidental duplicate same-token windows.

Affected Code

- contracts/src/RewardClaims.sol:124-151 configures a window and credits _tokenReserved[token].
- contracts/src/RewardClaims.sol:219 checks _claimed[claimWindowId].get(index).
- contracts/src/RewardClaims.sol:223-224 verifies the leaf (index, account, amount).
- contracts/src/RewardClaims.sol:227-229 marks the index claimed and decrements both remaining and _tokenReserved.
- contracts/docs/reward-claims-spec.md:72 defines the claimed flag per (claimWindowId, index).
- contracts/docs/reward-claims-spec.md:76 deliberately separates claimWindowId from merkleRoot.

Code Details

Window configuration does not check root uniqueness:

```
// contracts/src/RewardClaims.sol:124-151
function configureWindow(
    bytes32 claimWindowId,
    address token,
    bytes32 merkleRoot,
    uint256 totalAmount
) external onlyOwner {
    if (claimWindowId == bytes32(0)) revert ZeroWindowId();
```

```
    if (_windows[claimWindowId].token != address(0)) revert
WindowAlreadyConfigured();
    if (token == address(0)) revert ZeroAddress();
    if (merkleRoot == bytes32(0)) revert ZeroMerkleRoot();
    if (totalAmount == 0) revert ZeroAmount();

    ...

    _windows[claimWindowId] = Window({
        token: token,
        merkleRoot: merkleRoot,
        totalAmount: totalAmount,
        remaining: totalAmount,
        active: true,
        expiresAt: expiresAt,
        swept: false
    });
    _tokenReserved[token] += totalAmount;
```

Claims are one-shot per window and index:

```
// contracts/src/RewardClaims.sol:219-229
if (_claimed[claimWindowId].get(index)) revert AlreadyClaimed();

bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(index, account,
amount))));
if (!MerkleProof.verify(proof, w.merkleRoot, leaf)) revert InvalidProof();
if (amount > w.remaining) revert InsufficientRemaining();

_claimed[claimWindowId].set(index);
w.remaining -= amount;
_tokenReserved[w.token] -= amount;
```

The spec explicitly says claimed state is per (claimWindowId, index):

```
contracts/docs/reward-claims-spec.md:72
Plus a claimed flag per `(claimWindowId, index)` pair recorded in a bitmap...
```

Failure Scenario

1. Operator configures W1:
   - token: USDC
   - root: R
   - totalAmount: 100,000e6
2. Operator accidentally configures W2:
   - token: USDC
   - root: same R
   - totalAmount: 100,000e6
3. Alice has a valid leaf (index = 0, account = Alice, amount = 1,000e6) in root R.
4. Alice claims in W1; _claimed[W1][0] is set.
5. Alice claims in W2; _claimed[W2][0] was still unset, so the claim succeeds again.

Impact

If the duplicate same-token window was accidental, recipients can receive duplicated rewards. This is an operator mistake, but the contract has no on-chain guard or warning to prevent it.

The impact should be described as duplicate obligations, not reserve theft. _tokenReserved[token] increases for both windows at configuration, so recoverToken() correctly protects the combined amount.

Recommendation

If duplicate same-token roots are never intended, add a (token, merkleRoot) duplicate guard:

```
mapping(address => mapping(bytes32 => bool)) private _seenTokenRoot;

if (_seenTokenRoot[token][merkleRoot]) revert DuplicateTokenRoot();
_seenTokenRoot[token][merkleRoot] = true;
```

Avoid a global merkleRoot uniqueness guard because the spec allows same-content distributions in different tokens:

```
contracts/docs/reward-claims-spec.md:76
Conflating them would collide on same-content distributions
(e.g. same wallets and amounts paid in a different token).
```

Do not add claimWindowId to the leaf unless the sponsor intentionally changes the documented leaf format and backend proof builder.

Validation steps

1. Build a single-leaf merkle root for:

```
index = 0
account = Alice
amount = 1,000e6
leaf = keccak256(bytes.concat(keccak256(abi.encode(index, account, amount))))
root = leaf
proof = []
```

2. Fund RewardClaims with 2,000e6 USDC.

3. As owner, call:

```
configureWindow(W1, USDC, root, 1_000e6);
configureWindow(W2, USDC, root, 1_000e6);
```

4. Confirm both calls succeed because contracts/src/RewardClaims.sol:124-151 rejects duplicate claimWindowId, but does not reject duplicate (token, merkleRoot).

5. Call:

```
claim(W1, 0, Alice, 1_000e6, proof);
claim(W2, 0, Alice, 1_000e6, proof);
```

6. Confirm Alice receives 2,000e6 total. Also confirm this is not a reserve bypass: _tokenReserved[USDC] was credited by 2,000e6 across both windows and decremented by 1,000e6 on each successful claim at contracts/src/RewardClaims.sol:227-229.

Comments history

ALGOXEN

Thanks!

However, same root windows are a possible and desired scenario when two windows have exactly the same rewards for the same recipients. This is very plausible if there are very few stakers.

ABOUT PROJECT

Algoxen is an institutional-grade algorithmic trading and quantitative asset management platform.

DISCLAIMER

HackenProof Disclaimer

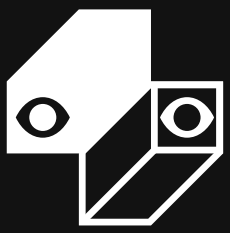
The application provided for assessment has been analyzed in accordance with generally accepted industry practices applicable at the time this report was prepared. The assessment focused on identifying cybersecurity vulnerabilities and security issues in the application's source code, deployment, configuration, and functionality, based solely on the materials, access, and scope made available for the engagement.

This report does not provide any representation or warranty that all vulnerabilities, weaknesses, or misconfigurations have been identified, nor does it guarantee the absolute security, reliability, or bug-free status of the application. The conclusions herein apply only to the specific version and scope reviewed and may no longer remain valid after any modification, update, or environmental change. This report should not be regarded as a definitive or exhaustive statement of the application's security posture.

Technical Disclaimer

Applications, whether decentralized applications (DApps) or other software systems, are deployed and operated within technical environments that may include blockchains, cloud services, operating systems, programming languages, frameworks, libraries, APIs, third-party services, and infrastructure components. Any of these elements may contain inherent weaknesses, undisclosed vulnerabilities, or configuration issues that can result in security incidents, operational failures, or unauthorized access.

The assessment reflects the condition of the application only at the time of testing and within the agreed scope, and the absence of findings in this report must not be interpreted as evidence that no vulnerabilities exist. Continuous security practices, including secure development, independent reviews, monitoring, patch management, and periodic reassessments, remain necessary to maintain an appropriate level of security.



HackenProof

Expert crowdsourced security provider

PRIVATE SECURITY AUDIT

Contact us:

Linkedin | X | Website | Mail

2026