// Smart Contract Security Assessment

02.05.2025 - 02.07.2025

Treasury Vesting

SolanaForg

HALBORN

TreasuryVesting - SolanaForg

Prepared by: HALBORN

Last Updated 02/19/2025

Date of Engagement: February 5th, 2025 - February 7th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
12	1	0	5	3	3

TABLE OF CONTENTS

1. Introduction

`SolanaForg` engaged Halborn to conduct a security assessment on smart contracts beginning on February 5th, 2025 and ending on February 7th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn dedicated 3 days for the engagement and assigned one full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were all addressed by the `SolanaForg team`. The main ones were the following:

- `Implement Correct logic for token distribution`
- `Ensure accurate accounting for multi sig approvals`
- `Strengthening validation during token allocation`
- `Implement proper access control on release functions.`
- `Add an expiry mechanism to time-lock operations.`
- `Add explicit category validation at the start of the executeAddCategory function.`
- `Disable the initializer in the implementation contract.`

3. Test Approach And Methodology

Halborn performed a combination of manual, semi-automated and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any vulnerability classes
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Local deployment and testing (`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE

Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: SolanaForg Private Repository

(b) Assessed Commit ID: V2

(c) Items in scope:

- contracts/TreasuryVesting.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- V6
- V3
- V5

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	5	3	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NO TOKEN DISTRIBUTION IN BATCHRELEASE DUE TO PREMATURE STATE UPDATES	CRITICAL	SOLVED - 02/19/2025
TOTAL AMOUNT LIMIT CAN BE BYPASSED DURING ALLOCATIONS	MEDIUM	SOLVED - 02/13/2025
MISSING ACCESS CONTROL ON TOKEN RELEASE FUNCTIONS	MEDIUM	SOLVED - 02/13/2025
CLEANUP REVERTS FOR PAUSE/UNPAUSE APPROVALS	MEDIUM	SOLVED - 02/19/2025
DOUBLE-COUNTING IN MULTI-SIGNATURE APPROVAL FOR USERS WITH MULTIPLE ROLES	MEDIUM	SOLVED - 02/19/2025
OPERATOR APPROVALS EQUAL TO ADMIN APPROVALS	MEDIUM	SOLVED - 02/19/2025
TIMELOCK OPERATIONS WITHOUT EXPIRY	LOW	SOLVED - 02/13/2025
MISSING CATEGORY VALIDATION	LOW	SOLVED - 02/13/2025
INITIALIZER NOT DISABLED	LOW	SOLVED - 02/13/2025

INSUFFICIENT VALIDATION OF VESTING DURATION	INFORMATIONAL	SOLVED - 02/13/2025
CENTRALIZATION RISKS	INFORMATIONAL	SOLVED - 02/13/2025
REDUNDANT CONSTANTS	INFORMATIONAL	SOLVED - 02/19/2025

7. FINDINGS & TECH DETAILS

7.1 NO TOKEN DISTRIBUTION IN BATCHRELEASE DUE TO PREMATURE STATE UPDATES

// CRITICAL

Description

In the `batchRelease()` function of `TreasuryVesting`, currently separates state updates and token transfers into two separate loops, following the checks-effects-interactions pattern.

```
// Current implementation
function batchRelease(bytes32 category, address[] calldata users) external {

    //...

    // First loop: Updates state
    for (uint256 i = 0; i < users.length; i++) {
        uint256 releasable = getReleasableAmount(users[i], category); // Returns X tokens
        if (releasable > 0) {
            userReleased[users[i]][category] += releasable;  // Updates state
            categoryVestings[category].released += releasable;
            totalReleased += releasable;
            processed++;
        }
    }

    // Second loop: Attempts transfers
    for (uint256 i = 0; i < users.length; i++) {
        uint256 releasable = getReleasableAmount(users[i], category); // Returns 0 because state was updated
        if (releasable > 0) { // This condition is never true
            bdagToken.safeTransferFrom(msg.sender, users[i], releasable);
        }
    }

    //...

}
```

However, this implementation causes the second loop's `getReleasableAmount()` calls to return 0 since the state has already been updated in the first loop, resulting in no tokens being transferred to users.

Proof of Concept

Add the following test function in the foundry test file:

```
    function test_BatchRelease() public {
        test_AddEarlyBirdCategory();

        // Setup allocations
        vm.startPrank(operator);
        treasuryVesting.allocateTokens(user1, treasuryVesting.EARLY_BIRD_CATEGORY(), 1000e18);
```

```
        treasuryVesting.allocateTokens(user2, treasuryVesting.EARLY_BIRD_CATEGORY(), 2000e18);
        vm.stopPrank();

        address[] memory users = new address[](2);
        users[0] = user1;
        users[1] = user2;

        vm.startPrank(admin);
        treasuryVesting.batchRelease(treasuryVesting.EARLY_BIRD_CATEGORY(), users);
        vm.stopPrank();
        assertEq(token.balanceOf(user1), 1000e18);
        assertEq(token.balanceOf(user2), 2000e18);
    }
```

Output:

```
    ├─ [708] ERC1967Proxy::EARLY_BIRD_CATEGORY() [staticcall]
    │   ├─ [318] TreasuryVesting::EARLY_BIRD_CATEGORY() [delegatecall]
    │   │   └─ ← [Return] 0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244
    │   └─ ← [Return] 0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244
    ├─ [86325] ERC1967Proxy::batchRelease(0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244, [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF, 0x537C8f3d3E18dF5517a58B3fB9D9143697996802])
    │   ├─ [85914] TreasuryVesting::batchRelease(0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244, [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF, 0x537C8f3d3E18dF5517a58B3fB9D9143697996802]) [delegatecall]
    │   │   ├─ emit BatchReleaseStarted(category: 0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244, numberOfBeneficiaries: 2, timestamp: 172801 [1.728e5])
    │   │   ├─ emit BatchReleaseCompleted(category: 0x2608ef2c6d76ec500d889693eac4d248785aec4930db833ddcd943fe5ed44244, totalReleased: 3000000000000000000000 [3e21], beneficiariesProcessed: 2, timestamp: 172801 [1.728e5])
    │   │   └─ ← [Stop]
    │   └─ ← [Return]
    ├─ [0] VM::stopPrank()
    │   └─ ← [Return]
    ├─ [2562] MockERC20::balanceOf(user1: [0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF]) [staticcall]
    │   └─ ← [Return] 0
    ├─ [0] VM::assertEq(0, 1000000000000000000000 [1e21]) [staticcall]
    │   └─ ← [Revert] assertion failed: 0 != 1000000000000000000000
    └─ ← [Revert] assertion failed: 0 != 1000000000000000000000

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 9.48ms (2.02ms CPU time)

Ran 1 test suite in 1.22s (9.48ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/TreasuryVestingV5.t.sol:TreasuryVestingV5Test
[FAIL. Reason: assertion failed: 0 != 1000000000000000000000] test_BatchRelease() (gas: 807715)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:H/D:N/Y:L (10.0)

Recommendation

Combine the state updates and token transfers into a single loop to ensure that tokens are properly distributed:

```
function batchRelease(bytes32 category, address[] calldata users)
    external
    onlyRole(ADMIN_ROLE)
    nonReentrant
    whenNotPaused
{
```

```
    require(users.length > 0, "Empty users array");
    require(users.length <= MAX_BATCH_SIZE, "Batch too large");

    emit BatchReleaseStarted(category, users.length, block.timestamp);

    uint256 totalReleased;
    uint256 processed;

    for (uint256 i = 0; i < users.length; i++) {
        uint256 releasable = getReleasableAmount(users[i], category);
        if (releasable > 0) {
            userReleased[users[i]][category] += releasable;
            categoryVestings[category].released += releasable;
            totalReleased += releasable;
            processed++;
            bdagToken.safeTransferFrom(msg.sender, users[i], releasable);
        }
    }

    emit BatchReleaseCompleted(category, totalReleased, processed, block.timestamp);
}
```

Remediation

SOLVED: The suggested mitigation was implemented by the **BlockDAG team.**

Remediation Hash

V6

7.2 TOTAL AMOUNT LIMIT CAN BE BYPASSED DURING ALLOCATIONS

// MEDIUM

Description

The `allocateTokens` function in the `TreasuryVesting` contract contains a vulnerability that allows operators to allocate more tokens than the category's total amount limit. This occurs because the check that's meant to enforce the category limit uses `vesting.released`, which is never updated during token allocation or release, making the limit check ineffective.

```
function allocateTokens(address user, bytes32 category, uint256 amount) external {
    // ...
    require(vesting.released + amount <= vesting.totalAmount, "Exceeds category limit");
    // vesting.released is never updated, always remains 0
    // ...
}
```

The vulnerability allows unlimited token allocations within a category, bypassing the intended total amount limit. This could lead to:

- More tokens being allocated than intended by the protocol
- Potential insolvency if more tokens are promised than available
- Breaking of tokenomics and vesting schedules
- Loss of funds if the contract doesn't have enough tokens to cover all allocations

Proof of Concept

Here's a test function proving that token allocations can exceed total amount:

```
function test_ExceedCategoryLimit() public {
    // Setup category with 1000e18 total amount limit
    uint256[] memory releaseSteps = new uint256[](1);
    uint256[] memory timeSteps = new uint256[](1);
    releaseSteps[0] = 10000;
    timeSteps[0] = 0;

    vm.startPrank(admin);
    bytes32 operationId = keccak256(
        abi.encode(
            treasuryVesting.OPERATION_ADD_CATEGORY(),
            treasuryVesting.EARLY_BIRD_CATEGORY(),
            block.timestamp,
            0,
            1000e18,
            releaseSteps,
            timeSteps
        )
    );

    treasuryVesting.scheduleAddCategory(
        treasuryVesting.EARLY_BIRD_CATEGORY(),
        block.timestamp,
        0,
        1000e18,
        releaseSteps,
        timeSteps
    );

    vm.warp(block.timestamp + treasuryVesting.TIMELOCK_DURATION_ADD_CATEGORY());
    treasuryVesting.executeAddCategory(operationId);
    vm.stopPrank();

    vm.startPrank(operator);
    // Allocate maximum amount to first user
    treasuryVesting.allocateTokens(user1, treasuryVesting.EARLY_BIRD_CATEGORY(), 1000e18);

    // Should fail but succeeds: allocate same amount to second user
    treasuryVesting.allocateTokens(user2, treasuryVesting.EARLY_BIRD_CATEGORY(), 1000e18);
    vm.stopPrank();

    // Verify total allocated amount exceeds limit
    uint256 totalAllocated = treasuryVesting.getAllocation(user1, treasuryVesting.EARLY_BIRD_CATEGORY()) +
                            treasuryVesting.getAllocation(user2, treasuryVesting.EARLY_BIRD_CATEGORY());
    assertEq(totalAllocated, 2000e18); // Double the intended limit
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (6.3)

Recommendation

Replace the current check with a tracking mechanism for total allocations.

Remediation

SOLVED: The **BlockDAG team** solved this issue as follows:

- Adds proper tracking of total allocations via `totalAllocated`
- Changes the limit check to use total allocations instead of released amounts
- Updates the running total when new allocations are made
- Maintains the same check in both single and batch allocation functions

Remediation Hash

V3

7.3 MISSING ACCESS CONTROL ON TOKEN RELEASE FUNCTIONS

// MEDIUM

Description

The `releaseTokens` and `batchRelease` functions lack access control, allowing anyone to release tokens:

```
function releaseTokens(
        address user,
        bytes32 category
    ) external nonReentrant whenNotPaused returns (uint256)

function batchRelease(
        bytes32 category,
        address[] calldata users
    ) external nonReentrant whenNotPaused
```

This implementation does not follow the intention as stated in the provided documentation:

> **3.3 Admin-Only Release**
>
> - The admin triggers batch release for each category after verifying the correct vesting stage has arrived.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (6.3)

Recommendation

Add appropriate access control to both release functions.

Remediation

SOLVED: The **SolForg team** solved this issue as follows:

- Adds the `onlyRole(ADMIN_ROLE)` modifier to both functions
- Ensures only authorized admins can trigger token releases
- Maintains existing nonReentrant and pause checks
- Aligns with the requirement for admin-only releases

Remediation Hash

V3

7.4 CLEANUP REVERTS FOR PAUSE/UNPAUSE APPROVALS

// MEDIUM

Description

The `TreasuryVesting` contract implements two different patterns for handling operations. While standard operations (like adding categories) use a structured `TimelockOperation` system with proper expiry and cleanup mechanisms, emergency operations (pause/unpause) use a simplified direct hashing approach.

```
// Emergency operations use direct hashing
bytes32 pauseOperationId = keccak256("EMERGENCY_PAUSE");
```

```
// While standard operations use TimelockOperation struct
struct TimelockOperation {
    bytes32 operationId;
    uint256 executeTime;
    uint256 expiryTime;
    bool executed;
    bytes encodedParams;
}
```

This inconsistency prevents the cleanup function from working properly for emergency operations, as they don't exist in the `timelockOperations` mapping.

Impact:

- Stale pause/unpause approvals remain in the system
- The approval state doesn't get reset after execution
- This could lead to confusion and potential security issues if old approvals are reused

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:H/D:N/Y:L (5.9)

Recommendation

- Standardize the operation handling:
  - Use the same timelock and operation ID pattern for all operations including pause/unpause
  - Include pause/unpause in the standard operation types
- Reset approval states after execution:
  - Clear approvals after successful pause/unpause
  - Implement proper cleanup for all operation types
- Add proper expiry mechanism for pause/unpause operations

Remediation

SOLVED: The suggested mitigation was implemented by the **BlockDAG team** - *Use the same timelock and operation ID pattern for all operations including pause/unpause*

Remediation Hash

V5

7.5 DOUBLE-COUNTING IN MULTI-SIGNATURE APPROVAL FOR USERS WITH MULTIPLE ROLES

// MEDIUM

Description

In `TreasuryVesting::approveOperation()` function, when a user with both `ADMIN_ROLE` and `OPERATOR_ROLE` approves an operation, both approval counters are incremented, effectively giving them two votes from a single address.

```
function approveOperation(bytes32 operationId) external {
    require(hasRole(ADMIN_ROLE, msg.sender) || hasRole(OPERATOR_ROLE, msg.sender), "Not authorized");

    MultiSigApproval storage approval = multiSigApprovals[operationId];
    require(!approval.hasApproved[msg.sender], "Already approved");
    require(!approval.executed, "Operation already executed");

    approval.hasApproved[msg.sender] = true;

    // Issue: Both counters increment if user has both roles
    if (hasRole(ADMIN_ROLE, msg.sender)) {
        approval.adminApprovalCount++;
    }
    if (hasRole(OPERATOR_ROLE, msg.sender)) {
        approval.operatorApprovalCount++;
    }

    emit MultiSigApprovalSubmitted(
        operationId, msg.sender, approval.adminApprovalCount, approval.operatorApprovalCount
    );
}
```

This undermines the security of the multi-signature system by allowing users with multiple roles to have disproportionate voting power.

Proof of Concept

Add the following to the foundry test file:

```
   // Test multi-sig approval tracking
    function test_MultiSigApprovalTracking() public {
        uint256[] memory releaseSteps = new uint256[](1);
        uint256[] memory timeSteps = new uint256[](1);
        releaseSteps[0] = 10000;
        timeSteps[0] = 0;

        vm.startPrank(admin);
        bytes32 operationId = keccak256(
            abi.encode(
                treasuryVesting.OPERATION_ADD_CATEGORY(),
                treasuryVesting.EARLY_BIRD_CATEGORY(),
                block.timestamp,
                100000e18,
                releaseSteps,
```

```
                timeSteps
            )
        );

        treasuryVesting.scheduleAddCategory(
            treasuryVesting.EARLY_BIRD_CATEGORY(), block.timestamp, 100000e18, releaseSteps, timeSteps
        );

        // Track admin approvals
        treasuryVesting.approveOperation(operationId);
        (uint256 adminCount, uint256 operatorCount,,) = treasuryVesting.multiSigApprovals(operationId);
        assertEq(adminCount, 1);
        assertEq(operatorCount, 0);
        vm.stopPrank();

        // Track operator approvals
        vm.prank(operator);
        treasuryVesting.approveOperation(operationId);
        (adminCount, operatorCount,,) = treasuryVesting.multiSigApprovals(operationId);
        assertEq(adminCount, 1);
        assertEq(operatorCount, 1);
    }
```

Output:

The assertions fail, proving the failing integrity of approval counts.

```
    │   │   └─ ← [Stop]
    │   └─ ← [Return]
    ├─ [75116] ERC1967Proxy::approveOperation(0xfc4db19074a1cf2563cd74393b45ae3b7150a722832f7993f0c95c0a65b17d61)
    │   ├─ [74726] TreasuryVesting::approveOperation(0xfc4db19074a1cf2563cd74393b45ae3b7150a722832f7993f0c95c0a65b17d61) [delegatecall]
    │   │   ├─ emit MultiSigApprovalSubmitted(operationId: 0xfc4db19074a1cf2563cd74393b45ae3b7150a722832f7993f0c95c0a65b17d61, approver: admin: [0xaA10a84C
E7d9AE517a52c6d5cA153b369Af99ecF], adminApprovals: 1, operatorApprovals: 1)
    │   │   └─ ← [Stop]
    │   └─ ← [Return]
    ├─ [3369] ERC1967Proxy::multiSigApprovals(0xfc4db19074a1cf2563cd74393b45ae3b7150a722832f7993f0c95c0a65b17d61) [staticcall]
    │   ├─ [2964] TreasuryVesting::multiSigApprovals(0xfc4db19074a1cf2563cd74393b45ae3b7150a722832f7993f0c95c0a65b17d61) [delegatecall]
    │   │   └─ ← [Return] 1, 1, false
    │   └─ ← [Return] 1, 1, false
    ├─ [0] VM::assertEq(1, 1) [staticcall]
    │   └─ ← [Return]
    ├─ [0] VM::assertEq(1, 0) [staticcall]
    │   └─ ← [Revert] assertion failed: 1 != 0
    └─ ← [Revert] assertion failed: 1 != 0

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.95ms (1.16ms CPU time)

Ran 1 test suite in 1.88s (8.95ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/TreasuryVestingV5.t.sol:TreasuryVestingV5Test
[FAIL. Reason: assertion failed: 1 != 0] test_MultiSigApprovalTracking() (gas: 381187)
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:H/D:N/Y:N (5.9)

Recommendation

Modify the approval logic to use mutually exclusive conditions, ensuring a user can only increment one counter even if they have multiple roles:

```
function approveOperation(bytes32 operationId) external {
    require(hasRole(ADMIN_ROLE, msg.sender) || hasRole(OPERATOR_ROLE, msg.sender), "Not authorized");

    MultiSigApproval storage approval = multiSigApprovals[operationId];
    require(!approval.hasApproved[msg.sender], "Already approved");
    require(!approval.executed, "Operation already executed");

    approval.hasApproved[msg.sender] = true;

    // Fix: Mutually exclusive conditions prevent double counting
    if (hasRole(ADMIN_ROLE, msg.sender)) {
        approval.adminApprovalCount++;
    } else if (hasRole(OPERATOR_ROLE, msg.sender)) {
        approval.operatorApprovalCount++;
    }

    emit MultiSigApprovalSubmitted(
        operationId, msg.sender, approval.adminApprovalCount, approval.operatorApprovalCount
    );
}
```

Remediation

SOLVED: The suggested mitigation was implemented by **BlockDAG team.**

Remediation Hash

V6

7.6 OPERATOR APPROVALS EQUAL TO ADMIN APPROVALS

// MEDIUM

Description

The current implementation doesn't distinguish between operator and admin approvals in the `approveOperation` function. Both operators and admins contribute to the same approval count, which could lead to security risks.

```
function approveOperation(bytes32 operationId) external {
    require(hasRole(ADMIN_ROLE, msg.sender) || hasRole(OPERATOR_ROLE, msg.sender), "Not authorized");

    MultiSigApproval storage approval = multiSigApprovals[operationId];
    require(!approval.hasApproved[msg.sender], "Already approved");
    require(!approval.executed, "Operation already executed");

    approval.hasApproved[msg.sender] = true;
    approval.approvalCount++;  // Both operator and admin approvals increment the same counter
}
```

Impact:

- Operators could potentially contribute to admin-level operations
- The system doesn't properly enforce the separation between `REQUIRED_ADMIN_SIGNATURES (3)` and `REQUIRED_OPERATOR_SIGNATURES (2)`
- This could lead to unauthorized execution of sensitive operations

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:N/Y:L (5.4)

Recommendation

Separate the approval tracking for operators and admins:

- Add separate counters for admin and operator approvals
- Validate the appropriate threshold based on the operation type
- Implement proper role-based approval counting

Remediation

SOLVED: Separate counters for admin and operator approvals were implemented by the **BlockDAG team.**

Remediation Hash

V5

7.7 TIMELOCK OPERATIONS WITHOUT EXPIRY // LOW

Description

Timelock operations don't have an expiration time, allowing them to be executed indefinitely after the timelock period:

```
    struct TimelockOperation {
        bytes32 operationId;      // Unique identifier for the operation
        uint256 executeTime;      // When the operation can be executed
        bool executed;            // Whether operation has been completed
        bytes encodedParams;      // Encoded function parameters
    }
```

This means old operations could be executed long after they become irrelevant.

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:L/I:M/D:M/Y:N (3.1)

Recommendation

Add expiration time to `TimelockOperation` struct:

```
struct TimelockOperation {
    bytes32 operationId;
    uint256 executeTime;
    uint256 expiryTime;
    bool executed;
    bytes encodedParams;
}
```

Remediation

SOLVED: Added expiration time to TimelockOperation struct.

Remediation Hash

V3

7.8 MISSING CATEGORY VALIDATION

// LOW

Description

The `executeAddCategory` function in the TreasuryVesting contract allows the creation of arbitrary categories without validating if they match one of the predefined category types (EARLY_BIRD_CATEGORY, PRESALE_CATEGORY, or TEAM_CATEGORY). While the function includes specific validation logic for known categories, it doesn't prevent the creation of undefined categories:

```
function executeAddCategory(bytes32 operationId) external onlyRole(ADMIN_ROLE) {
    // ... decode parameters ...

    // Category-specific validations
    if (category == EARLY_BIRD_CATEGORY) {
        // Early Bird validations
    }
    else if (category == PRESALE_CATEGORY) {
        // Presale validations
    }
    else if (category == TEAM_CATEGORY) {
        // Team validations
    }
    // No else clause to prevent undefined categories
```

Impact:

- Allows creation of non-standard vesting categories
- Could lead to confusion in token distribution
- Inconsistent vesting rules across the protocol

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:L/I:L/D:M/Y:N (2.8)

Recommendation

Add explicit category validation at the start of the function:

```
function executeAddCategory(bytes32 operationId) external onlyRole(ADMIN_ROLE) {
    TimelockOperation storage operation = timelockOperations[operationId];
    require(operation.operationId != bytes32(0), "Operation doesn't exist");
    require(!operation.executed, "Already executed");
    require(block.timestamp >= operation.executeTime, "Timelock not expired");

    operation.executed = true;
```

```
    // Decode operation parameters
    (
        bytes32 category,
        uint256 start,
        uint256 duration,
        uint256 totalAmount,
        uint256[] memory releaseSteps,
        uint256[] memory timeSteps
    ) = abi.decode(
        operation.encodedParams,
        (bytes32, uint256, uint256, uint256, uint256[], uint256[])
    );

    // Add explicit category validation
    require(
        category == EARLY_BIRD_CATEGORY ||
        category == PRESALE_CATEGORY ||
        category == TEAM_CATEGORY,
        "Invalid category type"
    );

    // Rest of the function...
}
```

Remediation

SOLVED: The **BlockDAG team** solved this issue as follows:

- Added explicit validation of category types at the start
- Maintains all existing category-specific validations
- Ensures only predefined categories can be created
- Keeps the error message clear and descriptive

Remediation Hash

V3

7.9 INITIALIZER NOT DISABLED

// LOW

Description

The `TreasuryVesting` contract is designed to be used with the proxy pattern, as evidenced by its inheritance of `Initializable` and use of the `initialize()` function. However, the contract does not disable the initializer in its constructor, leaving it vulnerable to potential initialization attacks:

```
contract TreasuryVesting is
    Initializable,
    AccessControlUpgradeable,
    ReentrancyGuardUpgradeable,
    PausableUpgradeable
{

    function initialize(
        address admin,
        address _bdagToken
    ) public initializer {
        // ... initialization logic ...
    }
}
```

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:L/I:L/D:M/Y:N (2.8)

Recommendation

Add a constructor that disables initializers:

```
contract TreasuryVesting is
    Initializable,
    AccessControlUpgradeable,
    ReentrancyGuardUpgradeable,
    PausableUpgradeable
{
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(
        address admin,
        address _bdagToken
    ) public initializer {
        // ... rest of initialization logic ...
    }
}
```

This ensures that the implementation contract cannot be initialized directly, following the proper proxy pattern implementation.

Remediation

SOLVED:

- Adds the constructor with `_disableInitializers()`
- Includes the OpenZeppelin annotation for upgrades safety
- Maintains all existing initialization logic

- Prevents potential reinitialization attacks
- Follows best practices for upgradeable contracts

Remediation Hash

V3

7.10 INSUFFICIENT VALIDATION OF VESTING DURATION

// INFORMATIONAL

Description

The `getReleasableAmount` function in the TreasuryVesting contract appears to lack validation for the `duration` field. However, this is not a security concern as the token release schedule is effectively controlled by the `releaseSteps` and `timeSteps` arrays, which are properly validated during category creation:

```
function getReleasableAmount(address user, bytes32 category) public view returns (uint256) {
    CategoryVesting storage vesting = categoryVestings[category];
    if (block.timestamp < vesting.start) return 0;

    uint256 allocation = userAllocations[user][category];
    uint256 totalReleased = userReleased[user][category];
    uint256 releasable = 0;

    // Release schedule is controlled by timeSteps
    for (uint256 i = 0; i < vesting.timeSteps.length; i++) {
        if (block.timestamp >= vesting.start + vesting.timeSteps[i]) {
            releasable += (allocation * vesting.releaseSteps[i]) / 10000;
        }
    }

    return releasable > totalReleased ? releasable - totalReleased : 0;
}
```

Impact:

- Code readability and maintainability could be improved
- duration field in CategoryVesting struct is effectively redundant
- Minor gas cost for storing unused duration value

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:L/I:L/D:L/Y:N (1.7)

Recommendation

Consider removing the redundant duration field or using it for validation.

Remediation

SOLVED: Removed the redundant duration field.

Remediation Hash

V3

7.11 CENTRALIZATION RISKS
// INFORMATIONAL

Description

`TreasuryVesting` contract relies heavily on admin and operator roles for critical functions. Therefore it creates single points of failure if admin/operator keys are compromised.

BVSS

AO:A/AC:M/AX:M/R:N/S:U/C:N/A:L/I:L/D:L/Y:N (1.7)

Recommendation

It is recommended to implement multi-signature requirements for admin/operator actions.

Remediation

SOLVED: The **BlockDAG team** solved this issue as follows:

- Requires multiple signatures for admin and operator actions
- Tracks approvals per operation
- Clears approvals after execution
- Emits events for transparency
- Maintains existing role-based access control

Remediation Hash

V3

7.12 REDUNDANT CONSTANTS

// INFORMATIONAL

Description

The following constants in the `TreasuryVesting` contract are not used:

```
bytes32 public constant OPERATION_UPDATE_SCHEDULE = keccak256("UPDATE_SCHEDULE");
bytes32 public constant OPERATION_PAUSE = keccak256("OPERATION_PAUSE");
bytes32 public constant OPERATION_UNPAUSE = keccak256("OPERATION_UNPAUSE");
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove them for code clarity and maintainability.

Remediation

SOLVED: The suggested mitigation was implemented.

Remediation Hash

V6

8. AUTOMATED TESTING

Introduction

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base. The security team conducted a comprehensive review of findings generated by the Slither static analysis tool. No major issues were found. The issue related to reentrancy is a false positive as the external call is made to a trusted contract.

```
INFO:Detectors:
Reentrancy in TreasuryVesting.batchRelease(bytes32,address[]) (contracts/TreasuryVestingV2.sol#372-396):
        External calls:
        - bdagToken.safeTransferFrom(msg.sender,users[i],releasable) (contracts/TreasuryVestingV2.sol#389)
        State variables written after the call(s):
        - categoryVestings[category].released += releasable (contracts/TreasuryVestingV2.sol#388)
        TreasuryVesting.categoryVestings (contracts/TreasuryVestingV2.sol#68) can be used in cross function reentrancies:
        - TreasuryVesting.allocateTokens(address,bytes32,uint256) (contracts/TreasuryVestingV2.sol#284-298)
        - TreasuryVesting.allocateTokensBatch(address[],bytes32,uint256[]) (contracts/TreasuryVestingV2.sol#300-324)
        - TreasuryVesting.categoryVestings (contracts/TreasuryVestingV2.sol#68)
        - TreasuryVesting.executeAddCategory(bytes32) (contracts/TreasuryVestingV2.sol#200-282)
        - TreasuryVesting.getReleasableAmount(address,bytes32) (contracts/TreasuryVestingV2.sol#338-353)
        - TreasuryVesting.scheduleAddCategory(bytes32,uint256,uint256,uint256,uint256[],uint256[]) (contracts/TreasuryVestingV2.sol#154-198)
        - userReleased[users[i]][category] += releasable (contracts/TreasuryVestingV2.sol#387)
        TreasuryVesting.userReleased (contracts/TreasuryVestingV2.sol#70) can be used in cross function reentrancies:
        - TreasuryVesting.getReleasableAmount(address,bytes32) (contracts/TreasuryVestingV2.sol#338-353)
        - TreasuryVesting.getReleasedAmount(address,bytes32) (contracts/TreasuryVestingV2.sol#330-332)
        - TreasuryVesting.getRemainingAmount(address,bytes32) (contracts/TreasuryVestingV2.sol#334-336)
        - TreasuryVesting.userReleased (contracts/TreasuryVestingV2.sol#70)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

```
INFO:Detectors:
TreasuryVesting.batchRelease(bytes32,address[]).totalReleased (contracts/TreasuryVestingV2.sol#381) is a local variable never initialized
TreasuryVesting.batchRelease(bytes32,address[]).processed (contracts/TreasuryVestingV2.sol#382) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
TreasuryVesting.scheduleAddCategory(bytes32,uint256,uint256,uint256,uint256[],uint256[]) (contracts/TreasuryVestingV2.sol#154-198) uses timestamp for compariso
ns
        Dangerous comparisons:
        - require(bool,string)(timelockOperations[operationId].operationId == bytes32(0),Operation exists) (contracts/TreasuryVestingV2.sol#176)
TreasuryVesting.executeAddCategory(bytes32) (contracts/TreasuryVestingV2.sol#200-282) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= operation.executeTime,Timelock not expired) (contracts/TreasuryVestingV2.sol#204)
TreasuryVesting.getReleasableAmount(address,bytes32) (contracts/TreasuryVestingV2.sol#338-353) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp < vesting.start (contracts/TreasuryVestingV2.sol#340)
        - block.timestamp >= vesting.start + vesting.timeSteps[i] (contracts/TreasuryVestingV2.sol#347)
TreasuryVesting.executeSchedulePause(bytes32) (contracts/TreasuryVestingV2.sol#432-442) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= operation.executeTime,Timelock not expired) (contracts/TreasuryVestingV2.sol#436)
TreasuryVesting.executeUnpause(bytes32) (contracts/TreasuryVestingV2.sol#463-473) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= operation.executeTime,Timelock not expired) (contracts/TreasuryVestingV2.sol#467)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project’s integrity and addressing potential vulnerabilities introduced by code modifications.