// Security Assessment

12.16.2025 - 01.08.2026

Earn
Mezo

HALBORN

Earn - Mezo

Prepared by: HALBORN

Last Updated 03/17/2026

Date of Engagement: December 16th, 2025 - January 8th, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
16	0	1	4	4	7

TABLE OF CONTENTS

1. INTRODUCTION

`Mezo` engaged `Halborn` to perform a security assessment of their smart contracts starting on December 16th 2025 and ending on January 8th, 2026. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

The `Mezo` codebase implements a dual-token, vote-escrow–based economic and governance system for the Mezo chain, designed to align Bitcoin holders and MEZO participants through a boosted virtual voting mechanism. The system combines time-locked Bitcoin (veBTC) and time-locked MEZO (veMEZO), represented as transferable ERC-721 NFTs, to derive voting power used for allocating protocol fees, emissions, and incentives across liquidity pools, validators, ecosystem applications, and chain-level components.

2. ASSESSMENT SUMMARY

`Halborn` was allocated 16 day for this engagement and assigned 1 full-time security engineers to conduct a comprehensive review of the smart contracts within scope. The engineers are experts in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were partially addressed by the `Mezo` `team`. The main recommendations were:

- `Implemented a minimum deposit duration requirement to prevent flash loan attacks.`
- `Users should be allowed to claim their accrued yield regardless of their current sMUSD balance.`
- `Restrict reward beneficiary updates to validator or operator.`
- `Implement a mechanism in _onGaugeKilled() to return unclaimed bribe tokens to an appropriate recipient.`
- `Track returned funds separately and distribute them correctly.`

3. TEST APPROACH AND METHODOLODY

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts (`Foundry`).
- Fork testing against main networks (`Foundry`).
- Static security analysis of scoped contracts, and imported functions (Slither).

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

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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 (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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

REPOSITORY

(a) Repository: tigris-token-launch

(b) Assessed Commit ID: 2ed567e

(c) Items in scope:

- solidity/contracts/BoostVoter.sol
- solidity/contracts/MEZOChainSplitter.sol
- solidity/contracts/MEZOEcosystemSplitter.sol
- solidity/contracts/MEZOMinter.sol
- solidity/contracts/NonStakingVoter.sol
- solidity/contracts/Splitter.sol
- solidity/contracts/ThirdPartyVoter.sol
- solidity/contracts/ValidatorsVoter.sol
- solidity/contracts/Voter.sol
- solidity/contracts/VotingEscrow.sol
- solidity/contracts/ve/Balance.sol
- solidity/contracts/ve/Delegation.sol
- solidity/contracts/ve/Escrow.sol
- solidity/contracts/ve/Grant.sol
- solidity/contracts/ve/ManagedNFT.sol
- solidity/contracts/ve/NFT.sol
- solidity/contracts/ve/VeERC2771Context.sol
- solidity/contracts/ve/VotingEscrowState.sol
- solidity/contracts/vaults/IdleStrategy.sol
- solidity/contracts/vaults/MUSDSavingsRate.sol
- solidity/contracts/gauges/Gauge.sol
- solidity/contracts/gauges/NoFeesGauge.sol
- solidity/contracts/gauges/NonStakingGauge.sol
- solidity/contracts/gauges/PoolGauge.sol
- solidity/contracts/gauges/StakingGauge.sol
- solidity/contracts/gauges/VaultGauge.sol

Out-of-Scope: External dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 67b8d87
- e976ae6
- 7e77f12
- 73c40c4

- 39220c3
- 2d1de64
- f613ac0
- 845bf88
- 1fafbb0

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	4	4

INFORMATIONAL
7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
YIELD INDEX MANIPULATION THROUGH FLASH LOAN ATTACK	HIGH	SOLVED - 02/10/2026
ACCRUED YIELD BECOMES LOCKED DUE TO TRANSFERRING OF TOKENS	MEDIUM	SOLVED - 02/10/2026
REWARDS BENEFICIARY CAN BE CHANGED WITHOUT VALIDATOR OR GOVERNANCE APPROVAL	MEDIUM	SOLVED - 02/10/2026
PERMANENT LOSS OF BRIBE TOKENS WHEN GAUGE IS KILLED WITHOUT VOTERS	MEDIUM	SOLVED - 02/10/2026
INCORRECT TOKEN DISTRIBUTION WHEN GAUGES ARE KILLED	MEDIUM	RISK ACCEPTED - 02/10/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POTENTIAL YIELD MISALLOCATION DURING STRATEGY MIGRATION DUE TO RACE CONDITION	LOW	SOLVED - 02/10/2026
FIRST DEPOSITOR CAN CAPTURE ALL ACCUMULATED PENDING YIELD	LOW	RISK ACCEPTED - 02/10/2026
UNBOUNDED LOOP IN POKEBOOSTS CAN CAUSE DOS	LOW	RISK ACCEPTED - 02/10/2026
VALIDATOR OPERATOR ADDRESS CAN BE REUSED AFTER EXIT DESPITE NON-REVIVAL ASSUMPTION	LOW	SOLVED - 02/10/2026
OWNERSHIP TRANSFER USES SINGLE-STEP PATTERN INSTEAD OF TWO-STEP OWNERSHIP	INFORMATIONAL	ACKNOWLEDGED - 02/11/2026
SELF-TRANSFERS TRIGGER REDUNDANT YIELD ACCOUNTING UPDATES	INFORMATIONAL	SOLVED - 02/10/2026
LATE VALIDATION OF GAUGE EXISTENCE AND FACTORY APPROVAL CAUSES UNNECESSARY GAS CONSUMPTION	INFORMATIONAL	ACKNOWLEDGED - 02/11/2026
OUTDATED SOLIDITY PRAGMA	INFORMATIONAL	SOLVED - 02/10/2026
MISSING ZERO ADDRESS VALIDATION IN SETBOOSTER	INFORMATIONAL	SOLVED - 02/10/2026
REDUNDANT STATE WRITE AND EVENT EMISSION	INFORMATIONAL	ACKNOWLEDGED - 02/11/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION RISK DURING SETSTRATEGY()	INFORMATIONAL	ACKNOWLEDGED - 02/11/2026

7. FINDINGS & TECH DETAILS

7.1 YIELD INDEX MANIPULATION THROUGH FLASH LOAN ATTACK

// HIGH

Description

The `deposit()` function in the `MUSDSavingsRate` contract allows users to deposit MUSD and receive sMUSD tokens at a 1:1 ratio without any time-locking mechanism or minimum holding period. This creates a vulnerability where an attacker can manipulate yield distribution through flash loan attacks.

An attacker can exploit this by:

1. Flash loaning a massive amount of MUSD.
2. Depositing into the vault to receive a disproportionately large amount of sMUSD.
3. Triggering yield distribution via `receiveProtocolYield()` or `receiveYieldFromStrategy()`.
4. Claiming the majority of the distributed yield due to their temporarily inflated share.
5. Withdrawing their principal and repaying the flash loan while keeping the stolen yield.

The vulnerability exists because the yield distribution mechanism uses the current total supply at the time of distribution:

Copy Code

```
238 function _receiveYield(uint256 amount) internal returns (uint256 _totalSupply) {
239     _totalSupply = totalSupply();
240     if (_totalSupply > 0) {
241         uint256 totalYieldToDistribute = amount + pendingYield;
242         pendingYield = 0;
243 
244         uint256 _ratio = (totalYieldToDistribute * 1e18) / _totalSupply;
245         if (_ratio > 0) {
246             yieldIndex += _ratio;
247         }
248     }
249 }
```

Since `receiveProtocolYield()` and `receiveYieldFromStrategy()` are both permissionless functions that anyone can call, an attacker can trigger the distribution themselves.

Proof of Concept

This proof-of-concept demonstrates a critical yield manipulation vulnerability in the `MUSDSavingsRate` contract: an attacker can exploit the permissionless yield distribution and lack of deposit time-locking to dilute legitimate depositor's yield by temporarily inflating their share of the total supply during yield distribution events.

Copy Code

```
1 function testYieldDilutionAttack() public {
2     // Setup: Normal users deposit
3     vm.prank(address(owner));
```

```
        vault.deposit(TOKEN_1M); // 1M sMUSD

        vm.prank(address(owner2));
        vault.deposit(TOKEN_1M); // 1M sMUSD

        // Total supply: 2M sMUSD
        assertEq(vault.totalSupply(), TOKEN_1M * 2);

        // Attacker deposits massive amount (simulating flash loan)
        deal(address(mUSD), address(owner3), TOKEN_1M * 1000, true);
        vm.startPrank(address(owner3));
        mUSD.approve(address(vault), type(uint256).max);
        vault.deposit(TOKEN_1M * 1000); // 1000M sMUSD
        vm.stopPrank();

        // Total supply now: 1002M sMUSD
        assertEq(vault.totalSupply(), TOKEN_1M * 1002);

        // Yield distribution: 100K MUSD
        vm.prank(yieldDistributor);
        vault.receiveProtocolYield(TOKEN_100K);

        // Attacker claims yield
        vm.prank(address(owner3));
        uint256 attackerYield = vault.claimYield();

        // Attacker withdraws
        vm.prank(address(owner3));
        vault.withdraw(TOKEN_1M * 1000);

        // Legitimate users claim their yield
        vm.prank(address(owner));
        uint256 owner1Yield = vault.claimYield();

        vm.prank(address(owner2));
        uint256 owner2Yield = vault.claimYield();

        // Attacker stole almost all the yield
        // Expected fair distribution: owner1 = 50K, owner2 = 50K
        // Actual: attacker gets ~99.8K, legitimate users get ~0.1K each
        assertGt(attackerYield, TOKEN_100K * 99 / 100); // Attacker gets >99%
        assertLt(owner1Yield, TOKEN_100K / 100); // Owner1 gets <1%
        assertLt(owner2Yield, TOKEN_100K / 100); // Owner2 gets <1%
    }
```

The test passes, confirming that the `MUSDSavingsRate` contract suffers from a severe yield dilution vulnerability due to the absence of deposit time-locking. In this PoC, an attacker deposits a massive amount immediately before yield distribution, capturing 99%+ of the yield that should have been distributed proportionally to legitimate long-term depositors.

```
Ran 1 test for test/MUSDSavingsRate.t.sol:MUSDSavingsRateTest
[PASS] testYieldDilutionAttack() (gas: 733735)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.08ms (1.18ms CPU time)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (7.5)

Recommendation

Implement a minimum deposit duration requirement to prevent flash loan attacks. Add time-locking that enforces users must hold their sMUSD tokens for a minimum period before being eligible for withdrawals

or before their position earns yield.

Remediation Comment

SOLVED: The **Mezo team** solved this finding by forcing all accumulated protocol and strategy yield to be distributed to existing holders before minting or burning sMUSD shares, eliminating the ability for a flash-loan attacker to inflate supply and capture past yield.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/67b8d8730a90ed48d04a7ea9f7f743f14157d723

7.2 ACCRUED YIELD BECOMES LOCKED DUE TO TRANSFERRING OF TOKENS

// MEDIUM

Description

The `_claimYield()` function in the `MUSDSavingsRate` contract permanently locks users' accrued yield when they transfer all their sMUSD tokens to another address. The function incorrectly validates that the user has a non-zero balance before allowing yield claims, preventing users from claiming yield that was already credited to their `claimableYield` mapping.

When a user transfers their entire sMUSD balance, the `_beforeTokenTransfer()` hook correctly finalizes their accrued yield and adds it to `claimableYield[user]`. However, after the transfer completes and the user's balance becomes zero, any attempt to claim this yield will revert due to the `NoShares` check:

Copy Code

```
function _claimYield(address depositor) internal returns (uint256) {
    if (balanceOf(depositor) == 0) revert NoShares();

    _updateYieldFor(depositor);

    uint256 claimable = claimableYield[depositor];

    if (claimable > 0) {
        claimableYield[depositor] = 0;
        musdToken.safeTransfer(depositor, claimable);
        emit YieldClaimed(depositor, claimable);
    }

    return claimable;
}
```

Attack Scenario:

1. Alice deposits 1,000 sMUSD into the vault.
2. Over time, 50 MUSD in yield accrues to Alice's position (tracked via yield index).
3. Alice transfers all 1,000 sMUSD to Bob.
4. During the transfer, `_beforeTokenTransfer()` executes:
   - `_updateYieldFor(Alice)` calculates and adds 50 MUSD to `claimableYield[Alice]`
   - Alice's balance becomes 0 after the transfer completes
5. Alice attempts to claim her 50 MUSD yield.
6. `_claimYield()` reverts with `NoShares` error because `balanceOf(Alice) == 0.`
7. Alice's 50 MUSD is permanently locked in the contract with no way to retrieve it.

The `claimableYield[Alice]` mapping still contains 50 MUSD, but there is no function that allows claiming without holding sMUSD tokens. This creates a permanent loss of funds for users who transfer their entire balance.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

Users should be allowed to claim their accrued yield regardless of their current sMUSD balance.

Remediation Comment

SOLVED: The **Mezo team** solved this finding in the specified commit by following the mentioned recommendation.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/e976ae66242b9323e544611bc1dbeb428499834c

7.3 REWARDS BENEFICIARY CAN BE CHANGED WITHOUT VALIDATOR OR GOVERNANCE APPROVAL

// MEDIUM

Description

The `switchRewardsBeneficiary()` function in `NonStakingGauge` authorizes only the current rewards beneficiary to update the beneficiary address. As a result, validator operators and governance have no control over reward redirection once the gauge is created, even though the gauge represents the validator's identity and voting power.

This creates two problematic scenarios:

1. Operator Lockout
   If the validator operator wishes to change the rewards beneficiary (e.g., key rotation, treasury update, or operational restructuring), they are unable to do so unless the current beneficiary cooperates. This permanently removes the operator's ability to manage reward distribution.

2. Adversarial or Compromised Beneficiary
   If the rewards beneficiary becomes malicious, compromised, or otherwise adversarial to the operator:

- They can continue claiming all validator rewards.
- They can unilaterally redirect rewards to any arbitrary address.
- The operator and governance have no recovery or override mechanism.

Copy Code

```
33 function switchRewardsBeneficiary(address newBeneficiary) external {
34     if (_msgSender() != rewardsBeneficiary) revert NotAuthorized();
35     if (newBeneficiary == address(0)) revert ZeroAddress();
36
37     _switchRewardsBeneficiary(newBeneficiary);
38 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

Restrict reward beneficiary updates to **validator or operator.**

Remediation Comment

SOLVED: The **Mezo team** solved this finding by changing documentation to clarify that the beneficiary address **is controlled by the operator** and must remain secure for the validator gauge.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/7e77f12ed1ccfab02162f624b79418bc9db6ba8a

7.4 PERMANENT LOSS OF BRIBE TOKENS WHEN GAUGE IS KILLED WITHOUT VOTERS

// MEDIUM

Description

The `_onGaugeKilled()` function in the `NonStakingVoter` contract only returns emission rewards (`claimable[_gauge]`) to the splitter but fails to handle bribe tokens stored in the associated bribe contract (`gaugeToBribe[_gauge]`). This creates a vulnerability where bribe tokens become permanently locked when a gauge is killed if no users have voted for it.

The vulnerability exists because `_onGaugeKilled()` only handles emission rewards:

Copy Code

```
435 function _onGaugeKilled(address _gauge) internal {
436     if (!isGauge[_gauge]) revert GaugeDoesNotExist(_gauge);
437     if (!isAlive[_gauge]) revert GaugeAlreadyKilled();
438
439     // Only returns emission rewards to splitter
440     uint256 _claimable = claimable[_gauge];
441     if (_claimable > 0) {
442         IERC20(rewardToken).safeTransfer(splitter, _claimable);
443         delete claimable[_gauge];
444     }
445
446     isAlive[_gauge] = false;
447     emit GaugeKilled(_gauge);
448 }
```

Since no voting weight exists (`balanceOf[tokenId] = 0` for all tokenIds), the `earned()` function will always return 0, making the bribes unclaimable. This results in permanent loss of bribe tokens for users who deposited them, as the tokens remain locked in the bribe contract with no mechanism to recover them.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

Implement a mechanism in `_onGaugeKilled()` to return unclaimed bribe tokens to an appropriate recipient.

Remediation Comment

SOLVED: The **Mezo team** solved this finding by changing the documentation to warn about this corner case.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/73c40c45ec9eaa2d5d2bd1e1620b31dd6948ecc1

7.5 INCORRECT TOKEN DISTRIBUTION WHEN GAUGES ARE KILLED

// MEDIUM

Description

When gauges are killed, their unclaimed reward tokens are returned to the splitter and subsequently redistributed according to the standard splitter ratio (e.g., 90% to pool gauges, 10% to third-party voter). This causes tokens that were originally allocated to pool gauges to be partially redirected to third-party voters, resulting in an incorrect and unfair distribution of rewards.

The `killGauge()` function returns unclaimed rewards to the splitter without any mechanism to ensure these tokens are redistributed back to their original intended recipients (pool gauges only or Third party gauges).

These transferred tokens to splitter will be again redistributed according to splitter ratio.

Copy Code

```
506 /// @inheritdoc IVoter
507 function killGauge(address _gauge) external {
508     if (_msgSender() != emergencyCouncil) revert NotEmergencyCouncil();
509     if (!isAlive[_gauge]) revert GaugeAlreadyKilled();
510     // Return claimable back to splitter
511     uint256 _claimable = claimable[_gauge];
512     if (_claimable > 0) {
513         IERC20(rewardToken).safeTransfer(splitter, _claimable);
514         delete claimable[_gauge];
515     }
516     isAlive[_gauge] = false;
517     emit GaugeKilled(_gauge);
518 }
```

For any killed gauge with `claimable` balance `C`, and splitter ratio sending `X%` to pool gauges and `(100-X)%` to third-party:

- Incorrectly redistributed amount: `C × (100-X)%.`
- Pool gauge LPs lose this amount.
- Third-party voter incorrectly gains this amount.

When a third-party voter gauge is killed, the inverse occurs: pool gauges incorrectly receive a portion of the returned tokens that should have been redistributed exclusively back to third-party voter gauges.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

Track returned funds separately and distribute them correctly.

Remediation Comment

RISK ACCEPTED: The **Mezo team** made a business decision to accept the risk of this finding also mentioning that:

> Killing a gauge is an exceptional case executed only by the emergency council. At the time of removal, only pending emissions are returned. Emissions from previous epochs that were distributed via notifyRewardAmount are not returned to the splitter and remain claimable.

7.6 POTENTIAL YIELD MISALLOCATION DURING STRATEGY MIGRATION DUE TO RACE CONDITION

// LOW

Description

The `setStrategy()` function contains a **potential race condition** around yield claiming and strategy migration. While this issue is **not exploitable in the current deployment** because the default `IdleStrategy` does not generate yield and `claimYield()` always returns `0`, the same logic becomes **unsafe if a future yield-generating strategy is introduced**.

If a new strategy accumulates yield internally, the current migration flow can result in **unfair yield distribution or yield loss** due to timing assumptions and lack of migration guards.

Attack Scenario:

1. Old strategy has accrued significant yield (not yet claimed).
2. User monitors mempool and sees `setStrategy()` transaction.
3. User front-runs with `deposit()` of large amount.
4. Owner's `setStrategy()` executes → distributes all accumulated yield.
5. Attacker gets share of yield despite just depositing.
6. Attacker immediately withdraws.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:L (2.5)

Recommendation

Protocol should implement time-lock for `setStrategy()`.

Remediation Comment

SOLVED: The **Mezo team** solved this finding in the specified commit.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/39220c379b0ef589e0be55bf7ce24664a4fc6429

7.7 FIRST DEPOSITOR CAN CAPTURE ALL ACCUMULATED PENDING YIELD

// LOW

Description

When yield is received while the vault has zero sMUSD supply, the yield is accumulated in `pendingYield`. Once the first deposit occurs, all buffered yield is distributed to that depositor, allowing them to capture yield they did not economically earn.

While this behavior is intentional to prevent yield loss, it can lead to unfair outcomes, MEV-style timing strategies, and unexpected large yield allocations if yield accumulates for an extended period without any deposits.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:L (2.5)

Recommendation

Introduce **controls around pending yield accumulation** to limit the amount that can be captured by a single depositor and to reduce timing-based incentives.

Remediation Comment

RISK ACCEPTED: The **Mezo team** made a business decision to accept the risk of this finding and not alter the contracts.

7.8 UNBOUNDED LOOP IN POKEBOOSTS CAN CAUSE DOS

// LOW

Description

The `pokeBoosts` function IN `BoostVoter` iterates over a user-supplied array of `boostableTokenIds` without enforcing any upper bound on its length. If a caller supplies a very large array, the function may exceed the block gas limit and revert. This makes the function impractical or unusable for large inputs and creates a potential Denial of Service (DoS) vector for users attempting to batch-update boosts.

While this does not directly affect protocol funds, it degrades usability and reliability of the batch operation. Calls to `pokeBoosts` with a large number of token IDs may consistently revert due to gas exhaustion.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Impose a reasonable upper bound on `boostableTokenIds.length`.

Remediation Comment

RISK ACCEPTED: The **Mezo team** made a business decision to accept the risk of this finding and not alter the contracts.

7.9 VALIDATOR OPERATOR ADDRESS CAN BE REUSED AFTER EXIT DESPITE NON-REVIVAL ASSUMPTION

// LOW

Description

The `ValidatorsVoter` contract assumes that once a validator leaves the validator set, its associated gauge is permanently killed and must never be reused. However, the implementation does not enforce this restriction.

In `notifyValidatorLeft`, the mapping entry is deleted when a validator leaves. As a result, if the same `operator` address later rejoins the validator set and becomes active again, `_isActiveValidator(operator)` will return `true`, and `validatorToGauge[operator]` will be `address(0)`. This allows `createValidatorGauge` to successfully create a **new gauge for the same operator address**, contradicting the stated design intent.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

If operator reuse is intended to be permanently disallowed then track a separate `everHadGauge[operator]` or `operatorUsed[operator]` flag that is never cleared.

Remediation Comment

SOLVED: The **Mezo team** solved this finding by changing the documentation.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/2d1de64250c10d632412c04489eef45c6f9fcd0c

7.10 OWNERSHIP TRANSFER USES SINGLE-STEP PATTERN INSTEAD OF TWO-STEP OWNERSHIP

// INFORMATIONAL

Description

The `MUSDSavingsRate` contract inherits from `OwnableUpgradeable`, which uses a **single-step ownership transfer mechanism**. In this model, ownership is transferred immediately upon calling `transferOwnership()`, increasing the risk of **irreversible ownership loss** if an incorrect address is provided.

While this does not introduce a direct security vulnerability, it represents a **best-practice deviation** and increases operational risk for privileged administrative functions.

BVSS

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

Recommendation

Adopt a **two-step ownership transfer pattern** to ensure safe handover of control.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** made a business decision to acknowledge this finding and not alter the contracts.

7.11 SELF-TRANSFERS TRIGGER REDUNDANT YIELD ACCOUNTING UPDATES

// INFORMATIONAL

Description

The `_beforeTokenTransfer()` hook in `MUSDSavingsRate` updates yield accounting for both `from` and `to` addresses on every token transfer. When a self-transfer occurs (`from == to`), the same address is processed twice, resulting in redundant yield updates.
While this does not introduce any correctness or security issues in the current implementation, avoiding self-transfer processing is considered safer and more defensive, particularly for contracts with complex accounting logic that may evolve over time.

BVSS

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

Recommendation

Skip yield updates when `from == to` to prevent redundant accounting and improve defensive robustness.

Remediation Comment

SOLVED: The **Mezo team** solved this finding in the specified commit by following the mentioned recommendation.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/f613ac0b67178f05eb37afe226107907891bfbeb

7.12 LATE VALIDATION OF GAUGE EXISTENCE AND FACTORY APPROVAL CAUSES UNNECESSARY GAS CONSUMPTION

// INFORMATIONAL

Description

The internal `_createGauge` function in `Voter` contract performs two critical validation checks only after execution has already reached this low-level helper:

```
576 if (gauges[_stakingToken] != address(0)) revert GaugeExists();
577 if (
578     !IFactoryRegistry(factoryRegistry).isGaugeFactoryApproved(
579         _gaugeFactory
580     )
581 ) revert FactoryPathNotApproved();
```

Copy Code

These checks validate:

1. Whether a gauge already exists for the given staking token.
2. Whether the provided gauge factory is approved.

However, these validations are generic and deterministic and could be performed earlier in higher-level entry functions such as `createPoolGauge`, `createVaultGauge`, or `createNoFeesGauge`. If either condition fails, the transaction reverts **after** unnecessary calldata handling, memory allocation, and partial execution, resulting in avoidable gas consumption.

BVSS

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

Recommendation

Perform these validations as early as possible.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** made a business decision to acknowledge this finding and not alter the contracts.

7.13 OUTDATED SOLIDITY PRAGMA

// INFORMATIONAL

Description

The contract specifies an outdated Solidity compiler version `0.8.24`. Newer Solidity versions in the `0.8.x` series (e.g., `0.8.29`) include multiple bug fixes, optimizer improvements, and safety enhancements compared to earlier releases. Continuing to use an older compiler version may cause the codebase to miss important fixes and improvements that are already available.

BVSS

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

Recommendation

Upgrade the Solidity pragma to the latest stable version.

Remediation Comment

SOLVED: The **Mezo team** solved this finding in the specified commit by following the mentioned recommendation.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/845bf8860d3c3252774e9d9194aba186a8570fe8

7.14 MISSING ZERO ADDRESS VALIDATION IN SETBOOSTER

// INFORMATIONAL

Description

The `setBooster` function in the `Voting Escrow` contract forwards the provided address directly to internal state without validating it. There is no check preventing `_booster` from being the zero address. If the booster address is unintentionally or maliciously set to `address(0)`, the contract may lose its configured booster permanently or enter an unintended state.

BVSS

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

Recommendation

Add an explicit zero address validation before setting the booster.

Remediation Comment

SOLVED: The **Mezo team** solved this finding in the specified commit by following the mentioned recommendation.

Remediation Hash

https://github.com/mezo-org/tigris-token-launch/commit/1fafbb0fcf93549380c8c1c4a5c4a89240fdd185

7.15 REDUNDANT STATE WRITE AND EVENT EMISSION

// INFORMATIONAL

Description

The `_setGrantManagerAllowed` function in `Grant` contract updates the `isGrantManagerAllowed` mapping and emits an event without checking whether the new value differs from the existing value.

If `_allowed` is already equal to `self.isGrantManagerAllowed[_grantManager]`, the function performs a redundant storage write and emits an event that does not represent a real state change. This can lead to unnecessary gas usage and misleading off-chain signals, as indexers and monitoring systems may interpret the event as a meaningful configuration update.

BVSS

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

Recommendation

Add a guard to ensure state is only updated when the value actually changes.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** made a business decision to acknowledge this finding and not alter the contracts.

7.16 CENTRALIZATION RISK DURING SETSTRATEGY()

// INFORMATIONAL

Description

The `MUSDSavingsRate` vault allows the contract owner to update the investment strategy via `setStrategy.` A malicious or compromised owner can set a strategy where:

- `deallocate()` is a no-op.
- `allocate()` sends funds to the owner.
- `claimYield()` or `migrate()` behave maliciously.

Once funds are migrated into such a strategy, **users can no longer withdraw their principal**, as withdrawals rely on `strategy.deallocate().` This results in **permanent loss of user funds**, entirely at the discretion of the owner.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (1.5)

Recommendation

Use a multisig for all high-impact functions.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** made a business decision to acknowledge this finding and not alter the contracts.

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.