Safe Edges

2025

Smart Contract Audit REPORT

ASSESSED ON Nov, 2025

Clique

Security Assessment

SAFEEDGES.IN

Count

Severity	Count	Status
CRITICAL		Fixed
HIGH	1	Fixed
MEDIUM	4	Fixed
LOW	5	Fixed
INFO	4	Fixed
GAS		Fixed
Total	14	Fixed

Findings

[H-01] Claimer Can Be Overcharged During Claim

Severity

High

Description:

In the `claim` function, users can redeem tokens they are entitled to via Merkle proof verification and signature validation. The function is designed to deduct a service fee from the claim process by calling `calculate_fee(amount, &ctx.accounts.fee_config)?`. However, this fee is computed based on the gross `amount` parameter rather than the actual claimable value (`claimable = released - claim_status.claimed`).

As a result, if the user has previously claimed a portion of their airdrop, subsequent claims will still be charged the full fee again, even though they are only receiving the smaller, remaining portion.

Impact:

A claimer performing incremental claims is overcharged on every subsequent claim, paying a disproportionately higher fee than intended.

Recommended Mitigation:

Calculate the fee using the actual claimable amount rather than the gross amount.

Status: Fixed

Team Response:

N/A

[M-01] Base Account Constraint Prevents Protocol Integration

Severity

Medium

Description:

The `base` account in the `Claim` instruction is constrained to `SystemAccount`, which restricts it to EOAs owned by the System Program. This prevents using PDAs, multisigs, or governance-controlled accounts as the distributor controller and blocks integrations where another protocol or DAO should own and operate the distributor. The account is also marked `mut` and is the recipient of system transfers, further limiting composability.

Impact:

Reduces composability and decentralization by preventing PDA- or governance-controlled distributors, limiting integrations with protocols/DAOs and forcing EOA-controlled deployments.

Recommended Mitigation:

Allow any account type for `base` by switching to `AccountInfo<'info>` (or a more flexible account type). If specific authority constraints are needed (e.g., PDA seeds or multisig checks), enforce them explicitly rather than via `SystemAccount`.

```
/// CHECK: Base account that controls the distributor (PDA, multisig, or EOA)
#[account(mut)]
pub base: AccountInfo<'info>,
```

Status: Fixed

Team Response:

N/A

[M-02] Accepting Mints With Freeze Authority Enables Post-Claim Freezes

Severity

Medium

Description:

If the distributor uses a mint with a configured freeze authority, that authority can freeze recipients token accounts (including ATAs) after claims, creating a denial-of-service and custody risk outside the programs control. The program does not restrict or validate the mints `freeze_authority` at initialization, and `set_mint` accepts a raw `Pubkey` without on-chain validation.

```
// new_distributor.rs
pub mint: Account<'info, Mint>, // no check on freeze_authority
distributor.mint = ctx.accounts.mint.key();
```

```
// set_mint.rs
pub fn set_mint<'info>(ctx: Context<SetMint>, mint: Pubkey) -> Result<()> {
```

```
 ctx.accounts.distributor.mint = mint; // no validation of freeze authority
 Ok(())
}
```

Impact: Recipients ATAs can be frozen post-claim, enabling external censorship/DoS by the freeze authority.

Recommended Mitigation:

Enforce no freeze authority at initialization and require validation when updating the mint. Accept the full `Mint` account in `set_mint` and verify `freeze_authority.is_none()` before assignment. Document that distributions must use mints without a freeze authority; if retention is necessary, make the distributor PDA the freeze authority and clearly disclose the risks.

```
// new_distributor.rs
require!(
 ctx.accounts.mint.freeze_authority.is_none(),
 ErrorCode::MintHasFreezeAuthority
);
```

```
// set_mint accounts and handler
#[derive(Accounts)]
pub struct SetMint<'info> {
#[account(mut, seeds = [MerkleDistributor::SEED, base.key().to_bytes().as_ref()], bump)]
 pub distributor: Account<'info, MerkleDistributor>,
 pub base: Signer<'info>,
 pub mint: Account<'info, Mint>, // validated mint account
 pub system_program: Program<'info, System>,
}
pub fn set_mint(ctx: Context<SetMint>) -> Result<()> {
 require!(
 ctx.accounts.mint.freeze_authority.is_none(),
 ErrorCode::MintHasFreezeAuthority
);
 ctx.accounts.distributor.mint = ctx.accounts.mint.key();
 Ok(())
}
```

Status: Fixed

Team Response: N/A

[M-03] Missing Chain Identifier Enables Cross-Chain Signature Replay Attacks

Severity

Medium

Description:

The claim signature lacks a mandatory domain separator (chain identifier). The signed message includes the distributor PDA but omits any chain-unique identifier (e.g., program ID or cluster/chain ID). Since the distributor PDA is deterministically derived from `[MerkleDistributor::SEED, base.key()]`, the same PDA can exist across deployments on different SVM chains when using the same base key. A signature valid on one chain (e.g., devnet) can be replayed on another (e.g., mainnet) if roots and signer are reused. The `extra` parameter is optional and unconstrained, making it unreliable as a domain separator without explicit on-chain enforcement.

```
// claim.rs excerpt
let msg = [
 handler.as_ref(),
 claimer.key().to_bytes().as_ref(),
 amount.to_le_bytes().as_ref(),
 released.to_le_bytes().as_ref(),
 root.as_ref(),
 ctx.accounts.distributor.key().to_bytes().as_ref(),
 extra.as_ref(),
].concat();
```

Impact:

Enables cross-chain signature replay, allowing double-claims across deployments before operators notice, potentially draining vaults and undermining the distributions economic model.

Recommended Mitigation:

Add explicit domain separation by binding signatures to a specific deployment. Include the program ID and a chain/cluster identifier in the signed message, or enforce a structured `extra` that contains and validates a chain ID committed on-chain (e.g., stored in distributor state). Prefer a non-optional, validated field for reliability.

```
// Example: include program ID and chain identifier in the signed message
let chain_id: u64 = /* retrieve from distributor state or config constant */;
let msg = [
 handler.as_ref(),
 claimer.key().to_bytes().as_ref(),
 amount.to_le_bytes().as_ref(),
 released.to_le_bytes().as_ref(),
 root.as_ref(),
 ctx.accounts.distributor.key().to_bytes().as_ref(),
&crate::ID.to_bytes(),        // bind to this program deployment
&chain_id.to_le_bytes(),      // bind to this chain/cluster
 extra.as_ref(),
].concat();
```

```
// Alternative: enforce structured `extra` carrying chain_id (first 8 bytes)
require!(extra.len() >= 8, ErrorCode::InvalidExtra);
let provided_chain_id = u64::from_le_bytes(extra[0..8].try_into().unwrap());
let expected_chain_id: u64 = /* retrieve from state or constant */;
require!(provided_chain_id == expected_chain_id, ErrorCode::InvalidChainId);
```

Status: Fixed

Team Response:

N/A

[M-04] Mutable Mint Allows Token Swap After Root Publication

Description:

`distributor.mint` can be changed after a Merkle root is published, but `claim` derives ATAs using the current `distributor.mint`. This allows operators to swap the asset mid-drop so that valid proofs pay out a different

mint than the one intended when the root was created. The Merkle root does not bind the mint, enabling accidental or malicious asset substitution.

Impact: Medium-High. Users may receive an unintended or worthless asset while proofs still verify, undermining trust. Conversely, valuable assets in the vault can be drained in exchange for junk assets if the mint is swapped.

Recommended Mitigation:

Make the mint immutable after initialization, or bind the expected mint to each published root and enforce equality during `claim`. Additionally, block `set_mint` once a root is active or claims have begun unless the distributor is paused and no dynamic roots are active. Add tests that attempt mint swaps mid-drop and ensure they are rejected.

Status: Fixed

Team Response: N/A

[L-01] No Fee Caps Allow Unbounded Fee Extraction

Severity

Low

Description:

The `set_fixed_fee` and `set_fee_per_tier` functions do not enforce maximum fee limits, allowing the base controller to set arbitrarily high fees that could exceed claim amounts.

```
// set_fixed_fee.rs - no cap on fee parameter
pub fn set_fixed_fee(ctx: Context<SetFixedFee>, configurator: [u8; 32], fee: u64)
// set_fee_per_tier.rs - no cap on rate, max_fee, or min_fee
pub fn set_fee_per_tier(
 ctx: Context<SetFeePerTier>,
 configurator: [u8; 32],
 rate: u64,
 max_fee: u64,
 min_fee: u64,
```

```
)
```

This allows fees that exceed claim amounts, preventing users from claiming their allocations.

Recommended Mitigation:

Add fee validation constraints:

```
const MAX_FEE_LAMPORTS: u64 = 1_000_000_000; // 1 SOL max
const MAX_FEE_RATE: u64 = 100_000_000; // 10% max (out of 1B basis)
require!(fee <= MAX_FEE_LAMPORTS, ErrorCode::FeeTooHigh);
require!(rate <= MAX_FEE_RATE, ErrorCode::RateTooHigh);
require!(max_fee <= MAX_FEE_LAMPORTS, ErrorCode::FeeTooHigh);
require!(min_fee <= max_fee, ErrorCode::InvalidFeeRange);
```

Status: Fixed

Team Response: N/A

[L-02] Misleading comment

Severity

Low

Description:

The docs/comments suggest the vault `stores` tokens, but in practice tokens live in the vaults Associated Token Account (ATA), and claims transfer from that ATA. The ATAs authority is `distributor.vault`, while the CPI uses the distributor PDA as the transfer authority, implying a delegate-based flow (owner is vault, delegate is distributor PDA). This mismatch between comments and actual design can confuse integrators.

```
/// The vault to store the tokens in.
 pub vault: SystemAccount<'info>,
```

Impact:

Misleading comment.

Recommended Mitigation:

In `new_distributor.rs`, change `The vault to store the tokens in` to `Authority for the distributors token account (tokens reside in the vaults ATA)`.

Status: Fixed

Team Response: N/A

[L-03] `set_mint` Lacks Mint Account Validation and Legacy Token Enforcement

Severity

Low

Description:

The program uses legacy SPL Token (v1) throughout (e.g., Token, TokenAccount, CPIs via anchor_spl::token) and validates a legacy Mint at initialization, but `set_mint` function accepts a raw Pubkey without owner/layout checks. This allows setting a Token-2022 mint or even a non-mint, which will later mismatch legacy Token CPIs and ATAs during claim, causing failures or undefined behavior.

Impact:

Lowto core protocol logic but operationally disruptive: misconfiguration can DoS claimants (claims fail), strand distributions, and

force redeployments/rollbacks. This is primarily a configuration risk rather than direct fund loss.

Recommended Mitigation:

Change `set_mint` to accept and validate a full Mint account alongside the legacy Token program, enforcing that the mints owner equals the provided legacy Token program and thereby ensuring a valid legacy mint.

Keep initialization strict by continuing to accept `Account<'info, Mint>` and applying the same owner constraint there. This binds all ATAs and CPIs to Token v1, prevents Token-2022/non-mint keys from being set, and aligns on-chain checks with Anchors owner/layout guarantees.

Status: Fixed

Team Response: N/A

[L-04] Dynamic Recipient Toggle Bricks Pending Claims

Severity

Low

Description:

Flipping `distributor.dynamic_recipient` from `true` to `false` after publishing a dynamic-root breaks all pending dynamic claims. The handler derivation during `claim` is keyed off the mutable `dynamic_recipient` flag rather than data committed to the Merkle root, so toggling the flag forces recomputation as `keccak(claimer_pubkey)` and invalidates proofs that were valid at publication.

Setter:

```
pub fn set_is_dynamic_recipient(
 ctx: Context<SetIsDynamicRecipient>,
 is_dynamic_recipient: bool,
) -> Result<()> {
 ctx.accounts.distributor.dynamic_recipient = is_dynamic_recipient;
 emit!(IsDynamicRecipientUpdatedEvent { /* ... */ });
 Ok(())
}
```

Impact: A single toggle bricks all dynamic claims, halting distributions and trapping user funds until a fresh root (or distributor) is deployed.

Recommended Mitigation:

Bind the handler mode to the published root instead of global mutable state. Store `dynamic_recipient` alongside each root on creation and require `claim` to use the flag committed in the root metadata. Reject toggling while a dynamic-configured root remains active, or enforce a phased rollout (e.g., clearing outstanding dynamic roots before switching to static). Add integration tests covering dynamic-to-static transitions to prevent regressions.

```
pub struct RootConfig {
 pub merkle_root: [u8; 32],
 pub dynamic_recipient: bool,
}
```

Status: Fixed

Team Response: N/A

[L-05] Missing event emission for a critical state update function

Severity

Low

Description: The `set_claim_root` instruction updates configuration (root and configurator) without emitting an event. This reduces transparency and makes it hard for indexers, explorers, and monitoring tools to track claim root rotations and who changed them.

```
pub fn set_claim_root<'info>(
 ctx: Context<SetClaimRoot>,
 root: [u8; 32],
 configurator: [u8; 32],
) -> Result<()> {
 let claim_root = &mut ctx.accounts.claim_root;
 claim_root.distributor = ctx.accounts.distributor.key();
 claim_root.root = root;
 claim_root.configurator = configurator;
 Ok(())
}
```

Impact: off-chain systems cant reliably detect claim root/configurator updates, weakening analytics, alerts, and audits.

Recommended Mitigation: Add a event like this

```
#[event]
pub struct ClaimRootUpdatedEvent {
 pub distributor: Pubkey,
 pub root: [u8; 32],
 pub configurator: [u8; 32],
// Optional but recommended for auditability:
// pub authority: Pubkey,
}
```

Status: Fixed

Team Response:

N/A

[I-01] Unused Imports

Severity

Informational

Description:

Several files contain unused imports that should be removed to improve code cleanliness and maintainability.

```
--> programs/merkle-distributor/src/instructions/new_distributor.rs:3:26
|
3 |     token::{Mint, Token, TokenAccount},
|                          ^^^^^^^^^^^^
4 |     token_interface::spl_token_metadata_interface::instruction::emit,
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|
= note: `#[warn(unused_imports)]` on by default
```

Impact: Informational. No direct security impact, but unused imports add noise to reviews, increase warning volume, and may obscure dead code or lead to confusion during refactors.

Recommended Mitigation:

Run `cargo clippy` and remove unused imports across the codebase. Consider enabling CI to treat clippy warnings as errors for unused imports to prevent regressions.

Status: Fixed

Team Response: N/A

[I-02] Missing State Change Guards

Severity

Informational

Description:

Setter functions (e.g., `set_mint`, `set_vault`, `set_signer`) do not verify that the provided value differs from the current on-chain state, allowing no-op updates that consume compute and fees while producing redundant events.

Impact: Informational/Low. No security impact, but wastes compute/fees, increases write contention, and can clutter event logs with redundant updates.

Recommended Mitigation:

Add guards to reject or early-return on no-op updates. For example:

```
pub fn set_mint(ctx: Context<SetMint>, mint: Pubkey) -> Result<()> {
 require!(ctx.accounts.distributor.mint != mint, ErrorCode::NoStateChange);
 ctx.accounts.distributor.mint = mint;
// ... emit event
 Ok(())
}
```

Apply analogous checks in `set_vault` and `set_signer` against their current values.

Status: Fixed

Team Response: N/A

[I-03] Vault Parameter Should Use SystemAccount Type

Severity

Informational

Description:

In `set_vault`, the vault parameter is provided as a raw `Pubkey` without on-chain type/owner validation. This allows setting an uninitialized or non-system account as the vault, which is a configuration hazard and can result in zero-initialized or incorrect destinations being recorded.

Impact:

Primarily an operability/configuration risk rather than direct fund loss. An invalid or zero-initialized vault can cause later transfers, accounting, or withdrawals to fail or behave unexpectedly (DoS-like until corrected).

Recommended Mitigation:

Require a `SystemAccount<'info>` for the new vault and assign the distributors `vault` from that accounts key. This ensures the account exists and is owned by the system program. Optionally add rent-exemption checks if the vault must maintain minimum balance guarantees.

```
#[derive(Accounts)]
pub struct SetVault<'info> {
#[account(mut, seeds = [MerkleDistributor::SEED, base.key().to_bytes().as_ref()], bump)]
 pub distributor: Account<'info, MerkleDistributor>,
 pub base: Signer<'info>,
 pub new_vault: SystemAccount<'info>,  // Ensure it's a valid system account
 pub system_program: Program<'info, System>,
}
pub fn set_vault(ctx: Context<SetVault>) -> Result<()> {
 ctx.accounts.distributor.vault = ctx.accounts.new_vault.key();
 Ok(())
}
```

Status: Fixed

Team Response: N/A

[I-04] Misspelled Helper Function `receover_signer` Hurts Clarity

Severity

Informational

Description:

The helper function that recovers the signer from a secp256k1 signature is misspelled as `receover_signer` instead of `recover_signer`. While functionally harmless, this reduces readability and risks confusion or propagation of the typo in future refactors and integrations.

Impact: No runtime/security impact, but degrades code clarity and increases the chance of human error during maintenance or external referencing.

Recommended Mitigation: Rename the function and all call sites from `receover_signer` to `recover_signer`. Keep the implementation unchanged.

Status: Fixed

Team Response: N/A

Safe Edges

REVOLUTIONIZING BLOCKCHAIN AUDITS

Founded in 2023, Safe Edges is the largest blockchain security company that ensures the safety and reliability of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative technology, we support the success of our clients with best-in-class security, all whilst realizing our overarching vision: ensuring blockchain remains secure and trustworthy.

THANK YOU FOR WORKING WITH US

SAFEEDGES.IN