



SMART CONTRACT SECURITY AUDIT

Clique

Sui Distributor

CLIENT

Clique

PLATFORM

Sui

LANGUAGE

Move

REVIEW

November 2025

DOCUMENT

Document information

Project	Clique, Sui Distributor
Client	Clique
Platform	Sui
Language	Move
Repository	Not recorded in the original report
Commit	Not recorded in the original report
Methods	Not recorded in the original report
Assessed	November 2025
Report date	10 December 2025

Document history

VERSION	DATE	CHANGE
1.0	10 December 2025	Original report issued View original report
1.1	October 2026	Reformatted into the Safe Edges standard template. Findings unchanged

This report was reformatted into the Safe Edges standard template in October 2026. Findings, severities, recommendations and code references are unchanged from the original report, which is preserved unchanged in the [Safe Edges Audit Portfolio on GitHub](#).

CONTENTS

Contents

01	Executive summary	3
02	Scope	4
03	Methodology	5
04	Severity and status	6
05	Findings summary	7
06	Detailed findings	8
	H-01 Claimed Tracking Uses Recipient Address resulting in Blocked Claims	8
	H-02 Handler Can Redirect Tokens Away From Offchain Approved Recipient	9
	M-01 Default Hook Collision with Strategies hook breaks Claim functionality temporarily	10
	M-02 Unbounded Claimed Table Growth	12
	L-01 Missing Input Validation in Fee Configuration Allows Invalid Fee Parameters	13
	L-02 Withdrawal Function Allows Token Extraction During Active Distribution	14
	L-03 Allowing removal of Active Merkle Root Configuration Blocks Valid Claims	15
	L-04 Fee Calculation Ignores Decimal Differences Between Airdrop and Fee Tokens	16
	L-05 Missing Signature Expiration Enables Indefinite Signature Validity	17
	I-01 Missing Wrapped Token handling as per Docs	18
	I-02 Typo in FeeMode Enum Variant Name	19
	I-03 Missing Access Control Enables Claim Denial of Service	20
	Disclaimer	23

Executive summary

About Clique

Clique's Sui distributor lets handlers claim Merkle-tree token allocations using off-chain signatures, with configurable fee modes and hooks that route claimed balances.

Summary of the review

Safe Edges reviewed Clique's token distributor for Sui, written in Move, in November 2025. Twelve issues were identified: two High, two Medium, five Low and three Informational. The High issues concerned claim tracking keyed on the recipient rather than the handler, which can block valid claims, and handlers being able to send claimed tokens away from the recipient approved off-chain. The original report does not record a resolution status for any finding.

Findings by severity



Findings by status

STATUS	ISSUES
● Not recorded	12

Scope

Repository	Not recorded in the original report
Commit	Not recorded in the original report
Platform	Sui
Language	Move

IN-SCOPE FILES

```
distributor.move  
base_distributor.move  
fee_mode.move  
free_hook.move
```

Only the code at the commit above was reviewed. Changes made after this commit are not covered by this report.

Methodology

Methods recorded for this engagement: Not recorded in the original report.

Manual review

Auditors read the in-scope code line by line, tracing control flow, state changes and trust boundaries against the intended behaviour described by the client.

Threat modelling

We map privileged roles, external dependencies and value flows, then test the assumptions the code makes about each of them.

Functional and automated testing

Where tooling exists for the language and chain, we run static analysis and targeted tests to confirm suspected issues and check edge cases.

Fix review

After the client addresses the findings, we review the updated code and record the final status of each issue.

Severity and status

Severity levels

SEVERITY	DEFINITION
Critical	Direct loss or permanent lock of funds, or full compromise of the protocol, with few or no preconditions. Must be fixed before deployment.
High	Significant loss of funds, broken core functionality or manipulation of critical state under realistic conditions. Should be fixed before deployment.
Medium	Limited loss or degraded functionality, or issues that need specific conditions or privileged access to exploit. Should be fixed.
Low	Minor impact, low likelihood, or deviations from best practice. Fix when practical.
Informational	Code quality, documentation, gas and style observations with no direct security impact.

Status definitions

STATUS	DEFINITION
Resolved	The issue was fixed and the fix was verified.
Partially resolved	The risk was reduced but not fully removed.
Acknowledged	The client accepted the issue without changing the code.
Open	The issue was not addressed at the time of the report.
Not recorded	The original report did not record a resolution status for this issue.

Findings summary

ID	TITLE	SEVERITY	STATUS
H-01	Claimed Tracking Uses Recipient Address resulting in Blocked Claims	High	● Not recorded
H-02	Handler Can Redirect Tokens Away From Offchain Approved Recipient	High	● Not recorded
M-01	Default Hook Collision with Strategies hook breaks Claim functionality temporarily	Medium	● Not recorded
M-02	Unbounded Claimed Table Growth	Medium	● Not recorded
L-01	Missing Input Validation in Fee Configuration Allows Invalid Fee Parameters	Low	● Not recorded
L-02	Withdrawal Function Allows Token Extraction During Active Distribution	Low	● Not recorded
L-03	Allowing removal of Active Merkle Root Configuration Blocks Valid Claims	Low	● Not recorded
L-04	Fee Calculation Ignores Decimal Differences Between Airdrop and Fee Tokens	Low	● Not recorded
L-05	Missing Signature Expiration Enables Indefinite Signature Validity	Low	● Not recorded
I-01	Missing Wrapped Token handling as per Docs	Informational	● Not recorded
I-02	Typo in FeeMode Enum Variant Name	Informational	● Not recorded
I-03	Missing Access Control Enables Claim Denial of Service	Informational	● Not recorded

Detailed findings

H-01 Claimed Tracking Uses Recipient Address resulting in Blocked Claims

Severity **High** Status ● Not recorded

DESCRIPTION

The protocol tracks claimed amounts using the recipient address (`on_behalf_of`) rather than the allocation owner (handler) in the claimed storage table. When `dynamic_recipient = true` (allowing users to claim on behalf of others), this creates a issue

However:

The final recipient is not fixed at Merkle-tree generation time so Merkle leaves bind token amount to the **handler**.

```
Leaf = keccak(handler || amount)
```

Claimed progress is recorded using (**merkle_root, on_behalf_of**) not (merkle_root, handler).

After the first handler claims for a recipient, the second handlers valid claim is rejected with `EAlreadyClaimed`, even though it corresponds to a separate Merkle leaf.

CODE REFERENCE: `BASE_DISTRIBUTOR::CLAIM_CHECK()`

```
let claimed_table_key = ClaimedTableKey { merkle_root, user: on_behalf_of };
let claimed_amount = if (self.claimed.contains(claimed_table_key)) {
    self.claimed[claimed_table_key]
} else {
    self.claimed.add(claimed_table_key, 0);
    0
};
```

IMPACT

Valid claims for distinct Merkle leaves become unclaimable when multiple handlers claim to the same recipient which means First handler's claim can blocks all subsequent handlers permanently.

ATTACK PATH

Merkle root R contains:

- Leaf A: (handler = H1, amount = 1000)
- Leaf B: (handler = H2, amount = 1000)

Distributor uses `dynamic_recipient = true`.

Both handlers try to route tokens to the same recipient X.

Suppose Released amount for each handler is 500.

- H1 claims 500 for X and Contract sets $\text{claimed}[(R, X)] = 500$.
- H2 attempts to claim 500 for X and Contract checks $\text{released}(500) > \text{claimed}[(R, X)](500)$ and returns False.
- Reverts with EAlreadyClaimed.
- Leaf Bs tokens become unclaimable
- Only workaround is for H2 to route to a different recipient

RECOMMENDED MITIGATION

We recommend tracking claimed amounts by handler instead of recipient in `claim_check`

```
- let claimed_table_key = ClaimedTableKey { merkle_root, user: on_behalf_of };  
+ let claimed_table_key = ClaimedTableKey { merkle_root, handler };
```

H-02 Handler Can Redirect Tokens Away From Offchain Approved Recipient

Severity **High** Status ● Not recorded

ASSUMPTION

Offchain will include the claim recipient `on_behalf_of` in signature explicitly to make sure handler send tokens to `on_behalf_of` recipient only during claim

DESCRIPTION

When `dynamic_recipient = true`, the Offchain signature explicitly specifies the intended recipient (`on_behalf_of`) as part of the signed message. However, after successful claim validation, hooks return extracted `Coin<T>` objects directly to the transaction sender (handler) without enforcing that tokens are transferred to the offchain-approved `on_behalf_of` address. This allows handlers to redirect tokens to arbitrary addresses, completely bypassing the offchain recipient authorization.

ATTACK PATH

Lets take simple example to understand this:

```
1. TeamLead Alice has 1M token allocation  
2. Offhchain signs 10 claims:  
   - on_behalf_of: Engineer_Bob (100k)  
   - on_behalf_of: Designer_Carol (100k)  
   - ... (10 team members)  
3. Offchain/Backend thinks: "I've authorized specific team members"  
4. Alice submits all claims but redirects ALL to her own wallet or some other address  
5. Team members get nothing, Alice gets everything
```

CODE REFERENCE

```
free_hook.move::free_balance(), distributor.move::extract_claimed_balance()
```

```
//free_hook.move
public fun free_balance<T>(claimed_balance: &mut ClaimedBalance<T>): Balance<T> {
    let hook_call_info = claimed_balance.hook_call_info<T, FreeHook>();
    let mut calldata = bcs::new(hook_call_info.calldata());
    let requested_amount = calldata.peel_u64();
    //Returns the tokens back to Handler and handler is free to choose whom to transfer
    extract_claimed_balance(
        claimed_balance,
        FreeHook {},
        requested_amount,
    )
}

//distributor.move
public fun extract_claimed_balance<T, H: drop>(
    claimed_balance: &mut ClaimedBalance<T>,
    _witness: H,
    requested_amount: u64,
): Balance<T> {
    //...
    //Returns the balance to free hook
    claimed_balance.remaining_balance.split(requested_amount)
}
```

IMPACT

Offchain signature explicitly specifies `on_behalf_of` recipient, but this is not enforced onchain. Currently the signature inclusion of `on_behalf_of` seems to be used only for tracking in claim function not for receiver of airdrop claimed tokens. So even though the offchain signature says send tokens to address X, the handler can take the tokens and send to any address.

RECOMMENDED MITIGATION

We recommend reviewing the architecture to enforce recipient authorization by storing `on_behalf_of` in the hot-potato `ClaimedBalance` and updating hooks to never return `Coin` objects. Instead, hooks must read `on_behalf_of` and perform transfers internally

M-01 Default Hook Collision with Strategies hook breaks Claim functionality temporarily

Severity **Medium** Status ● Not recorded

DESCRIPTION

The `claim()` function constructs a `VecMap` to store hook call information by inserting strategies first, then inserting the default hook. The `VecMap::insert` operation panics if attempting to insert a duplicate key. When the default hook type (DH generic parameter) matches any hook type present in strategies, the default hook insertion attempt triggers a panic via `assert!(!self.contains(&key), KeyErrorAlreadyExists)` during claiming tokens.

This occurs because there is no validation during `set_allocation_strategy()` to prevent configuring a strategy hook type that duplicates the default hook. Although `set_allocation_strategy` function will be

successfully and distributor can be activated but users Claim() will not work. This issue happens only when default hook equals one of strategies

CODE REFERENCE: DISTRIBUTOR.MOVE::CLAIM(), DISTRIBUTOR.MOVE::SET_ALLOCATION_STRATEGY()

```
//distributor.claim() function
// First, insert strategy hooks
let mut hook_call_infos = vec_map::empty();
strategies.zip_do!(hook_calldata, |strategy, calldata| {
    let max_allocation =
        (claimable.value() as u256) * (strategy.max_proportion as u256) /
        (STRATEGY_PROPORTION_DENOMINATOR as u256);
    let max_allocation = max_allocation as u64;
    hook_call_infos.insert(strategy.hook, HookCallInfo { max_allocation, calldata });
    // First insert succeeds for each unique strategy hook
});

// Then, insert default hook - PANICS if already exists
hook_call_infos.insert(
    default_hook, // @> If this equals any strategy.hook, PANIC!
    HookCallInfo { max_allocation: claimable.value(), calldata: default_hook_calldata },
);
```

STEPS TO REPRODUCE

- Allocation strategies are configured containing a hook type identical to the default hook (e.g., both set to FreeHook)
- Distributor successfully activated
- When user tries to claim tokens, they will get panic errors. This is because claim() call attempts to insert FreeHook twice into hook_call_infos – first from strategy and then default hook
- All claims revert with EKeyAlreadyExists from sui::vec_map, locking all funds indefinitely

IMPACT

User cannot claim the tokens unless additional strategy which is same as default hook strategy is removed from self.allocation_strategies. Protocol appears broken to users during misconfiguration period

RECOMMENDED MITIGATION

We recommend adding validation in set_allocation_strategy() to prevent configuring a default hook type that exists in the strategies vector:

```
public fun set_allocation_strategy<P, T, F, DH>(
    self: &mut Distributor<P, T, F>,
    configuration_id: Bytes32,
    strategies: vector<Strategy>,
) {
    assert!(
        strategies.all!(|s| s.max_proportion <= STRATEGY_PROPORTION_DENOMINATOR),
        EInvalidStrategies,
    );
    + // Validate default hook doesn't duplicate any strategy hook
    + let default_hook_type = type_name::with_defining_ids<DH>();
```

```

+   assert!(
+       !strategies.any!(|s| s.hook == default_hook_type),
+       EDefaultHookInStrategies
+   );
+   let allocation_strategy = AllocationStrategy {
+       strategies,
+       default_hook: type_name::with_defining_ids<DH>(),
+   };
+   if (self.allocation_strategies.contains(configuration_id)) {
+       *self.allocation_strategies.borrow_mut(configuration_id) = allocation_strategy;
+   } else {
+       self.allocation_strategies.add(configuration_id, allocation_strategy);
+   };
+   event::emit(AllocationStrategySet<P, T, F, DH> { configuration_id });
+ }

```

M-02 Unbounded Claimed Table Growth

Severity **Medium** Status ● Not recorded

DESCRIPTION

The claimed table in BaseDistributor stores an entry for every (merkle_root, user) pair created during claims. When a configuration is removed using `unset_configuration_id()`, the configuration ID mapping is deleted but the associated claimed entries in the claimed table are never cleaned up. These entries persist indefinitely in storage even when the merkle root is no longer actively used.

Over time, especially with multiple distribution rounds and large user bases, this creates unbounded table growth. Each new claim adds a permanent entry that consumes storage and increases the cost of table operations.

CODE REFERENCE: SOURCES/BASE_DISTRIBUTOR.MOVE

```

public struct BaseDistributor<phantom P, phantom T, phantom F> has store {
    id: UID,
    // ... other fields
    claimed: Table<ClaimedTableKey, u64>, // @> Grows indefinitely
    configuration_ids: Table<Bytes32, Bytes32>,
    // ... other fields
}

// Configuration can be removed
public fun unset_configuration_id<P, T, F>(
    self: &mut BaseDistributor<P, T, F>,
    _: &ProjectAdminCap<P>,
    merkle_root: Bytes32,
) {
    self.configuration_ids.remove(merkle_root); // Removes config mapping
    // @> Does NOT clean up claimed entries for this merkle_root
    event::emit(ConfigurationIdUnset<P, T, F> { merkle_root });
}

```

IMPACT

Storage size keeps increasing which leads to storage costs and higher gas costs

RECOMMENDED MITIGATION

Review the architecture to consider this approach -

Add an administrative function to clean up claimed entries for permanently concluded merkle tree airdrop distributions when they are all claimed (we can use bitmap to track the merkle tree airdrop claims). Do not delete claimed entries immediately on `unset_configuration_id()`, if admin reassigns the same merkle root to a new configuration, users could claim twice so make sure only when merkle tree are all claimed and concluded then only remove them.

L-01 Missing Input Validation in Fee Configuration Allows Invalid Fee Parameters

Severity **Low** Status ● Not recorded

DESCRIPTION

The `set_fee_mode()` function in the fee mode module lacks input validation, allowing administrators to configure fee parameters that violate logical constraints. Specifically, the function permits:

1. Inverted fee bounds: `min_fee > max_fee` (minimum exceeds maximum)
2. Invalid fee rates: `fee_rate > FEE_RATE_DENOMINATOR` (exceeds 100%)

These misconfigurations result in incorrect fee calculations and inconsistent behavior during claims, but the contract accepts them without any error.

CODE REFERENCE

```
public fun set_fee_mode<P, T, F>(  
    self: &mut BaseDistributor<P, T, F>,  
    _: &ProjectAdminCap<P>,  
    merkle_root: Bytes32,  
    fee_mode: FeeMode,  
) {  
    // No validation of fee_mode contents  
    if (self.fee_modes.contains(merkle_root)) {  
        *self.fee_modes.borrow_mut(merkle_root) = fee_mode;  
    } else {  
        self.fee_modes.add(merkle_root, fee_mode);  
    };  
}
```

IMPACT

ProjectAdmin can successfully configure invalid fee parameters without misconfiguration error message. This issue only discovered when users attempt claims.

RECOMMENDED MITIGATION

We recommend adding comprehensive validation during fee mode creation and update to prevent invalid configurations:

```
assert!(min_fee <= max_fee, EInvalidFeeRange);
assert!(
    (*fee_rate as u256) <= FEE_RATE_DENOMINATOR,
    EInvalidFeeRate
);
```

L-02 Withdrawal Function Allows Token Extraction During Active Distribution

Severity Low Status ● Not recorded

DESCRIPTION

The `withdraw_airdrop_token()` function permits holders of `AirdropTokenWithdrawCap` to withdraw tokens from the vault at any time, including during active claim windows when the distributor is operational and users are actively claiming.

CODE REFERENCE: `BASE_DISTRIBUTOR.MOVE`

```
public fun withdraw_airdrop_token<P, T, F>(
    self: &mut BaseDistributor<P, T, F>,
    _: &AirdropTokenWithdrawCap<P>,
    amount: u64,
    ctx: &mut TxContext,
): Coin<T> {
    // No check if distributor is active
    // No check if within claim window (start_time_ms to end_time_ms)
    // Can withdraw while users are claiming
    event::emit(AirdropTokenWithdrawn<P, T, F> { amount });
    coin::take(&mut self.vault, amount, ctx)
}
```

IMPACT

Users cannot claim tokens as vault tokens were already withdrawn by `AirdropTokenWithdrawCap` capability holder.

RECOMMENDED MITIGATION

We recommend restricting withdrawals to only occur when the distribution is inactive or the claim window has closed:

```
+ // Add new error constant
+ const EWithdrawalDuringActiveDistribution: u64 = 12;

public fun withdraw_airdrop_token<P, T, F>(
    self: &mut BaseDistributor<P, T, F>,
    _: &AirdropTokenWithdrawCap<P>,
+   clock: &Clock,
    amount: u64,
    ctx: &mut TxContext,
): Coin<T> {
+   // Only allow withdrawal when distribution is inactive OR after end time
+   let now = clock.timestamp_ms();
+   assert!(
+       !self.active || now > self.end_time_ms,
+       EWithdrawalDuringActiveDistribution
+   );
    event::emit(AirdropTokenWithdrawn<P, T, F> { amount });
    coin::take(&mut self.vault, amount, ctx)
}
```

L-03 Allowing removal of Active Merkle Root Configuration Blocks Valid Claims

Severity **Low** Status ● Not recorded

DESCRIPTION

The `unset_configuration_id()` function allows administrators to remove the configuration mapping for a merkle root at any time, including when users still have outstanding claimable tokens tied to that root. Once unset, the `claim()` function fails during configuration lookup preventing affected users from claiming their allocated tokens. The funds remain locked in the vault with no recovery mechanism unless the administrator manually restores the configuration.

CODE REFERENCE: `BASE_DISTRIBUTOR.MOVE`

```
public fun unset_configuration_id<P, T, F>(
    self: &mut BaseDistributor<P, T, F>,
    _: &ProjectAdminCap<P>,
    merkle_root: Bytes32,
) {
    // No check if users still have outstanding claims for this root
    // No validation that all allocated tokens have been claimed
    self.configuration_ids.remove(merkle_root);
    event::emit(ConfigurationIdUnset<P, T, F> { merkle_root });
}
```

IMPACT

Users with valid proofs for that merkle_root cannot claim, funds stay in vault.

RECOMMENDED MITIGATION

Track outstanding unclaimed amount per merkle_root and prevent `unset_configuration_id` until outstanding is zero, or require explicit admin confirmation and a delay.

L-04 Fee Calculation Ignores Decimal Differences Between Airdrop and Fee Tokens

Severity **Low** Status ● Not recorded

DESCRIPTION

When the fee is SingerTier with a fee rate, the fee calculation in `required_fee()` applies the fee rate directly to the raw airdrop token amount without adjusting for decimal place differences between the airdrop token (T) and fee token (F). The fee calculation uses the airdrop token's decimal precision but outputs in the fee token's decimal precision, causing fees to be incorrect by a factor of 10^n where $n = \text{fee_decimals} - \text{airdrop_decimals}$

Lets take an example: Airdrop Token Has More Decimals compared to Fee Token

```
Setup:
- Airdrop Token: USDC (6 decimals - available on Sui via Wormhole)
  → 1 USDC = 1,000,000 smallest units
- Fee Token: SUI (9 decimals - Sui native token)
  → 1 SUI = 1,000,000,000 smallest units
- Fee Rate: 1%
User Claims: 1,000 USDC
```

WHAT SHOULD HAPPEN (CORRECT)

```
1,000 USDC claim → 1% fee = 10 USDC worth of SUI
If 1 USDC = 0.5 SUI (market rate):
Expected fee = 5 SUI
= 5,000,000,000 smallest units of SUI ( $5 \times 10^9$ )
```

WHAT ACTUALLY HAPPENS (BUG)

```
Current calculation:
- Takes claim amount: 1,000 USDC = 1,000,000,000 units ( $1000 \times 10^6$ )
- Calculates 1%: 10,000,000 units
- Treats result as SUI units (but it's in USDC decimal scale!)
- Result: 0.01 SUI ( $10,000,000 / 10^9$ )
Protocol receives: 0.01 SUI instead of 5 SUI
Protocol loses: 99.8% of fee revenue
```

CODE REFERENCE: FEE_MODE.MOVE::REQUIRED_FEE()

```
const FEE_RATE_DENOMINATOR: u256 = 1_000_000_000_000_000_000; // 10^18

public fun required_fee(self: &FeeMode, amount: u64): u64 {
    match (self) {
        FeeMode::Fixed(fee) => *fee,
        FeeMode::SingerTier { fee_rate, min_fee, max_fee } => {
            // @> Uses amount directly without decimal adjustment
            let fee = ((amount as u256) * (fee_rate as u256) / FEE_RATE_DENOMINATOR) as u64;
            if (fee <= min_fee) {
                min_fee
            } else if (fee >= max_fee) {
                max_fee
            } else {
                fee
            }
        },
    }
}
```

IMPACT

Fee calculation can be incorrect if decimals are not same between airdrop token and fee token. It can undercharge or overcharge based on the difference between fee token decimals and airdrop token decimals

RECOMMENDED MITIGATION

Based on what kind of tokens are supported in this protocol, we recommend either enforcing same decimals tokens or implement decimal adjusted fee calculation by storing both token decimal information and adjusting the fee formula accordingly:

L-05 Missing Signature Expiration Enables Indefinite Signature Validity

Severity Low Status ● Not recorded

DESCRIPTION

The signature verification mechanism is missing two important security features that are standard in modern signing protocols and also referenced in the project documentation:

1. No Signature Expiration: The protocol documentation claims signatures include a deadline parameter for time-bound validity, but the implementation contains no deadline field or timestamp validation. This allows signatures to remain valid indefinitely after issuance.
2. No Chain ID and Domain Separator: The signature scheme lacks explicit chain identifier and domain separator fields similar to EIP-712. While the inclusion of distributor_id (Sui's globally unique UID) provides functional replay protection, the absence of explicit chain binding and protocol-level domain separation limits defense-in-depth and future extensibility.

Reference: <https://docs.clique.tech/token-distribution/quickstart/schedules-and-releases#signer-controlled-release>

CODE REFERENCE: BASE_DISTRIBUTOR.MOVE::SIGNATURE_CHECK()

```

fun signature_check(
    signature: &vector<u8>,
    signer_pubkey: &vector<u8>,
    handler: &Bytes32,
    on_behalf_of: address,
    amount: u64,
    released: u64,
    merkle_root: &Bytes32,
    distributor_id: &UID,
    extra: &vector<u8>,
    //NO deadline parameter
    //NO Clock parameter for timestamp validation
) {
    // ...
    //@> deadline is NOT included in signature and No timestamp check against current time
    assert!(
        ecdsa_k1::secp256k1_verify(signature, signer_pubkey, &message_hash, 1),
        EInvalidSignature,
    );
}

```

IMPACT

Signatures do not expire once issued and remain valid indefinitely. Any future flaw can allow an attacker to reuse old signatures to exploit the system.

RECOMMENDED MITIGATION

We recommend implementing signature expiration by adding deadline parameter and timestamp validation. Additionally, we recommend adding Chain ID and Domain Separator to the signature scheme.

I-01 Missing Wrapped Token handling as per Docs

Severity Informational Status ● Not recorded

DESCRIPTION

The protocol documentation describes wrapped token support via an isWrappedToken flag in the launch checklist, but this feature is not present in current codebase implementation as we could see the BaseDistributor struct contains a dynamic_recipient flag but no corresponding isWrappedToken field or any unwrapping logic.

Reference: <https://docs.clique.tech/token-distribution/quickstart/dry-run-and-go-live#launch-checklist>

CODE REFERENCE: BASE_DISTRIBUTOR.MOVE

```
public struct BaseDistributor<phantom P, phantom T, phantom F> has store {
    id: UID,
    active: bool,
    dynamic_recipient: bool,           // @> dynamicRecipient exists but isWrappedToken is
missing
    signer_pubkey: vector<u8>,
    vault: Balance<T>,
    collected_fee: Balance<F>,
    claimed: Table<ClaimedTableKey, u64>,
    configuration_ids: Table<Bytes32, Bytes32>,
    fee_modes: Table<Bytes32, FeeMode>,
    default_fee_mode: FeeMode,
    start_time_ms: u64,
    end_time_ms: u64,
}
```

IMPACT

Users following documentation expect wrapped token supported in this protocol

RECOMMENDED MITIGATION

Update the docs or update the code logic to accomodate this flag and handle wrapped tokens

I-02 Typo in FeeMode Enum Variant Name

Severity Informational Status ● Not recorded

DESCRIPTION

The enum variant is named SingerTier, which looks like a typo. The intended name is SingleTier. Not a functional bug, but confusing and hurts readability.

CODE REFERENCE: FEE_MODE.MOVE

```
public enum FeeMode has store, copy, drop {
    Fixed(u64),
    SingerTier {           // Typo: Should be "SingleTier"
        fee_rate: u128,
        min_fee: u64,
        max_fee: u64,
    },
}
```

IMPACT

Reduces code readability and professionalism

RECOMMENDED MITIGATION

Rename variant and update all references.

```
// sources/fee_mode.move
public enum FeeMode has store, copy, drop {
    Fixed(u64),
-   SingerTier {
+   SingleTier {
        fee_rate: u128,
        min_fee: u64,
        max_fee: u64,
    },
}

public fun single_tier(fee_rate: u128, min_fee: u64, max_fee: u64): FeeMode {
-   FeeMode::SingerTier { fee_rate, min_fee, max_fee }
+   FeeMode::SingleTier { fee_rate, min_fee, max_fee }
}

public fun required_fee(self: &FeeMode, amount: u64): u64 {
    match (self) {
        FeeMode::Fixed(fee) => *fee,
-       FeeMode::SingerTier { fee_rate, min_fee, max_fee } => {
+       FeeMode::SingleTier { fee_rate, min_fee, max_fee } => {
            // ... calculation logic
        },
    }
}
```

I-03 Missing Access Control Enables Claim Denial of Service

Severity **Informational** Status ● Not recorded

DESCRIPTION

The functions `set_allocation_strategy()` and `unset_allocation_strategy()` can be called without any admin capability (e.g., `ProjectAdminCap<P>`). Since Distributor objects are often shared publicly for airdrops, this lack of access control allows any caller to overwrite allocation strategies for active `configuration_ids` or remove existing strategies.

CODE REFERENCE: DISTRIBUTOR.MOVE::SET_ALLOCATION_STRATEGY(),
DISTRIBUTOR.MOVE::UNSET_ALLOCATION_STRATEGY()

```
public fun set_allocation_strategy<P, T, F, DH>(  
    self: &mut Distributor<P, T, F>,  
    configuration_id: Bytes32,  
    strategies: vector<Strategy>,  
) {  
    // NO capability like ProjectAdminCap<P> added here, so anyone can call this function  
    // ...  
}  
  
public fun unset_allocation_strategy<P, T, F>(  
    self: &mut Distributor<P, T, F>,  
    configuration_id: Bytes32,  
) {  
    // same here as well. Anyone can call this function and remove strategies  
    // ...  
}
```

IMPACT

Users cannot claim their tokens if a malicious actor repeatedly triggers these functions with incorrect hook strategies or by removing existing strategies.

ATTACK PATH

1. Attacker publishes a hook module AttackerHook exposing a witness type AttackerHookWitness.
2. Attacker calls (publicly) `Distributor::set_allocation_strategy::<P,T,F,AttackerHookWitness>(distributor_ref, config_id, strategies)`. No capability required.
3. default_hook becomes attacker's hook. Legitimate hooks cannot provide the attacker's witness; extract_claimed_balance will not succeed for normal hooks, preventing claims. Vault and collected fees remain locked. Attacker can repeat to re-lock funds.

RECOMMENDED MITIGATION

Add capability (`ProjectAdminCap<P>`) as a parameter in these functions. Similarly update for unset_allocation_strategy as well

```
public fun set_allocation_strategy<P, T, F, DH>(  
    self: &mut Distributor<P, T, F>,  
    _: &ProjectAdminCap<P>,          // Add this  
    configuration_id: Bytes32,  
    strategies: vector<Strategy>,  
) {  
    // ...  
}  
  
public fun unset_allocation_strategy<P, T, F>(  
    self: &mut Distributor<P, T, F>,  
    _: &ProjectAdminCap<P>,          // Add this  
    configuration_id: Bytes32,  
) {  
    // ...  
}
```

Disclaimer

This report is based on the code at the commit listed in the Scope section and on information the client provided during the review. It reflects Safe Edges' assessment at the time of the review and is not a guarantee that the code is free of vulnerabilities.

Changes made after the reviewed commit, deployment configuration, off-chain components and third-party dependencies are outside the scope of this report unless stated otherwise.

Security is ongoing. We recommend continued testing, monitoring and bug bounty coverage alongside any audit.

This report is not investment advice and is not an endorsement of the project, its team or any token.



Safe Edges is a Web3 security firm and official security partner for the Fuel, ENI and XI EcoChain ecosystems. We work across smart contract audits, Rust and WASM reviews, ZK circuit review, AI and MCP security, and penetration testing.

200+

PROJECTS AUDITED

\$4B+

TVL SECURED

0

POST-AUDIT HACKS