OtterSec

Merkle Token Distributor

Security Assessment

May 8th, 2025 — Prepared by OtterSec

Bartłomiej Wierzbiński dark@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

EthSign engaged OtterSec to assess the `merkle-token-distributor` program. This assessment was conducted between April 1st and April 16th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 11 findings throughout this audit engagement.

In particular, we identified several critical vulnerabilities, inlcuding one where the functionality for retrieving the distribution signer is publicly accessible, allowing anyone to reconstruct a signer for any distributor and withdraw its funds (OS-MTD-ADV-00), and another issue in the verification and claiming logic, which fails to check the result of Merkle proof verification, allowing anyone to submit invalid proofs and steal funds (OS-MTD-ADV-01).

Furthermore, the deployment logic is currently susceptible to a front-running attack, enabling an attacker to preemptively deploy with the same project ID, causing the legitimate deployment to fail (OS-MTD-ADV-02). Additionally, the fee calculation function may overflow when multiplying large `u64` values (OS-MTD-ADV-04).

We also made recommendations for implementing proper validations (OS-MTD-SUG-00) and explicit checks to improve overall error handling (OS-MTD-SUG-01). We further suggested updating the codebase for improved functionality, efficiency, and overall clarity (OS-MTD-SUG-02). Lastly, we advised adhering to coding best practices (OS-MTD-SUG-03) and removing redundant or unutilized code instances (OS-MTD-SUG-05).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/EthSign/merkle-token-distributor-move. This audit was performed against commit 8603a29.

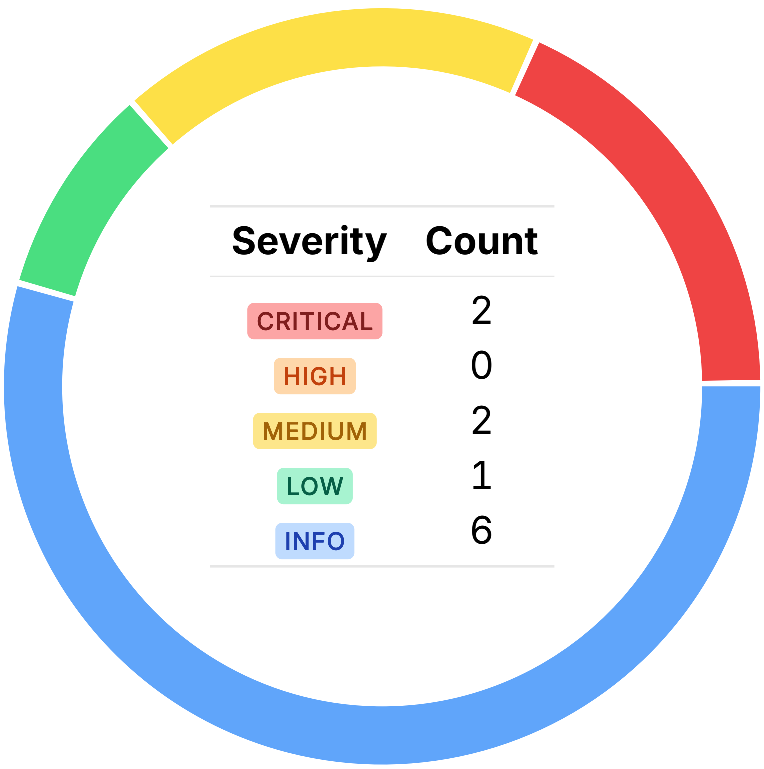
A brief description of the program is as follows:

Name	Description
merkle-token-distributor	A program for secure token airdrops utilizing Merkle proofs, enabling users to claim tokens by submitting a valid proof that verifies their entitlement without storing individual claims on-chain.

03 — Findings

Overall, we reported 11 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	2
HIGH	0
MEDIUM	2
LOW	1
INFO	6

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-MTD-ADV-00	CRITICAL	RESOLVED	`get_distribution_signer` is publicly accessible, allowing anyone to reconstruct a signer for any distributor and withdraw its funds.
OS-MTD-ADV-01	CRITICAL	RESOLVED	`verify_and_claim` fails to check the result of Merkle proof verification, allowing anyone to submit invalid proofs and steal funds.
OS-MTD-ADV-02	MEDIUM	RESOLVED	`deploy` is vulnerable to front-running attacks, where an attacker may preemptively deploy with the same `project_id`, failing the legitimate deployment.
OS-MTD-ADV-03	MEDIUM	RESOLVED	Setting `fee_bips` to `MAX_FEE_BIPS` results in `get_fee` charging zero fees due to a flawed check, allowing distributors to bypass fee payments.
OS-MTD-ADV-04	LOW	RESOLVED	It is possible for `get_fee` to overflow when multiplying large `u64` values.

Unrestricted Signer Access CRITICAL OS-MTD-ADV-00

Description

`get_distribution_signer` in `md_create2` exposes a critical vulnerability as it is currently publicly accessible. This allows any external caller to retrieve a signer for any distributor resource account utilizing the stored `SignerCapability`. With this signer, an attacker may execute privileged entry functions on behalf of the distributor. It will be possible for the attacker to call the primary fungible store API and withdraw funds stored by the distributor. Only modules trusted to act on behalf of distributors should be allowed to utilize this function.

>_ *sources/md_create2.move* RUST

```rust
public fun get_distribution_signer(
    distribution_addr: address
): signer acquires DistributorFactory, ModuleInfo {
    let module_addr = get_module_address();
    let factory = borrow_global<DistributorFactory>(module_addr);
    let signer_cap = table::borrow(&factory.signer_caps, distribution_addr);
    account::create_signer_with_capability(signer_cap)
}
```

Remediation

Mark `get_distribution_signer` as a `friend` function.

Patch

Resolved in 25238ec.

Failure to Abort on Incorrect Merkle Proof CRITICAL OS-MTD-ADV-01

Description

`verify_and_claim` in `merkle_distributor` is vulnerable as it calls `merkle_verifier::verify` but does not check its boolean return value. As a result, invalid Merkle proofs are not rejected, allowing anyone to submit a fraudulent proof and still claim tokens. This enables malicious users to steal funds by bypassing the intended verification process.

>_ *sources/merkle_distributor.move* RUST

```rust
fun verify_and_claim([...]): (address, u64) acquires DistributionData {
    [...]
    merkle_verifier::verify(proof, distribution_data.root, leaf);
    table::add(&mut distribution_data.used_leaves, leaf, true);
    let decoded_data = decode_leaf_data(data);
    if (decoded_data.claimableTimestamp > current_time
        || decoded_data.claimableTimestamp < distribution_data.start_time) {
        abort(decoded_data.claimableTimestamp);
    };
    (distribution_data.token, decoded_data.claimableAmount)
}
```

Remediation

Check the boolean value returned by `merkle_verifier::verify`, and ensure `verify_and_claim` aborts if the Merkle verification fails.

Patch

Resolved in 3fc010d.

DOS via Front-Running Deployment Transaction MEDIUM OS-MTD-ADV-02

Description

`md_create2::deploy` allows the possibility to front-run a deployment by deploying the same `project_id` before the original deployer's transaction is finalized. The deployments table addition will abort on duplicate `project_id` so if another transaction deploys it first, the original deployer's attempt will fail. This results in a denial of service for the legitimate deployer, preventing them from creating the distributor.

>_ sources/md_create2.move RUST

```rust
// Deploy a new distributor instance
public entry fun deploy(
    caller: &signer, project_id: String
) acquires DistributorFactory, ModuleInfo {
    [...]
    move_to(&resource_signer, distributor_data);
    table::add(&mut factory.signer_caps, resource_addr, resource_cap);
    table::add(&mut factory.deployments, copy project_id, resource_addr);
    [...]
}
```

Remediation

Update the logic to ensure deployments utilize non-colliding, unique identifiers.

Patch

Acknowledged by the EthSign team as an intended design choice so as to align with the Solidity version.

Improper Fee Collection Logic on Max Fee Bips MEDIUM OS-MTD-ADV-03

Description

There is a flaw in the program's current implementation of the fee collection logic. In `fee_collector::get_fee`, the fee calculation logic checks for the `fee_bips` value to determine the fee. If `fee_bips` equals `MAX_FEE_BIPS`, it implies that the fee should be the maximum allowed. However, currently, when the `fee_bips` is set to the maximum value (`MAX_FEE_BIPS`), the function returns zero (`if (fee_bips == MAX_FEE_BIPS)  return 0`), implying no fee will be charged.

>_ *sources/fee_collector.move* RUST

```rust
public fun get_fee(
    distributor_address: address, token_transferred: u64
): u64 acquires FeeCollectorData, ModuleInfo {
    [...]
    if (fee_bips == 0) {
        return fee_data.default_fee
    } else if (fee_bips == MAX_FEE_BIPS) {
        return 0
    };
    (token_transferred * fee_bips) / BIPS_PRECISION
}
```

Remediation

Update the condition in `get_fee` such that the function charges the maximum fee when `MAX_FEE_BIPS` is set.

Patch

This was acknowledged by Ethsign as an intended design choice, where setting `fee_bips` to `MAX_FEE_BIPS` implies zero fees

Risk of Overflow During Fee Calculation LOW OS-MTD-ADV-04

Description

`fee_collector::get_fee` risks a `u64` integer overflow as it multiplies two `u64` values (`token_transferred` and `fee_bips`) without scaling when computing `(token_transferred * fee_bips) / BIPS_PRECISION`. This will overflow, especially if large token amounts are involved, resulting in a runtime panic.

>_ *sources/fee_collector.move* RUST

```rust
public fun get_fee(
    distributor_address: address, token_transferred: u64
): u64 acquires FeeCollectorData, ModuleInfo {
    [...]
    (token_transferred * fee_bips) / BIPS_PRECISION
}
```

Remediation

Ensure `get_fee` properly scales during math operations to provide enough headroom to avoid math overflows.

Patch

Resolved in 6d988d7.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-MTD-SUG-00	There are several instances where proper validation is not performed, resulting in potential issues.
OS-MTD-SUG-01	Modifications to ensure the inclusion of explicit checks to prevent unexpected aborts or panics, improving the protocol's robustness and error handling.
OS-MTD-SUG-02	Recommendation for updating the codebase to improve functionality and efficiency.
OS-MTD-SUG-03	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-MTD-SUG-04	Code optimizations for better maintainability and readability.
OS-MTD-SUG-05	The codebase contains multiple cases of redundant and unutilized code that should be removed.

Missing Validation Logic

OS-MTD-SUG-00

Description

1. `merkle_distributor::update_merkle_root` should validate that the new Merkle root has the correct length to prevent storing invalid or malformed roots.

>_ *sources/merkle_distributor.move* RUST

```rust
public entry fun update_merkle_root(
    admin: &signer, distributor_address: address, new_root: vector<u8>
) acquires ModuleInfo, DistributionData {
    ownable::only_owner(admin, get_module_address());
    let distribution_data = borrow_global_mut<DistributionData>(distributor_address);
    distribution_data.root = new_root;
}
```

2. `merkle_distributor::initialize_distribution` should verify that `start_time` is less than the `end_time` to prevent creation of distributions with invalid claim periods, ensuring consistency with `update_time_range`, which reverts `DistributionData` creations with invalid parameters. It should also verify that `merkle_root` is correct to match expected Merkle tree hash sizes.

Remediation

Include the above validations in the codebase.

Patch

1. Issue #2 resolved in 7c85c1c.

Error Handling OS-MTD-SUG-01

Description

1. `get_fee_collector` and `get_fee_token` in `md_create2` currently assume that `DistributorData` exists at the provided address, which may result in runtime aborts if it does not. To improve error handling, these functions should check for resource existence `DistributorData` before borrowing.

>_ *sources/md_create2.move* RUST

```rust
public fun get_fee_collector(distribution_addr: address): address acquires DistributorData
    ↪ {
    let distributor_data = borrow_global<DistributorData>(distribution_addr);
    distributor_data.fee_collector
}

public fun get_fee_token(distribution_addr: address): address acquires DistributorData {
    let distributor_data = borrow_global<DistributorData>(distribution_addr);
    distributor_data.fee_token
}
```

2. In `merkle_distributor`, `charge_fee` should validate that the fee amount is not greater than the claim amount to avoid underflow. Currently, a static fee may exceed the claimed amount, resulting in a runtime abort. Add a check ensuring `fee_amount <= amount` to improve error handling. Also, `send` should check if the distributor has sufficient balance before attempting a transfer. This prevents unexpected failures.
3. `md_create2::get_distribution_address` directly borrows from the `deployments` table without checking if the `project_id` exists, which may result in a runtime error if the key is missing. Add an existence check to prevent panics and improve error handling.

>_ *sources/md_create2.move* RUST

```rust
public fun get_distribution_address(
    project_id: String
): address acquires DistributorFactory, ModuleInfo {
    let module_addr = get_module_address();
    let factory = borrow_global<DistributorFactory>(module_addr);
    *table::borrow(&factory.deployments, project_id)
}
```

Remediation

Update the codebase with explicit checks listed above to ensure proper error handling.

Patch

1. Issue #1 was acknowledged by the EthSign team.
2. Issue #2 was partially resolved as the EthSign team mentioned that the fee setting is done manually.

Code Refactoring OS-MTD-SUG-02

Description

1. `fee_collector` prioritizes custom fixed and custom bips-based fees for specific `distributor_address` values. However, once added, it is not possible to remove these custom fee entries. Fixed fees set to 0 are ignored, and bips fees of 0 default to the global fee instead of overriding or removing the entry. Without the ability to clean up obsolete fee entries, the `FeeCollectorData` tables will accumulate entries over time. Add functionality to remove custom fees and tokens from `FeeCollectorData`. Additionally, consider treating a 0 fee value as a valid override rather than defaulting to fallback logic.
2. The codebase repeatedly utilizes verbose `table::contains`, `borrow_mut`, and `add` sequences to update tables, as seen in functions such as `set_custom_fee_bips`, `set_custom_fee_token`, and `set_custom_fee_fixed` (as shown below). This approach increases boilerplate and reduces readability. These patterns may be replaced with `table::upsert` to improve code readability and maintainability.

>_ *sources/fee_collector.move* RUST

```rust
public entry fun set_custom_fee_fixed(
    caller: &signer, distributor_address: address, fixed_fee: u64
) acquires FeeCollectorData, ModuleInfo {
    [...]
    if (table::contains(&fee_data.custom_fees_fixed, distributor_address)) {
        *table::borrow_mut(&mut fee_data.custom_fees_fixed, distributor_address) =
            ↪ fixed_fee;
    } else {
        table::add(&mut fee_data.custom_fees_fixed, distributor_address, fixed_fee);
    };
    event::emit(CustomFeeSetFixed { distributor_address, fixed_fee });
}
```

3. `ownable` may be simplified by removing the `module_addr` parameter from its functions, as the `Owner` resource only resides at `@merkle_token_distributor`. It is initialized solely in `init_module` utilizing the `deployer` address, and no other address may hold the `Owner` resource.
4. Remove debug comments from all files to clean up the codebase, rendering the code easier to read and maintain without unnecessary clutter.
5. In `transfer_ownership`, the `OwnershipTransferred` event incorrectly includes the new owner in both the old and new owner fields due to reassignment of `owner_cap.owner` prior to emitting the event. This may create confusion and should be corrected.

Remediation

Incorporate the above-stated refactors.

Patch

1. Issue #1 was acknowledged by the Ethsign team, mentioning that the custom fee may be updated utilizing the setters.
2. Issue #2 resolved in `1f2405a`.
3. Issue #3 resolved in `e4bcbba`.
4. Issue #4 resolved by removing all debug comments.

Code Maturity

OS-MTD-SUG-03

Description

1. `DeployEvent` in `md_create2` should be annotated with `#[event]` to clearly indicate its purpose as an event.

>_ *sources/md_create2.move* RUST

```
struct DeployEvent has drop, store {
    project_id: String,
    distribution_address: address,
    deployer: address
}
```

2. Replace the pattern of `contains` and `borrow` calls with `borrow_with_default` in `fee_collector` functions such as `get_fee` to simplify code and improve readability by handling default logic in a single call.

>_ *sources/fee_collector.move* RUST

```
public fun get_fee(
    distributor_address: address, token_transferred: u64
): u64 acquires FeeCollectorData, ModuleInfo {
    let module_addr = get_module_address();
    let fee_data = borrow_global<FeeCollectorData>(module_addr);
    // Check for fixed fee first
    if (table::contains(&fee_data.custom_fees_fixed, distributor_address)) {
        let fixed_fee =
            *table::borrow(&fee_data.custom_fees_fixed, distributor_address);
        if (fixed_fee > 0) {
            return fixed_fee
        }
    };
    [...]
}
```

3. The current `ownable::transfer_ownership` logic immediately assigns a new owner, which risks locking the module if an incorrect address is inadvertently passed. A two-step process requiring the new owner to accept ownership will ensure a safer transfer approach.

4. All significant state-changing operations should emit an event, but only when a change has actually occurred. Events should be emitted in functions such as `initialize_distribution`, `update_merkle_root`, and `transfer_ownership`.

5. `merkle_verifier` and `merkle_distributor` utilize different hash functions, `keccak256` and `sha3_256` respectively, when processing Merkle tree data. This inconsistency mismatches in Merkle root computation and verification. Align the hashing strategy to prevent bugs and maintain correctness.

Remediation

Implement the above-mentioned suggestions.

Code Optimization OS-MTD-SUG-04

Description

1. In `merkle_verifier`, `compare_vectors` redundantly returns `len_a <= len_b` (as shown below) after asserting that lengths are equal, which is always true and misleading. It should instead return true to indicate full equality. Also, the function currently permits `a` and `b` to have differing lengths. A check should be added to ensure they are of equal length before proceeding. Further, the function name is vague and should be renamed for clarity. Also, in `hash_pair`, utilizing a double negation reduces readability and should be replaced with clearer logic.

   >_ *sources/merkle_verifier.move* RUST

   ```
   fun compare_vectors(a: &vector<u8>, b: &vector<u8>): bool {
       let i = 0;
       let len_a = vector::length(a);
       let len_b = vector::length(b);
       assert!(len_a == len_b, EINVALID_LEAF);
       [...]
       len_a <= len_b
   }
   ```

2. In multiple places across the codebase (such as in `md_create2`, which repeatedly borrows the `DistributorFactory` resource), borrowing of global resources may be extracted to a separate helper function to reduce code duplication and improve readability and maintainability.
3. The addresses of fee and distribution tokens are utilized solely to convert them into `Object<Metadata>` in `merkle_distribution::claim`. To simplify both logic and storage, the fee collector and distributor modules should store `Object<Metadata>` directly instead of the addresses.

Remediation

Include the above recommendations to optimize the logic.

Unutilized/Redundant Code

OS-MTD-SUG-05

Description

1. `merkle_distributor::deposit` is redundant, as it merely wraps `primary_fungible_store::transfer` without adding distribution-specific logic. It performs unnecessary actions by requiring a `distributor_address`, retrieving a signer for it, and then converting it back to an address. Deposit operations may be handled directly via the public primary fungible store API. Also, `merkle_distributor::get_module_address` is not utilized and may be removed.

   >_ *sources/merkle_distributor.move* RUST

   ```rust
   public entry fun deposit<T: key>(
       from: &signer,
       distributor_address: address,
       metadata: Object<T>,
       amount: u64
   ) {
       let distribution_signer =
           md_create2::get_distribution_signer(distributor_address);
       let distribution_addr = signer::address_of(&distribution_signer);
       // Transfer tokens to the distribution
       primary_fungible_store::transfer<T>(from, metadata, distribution_addr, amount);
   }
   ```

2. `ModuleInfo` structure in the various modules is unnecessary and should be removed, as it may only store the fixed `deployer` address (`@merkle_token_distributor`), since the `ModuleInfo` creation always happens in `init_module` utilizing the `deployer` address.

   >_ *sources/fee_collector.move* RUST

   ```rust
   struct ModuleInfo has key {
       module_address: address
   }
   ```

3. Several functions in `merkle_distributor` are currently written generically utilizing `<T: key>` and take `Object<T>` as a parameter. However, this generic abstraction is unnecessary and introduces extra complexity. Replace `Object<T>` with `Object<Metadata>`, to ensure type safety, and utilize `Metadata` in `address_to_object` calls.

4. `merkle_distributor::withdraw` redundantly derives the distribution address from the signer, even though `distributor_address` is already available and may be utilized directly for balance checks.

>_ sources/merkle_distributor.move RUST

```rust
public entry fun withdraw(
    caller: &signer,
    distributor_address: address,
    metadata: Object<Metadata>,
    amount: u64,
    withdraw_address: address
) {
    assert!(md_create2::is_distribution_owner(caller, distributor_address),
        ↪  ENOT_AUTHORIZED);
    let distribution_signer =
        md_create2::get_distribution_signer(distributor_address);
    let distribution_addr = signer::address_of(&distribution_signer);
    [...]
}
```

5. Several constants across the codebase are declared but unutilized and may be safely removed. Examples include `EINVALID_PROOF` in `merkle_verifier`, `EROOT_NOT_INITIALIZED`, `EDOES_NOT_EXIST`, and `EDISTRIBUTION_NOT_FOUND` in `merkle_distributor`, as well as all constants in `fee_collector`.
6. `fee_collector` declares `md_create2` as a `friend`, but no internal functions from `fee_collector` are actually utilized in `md_create2`. Since friend access is only required by `merkle_distributor`, this friend declaration is unnecessary and should be removed.
7. `merkle_distributor` unnecessarily specifies the `Metadata` type in `fungible_store` API calls (as shown in the example below), unlike `fee_collector`, which relies on type inference. Remove the explicit type definition.

>_ sources/merkle_distributor.move RUST

```rust
// Send tokens to a recipient
fun send([...]) {
    let distribution_signer =
        md_create2::get_distribution_signer(distributor_address);
    primary_fungible_store::transfer<Metadata>(
        &distribution_signer,
        metadata,
        recipient,
        amount
    );
}
```

8. `DistributorData` in `md_create2` stores `fee_token` data that is never utilized, since `merkle_distributor::charge_fee` relies on `fee_collector` to retrieve fee token info. This creates unnecessary duplication. To simplify code and avoid errors, the `fee_token` fields in `DistributorData` may be removed, and only `fee_collector` should manage fee token data.

Remediation

Remove the redundant and unutilized code instances highlighted above.

Patch

1. Issue #1 partially resolved in 49eaf58.
2. Issue #2 partially resolved in ceded44.
3. Issue #3 partially resolved in 54c11b0.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
HIGH	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
MEDIUM	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
LOW	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
INFO	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.