Pizza Drop Audit Report

0xOwain

August 30, 2025

Pizza Drop Audit Report

Version 1.0

0xOwain

August 30, 2025

Pizza Drop Audit Report

0xOwain

August 30, 2025

Prepared by: OwainPeters48 Lead Auditor: Owain Peters

Table of Contents

Protocol Summary

PizzaDrop is an Aptos/Move-based airdrop system that distributes APT tokens to registered users. Each user can claim exactly one allocation, determined randomly at registration. The contract includes functions for funding the pool, registering users, and claiming tokens.

Disclaimer

The 0xOwain team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Move implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

- pizza_drop::airdrop.move

Lines of code reviewed: ~180

Roles

Roles: - Owner: Deploys and funds the airdrop pool. - User: Registers for the airdrop and claims their allocation. - Contract (resource account): Holds APT and distributes it to users.

Executive Summary

The PizzaDrop contract was reviewed with a focus on access control, state management, and fairness of distribution. We identified 1 Medium severity issue related to predictable randomness, as well as 1 Low severity finding. Overall, the codebase is relatively small and straightforward, but improvements to randomness mechanisms and operational transparency are recommended.

Issues Found

- Medium: 1
- Low: 1

Findings

Medium (M-1): Predictable randomness via timestamp.

Low (L-1): Missing `Funded` event in `fund_pizza_drop`.

High

No High severity findings were identified in this audit.

Medium

[M-1] Predictable Randomness via Timestamp Allows Manipulation

Description The `get_random_slice()` function uses `timestamp::now_microseconds()` to generate a pseudo-random reward between 100 and 500 APT. However, timestamps are deterministic within a block and predictable across blocks. Since this function is invoked during registration by the contract owner, an attacker with influence over the registration timing can bias the outcome by choosing timestamps that favour higher rewards. This undermines the fairness of the airdrop by making the randomness manipulable.

Impact The use of `timestamp::now_microseconds()` as the sole source of randomness allows malicious actors to predict or brute-force high reward values. By repeatedly calling the `register_pizza_lover()` function, an attacker can: - Farm multiple addresses to extract only high-value airdrops (e.g., 500 APT instead of 100 APT), - Exploit predictable timestamp mod behaviour through scripting and off-chain simulation, - Collude with validators to control block timestamps and deterministically maximise rewards. In high-value deployments, it may result in financial loss, centralised reward capture, and reputational damage.

Proof of Concept: This vulnerability was demonstrated using the following unit test, which shows that reward values are entirely determined by the current timestamp, resulting in identical outcomes for multiple users registered within the same logical time window:

```
#[test(deployer = @pizza_drop, user1 = @0xabc, user2 = @0xdef, framework = @0x1)]
fun test_timestamp_based_randomness(deployer: &signer, user1: &signer, user2: &signer, frame
    use aptos_framework::timestamp;

    timestamp::set_time_has_started_for_testing(framework);

    // Register two users back-to-back
    register_pizza_lover(deployer, signer::address_of(user1));
    register_pizza_lover(deployer, signer::address_of(user2));

    // Fetch assigned amounts
    let amt1 = get_claimed_amount(signer::address_of(user1));
    let amt2 = get_claimed_amount(signer::address_of(user2));

    // Assert identical slice amounts
    assert!(amt1 == amt2, 999);
}
```

This test registers two separate users in quick succession and retrieves the reward amount assigned to each. Since both registrations occur within the same microsecond timestamp window - the randomness function (`get_random_slice()`) returns the same value to both users. The final assertion confirms this determinism. By extension, an attacker could simulate this pattern off-chain, identify when a high-value reward (e.g., 500 APT) would be granted, and then register multiple wallets during that same timestamp window to extract maximum value - fully bypassing the intended randomness of the airdrop.

Recommended Mitigation: To eliminate the exploitability of timestamp-based randomness, the contract should not rely solely on `timestamp::now_microseconds()` for generating pseudo-random values. Instead, consider the following mitigation options: 1. Commit-Reveal Scheme: Implement a two-phase process where: - In phase one, users commit to registering (by submitting a hash of their address + nonce). - In phase two, users reveal their original inputs, and rewards are assigned based on the hash. This prevents attackers from predicting outcomes at the time of registration.

2. VRF-based Randomness (Ideal): Use an external Verifiable Random Function (VRF) oracle (e.g., Chainlink VRF) to supply secure, tamper-proof randomness:
    - This makes it infeasible for attackers to predict or manipulate outcomes.
    - However, this adds cost and infrastructure complexity (e.g., off-chain oracle integration).

Low

[L-1] Missing `Funded` Event in `fund_pizza_drop`

Description The `fund_pizza_drop` function transfers tokens into the contract and updates the internal balance, but no event is emitted to record the deposit.

Impact Lack of event emission reduces transparency and makes it harder for off-chain services, auditors, or users to track funding activity. This can complicate monitoring and incident response.

Proof of Concept: Calling `fund_pizza_drop` with any amount succeeds, but there is no corresponding on-chain log for external observers to verify deposits.

Recommended Mitigation: Introduce and emit a dedicated `Funded` event whenever the pizza pool is replenished.