



Kode-n-Rolla

Aptos Pizza Drop Security Review

Aug 28th 2025 – Sep 4th 2025



Table of Contents

➤ Whoami.....	3
➤ Disclaimer.....	3
➤ Risk Classification.....	3
➤ Protocol Summary.....	4
➤ Audit Details.....	5
○ Scope.....	5
○ Roles.....	5
➤ Executive Summary.....	6
○ Issue found.....	7
➤ Findings.....	7
➤ High.....	7
➤ [H-01] Spec / Units Mismatch in APT Distribution....	7
➤ [H-02] Unrestricted Registration Enables Sybil Pool Drain.....	10
➤ Medium.....	13
➤ [M-01] Predictable Timestamp-Based Randomness Enables Payout Gaming.....	13



Whoami

I am Pavel (aka `kode-n-rolla`), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a [CodeHawks](#) audit contest.

The analysis focuses exclusively on the security aspects of the reviewed `Move` codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

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

I used [CodeHawks](#) severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact:*
 - o Funds are directly or nearly directly at risk



- o There's a severe disruption of protocol functionality or availability.
- *Medium Impact:*
 - o Funds are indirectly at risk.
 - o There's some level of disruption to the protocol's functionality or availability.
- *Low Impact:*
 - o Funds are not at risk.
 - o However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

PizzaDrop is a Move-based smart contract deployed on the **Aptos** blockchain that implements a token distribution mechanism using the native Aptos Coin (**APT**). The protocol allows eligible participants to claim a one-time token allocation of a randomized amount within predefined bounds.

The contract is administered by a privileged owner account responsible for registering eligible participants and funding the distribution pool. Only registered accounts are permitted to claim tokens, and each registered address is intended to be able to claim exactly once.

Upon a successful claim, the contract transfers a pseudo-random amount of APT-bounded between a minimum and maximum value-to the caller. Claims are only expected to succeed if the contract holds sufficient APT balance to cover the transfer.

From a security perspective, the protocol relies on correct enforcement of access control, participant registration logic, single-claim guarantees, and safe handling of native coin transfers. Additionally, the correctness and unpredictability of the randomization mechanism is critical to maintaining fairness and preventing manipulation of the distribution process.



Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the `CodeHawks` platform. The assessment focused exclusively on the `Move` smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "[Aptos Pizza Drop](#)" `CodeHawks` audit event.

Scope

The audit scope consisted of the following smart contract:

```
./sources/  
└─ pizza_drop.move  
./Move.toml
```

Roles

- Owner (PizzaCoin Team)

The Owner is a privileged account responsible for administering the PizzaDrop protocol. This role is authorized to register eligible participants, fund the distribution pool with APT, and manage access to the token distribution process.

- Participant (Pizza Lover)

A Participant is a registered account eligible to claim a one-time allocation of APT from the PizzaDrop pool. Participants are expected to be able to claim exactly once and only after being explicitly registered by the Owner.



Executive Summary

This security review evaluated the PizzaDrop smart contract on Aptos, focusing on access control, participant registration, randomness generation, and the correctness of token distribution logic.

The review identified two **high**-severity issues and one **medium**-severity issue that collectively undermine the protocol's core security and economic assumptions. A critical specification mismatch causes the advertised reward range of "**100-500 APT**" to be implemented as 100-500 octas, resulting in payouts that are several orders of magnitude smaller than intended. This discrepancy breaks user expectations and the documented behavior of the protocol.

Additionally, participant registration is not properly restricted. A publicly callable entry function enables unrestricted registration, allowing an attacker to create or register an arbitrary number of accounts and perform repeated claims. This issue enables **Sybil-style** attacks and uncontrolled depletion of the distribution pool.

A **medium**-severity issue was also identified in the randomness mechanism. The use of predictable, timestamp-based values allows attackers to influence or precompute favorable claim outcomes. When combined with unrestricted registration, this significantly accelerates pool drainage and further amplifies the impact of the **high**-severity issues.

Overall, the protocol fails to enforce critical access control, randomness, and specification guarantees. These issues can lead to unfair distribution, economic manipulation, and rapid exhaustion of the PizzaDrop pool, rendering the system unreliable in adversarial conditions.

Several of the identified issues compose into a cascading failure scenario rather than isolated weaknesses.

Issue found

Severity	Issue
High	Spec / Units Mismatch in APT Distribution
High	Unrestricted Registration Enables Sybil Pool Drain
Medium	Predictable Timestamp-Based Randomness Enables Payout Gaming

Findings

High

[H-01] Spec / Units Mismatch in APT Distribution

Description

The contract assigns and transfers a raw `u64` amount of AptosCoin, which is denominated in **octas** (where `1 APT = 100,000,000 octas`).

```
#[randomness]
entry fun get_random_slice(user_addr: address) acquires ModuleData, State {
  let state = borrow_global_mut<State>(get_resource_address());
  let time = timestamp::now_microseconds();
  let random_val = time % 401;
  @> let random_amount = 100 + random_val; // 100..500 OCTAS, not APT
  table::add(&mut state.users_claimed_amount, user_addr, random_amount);
}

public entry fun claim_pizza_slice(user: &signer) acquires ModuleData, State
{
  ...
  @> let amount = *table::borrow(&state.users_claimed_amount, user_addr); //
  OCTAS
  assert!(state.balance >= amount, E_INSUFFICIENT_FUND);
  ...
  @> transfer_from_contract(user_addr, amount); // sends OCTAS 1:1
  ...
}
```



As a result, users receive **100–500 octas** (approximately **0.000001–0.000005 APT**) instead of the advertised **100–500 APT**.

This mismatch originates from the absence of unit conversion between APT and its base denomination during reward calculation and transfer.

Impact

- **User Trust / UX:** Participants expecting 100–500 APT receive a negligible fraction of that amount, which may be perceived as misleading or broken behavior.
- **Operational Integrity:** Pool accounting, dashboards, and distribution metrics diverge from the intended economic design, leading to incorrect assumptions about remaining balances and payout effectiveness.
- **Economic Safety:** The protocol's effective distribution value is orders of magnitude lower than specified, masking the true financial exposure of the system.

Likelihood

- AptosCoin uses octas as its base unit, and the contract never applies a conversion factor (`OCTAS_PER_APT`).
- The issue affects **all claims** consistently and does not rely on specific conditions or attacker interaction.

Proof of Concept

```
#[test(deployer = @pizza_drop, user = @0xABC, framework = @0x1)]
fun poc_unit_scale_is_octas_not_full_APT(
  deployer: &signer,
  user: &signer,
  framework: &signer
) acquires State, ModuleData {
  use aptos_framework::timestamp;
  use aptos_framework::account;
  use aptos_framework::aptos_coin;
  use aptos_framework::coin;

  timestamp::set_time_has_started_for_testing(framework);
  let (burn_cap, mint_cap) = aptos_coin::initialize_for_test(framework);
```



```

account::create_account_for_test(@pizza_drop);
account::create_account_for_test(signer::address_of(user));
init_module(deployer);

timestamp::update_global_time_for_test(42);
let u = signer::address_of(user);
register_pizza_lover(deployer, u);

let assigned = get_claimed_amount(u);
assert!(assigned >= 100 && assigned <= 500, 10); // OCTAS range
assert!(assigned < 100_000_000, 11); // strictly < 1 APT

coin::register<aptos_coin::AptosCoin>(deployer);
let owner_coins = coin::mint<aptos_coin::AptosCoin>(assigned, &mint_cap);
coin::deposit<aptos_coin::AptosCoin>(@pizza_drop, owner_coins);
fund_pizza_drop(deployer, assigned);

claim_pizza_slice(user);
let bal = coin::balance<aptos_coin::AptosCoin>(u);
assert!(bal == assigned, 12); // received OCTAS 1:1

coin::destroy_burn_cap(burn_cap);
coin::destroy_mint_cap(mint_cap);

```

Recommended Mitigation

```

+ // Choose one of the two: (A) pay real APT; (B) clarify that amounts are
+ // OCTAS.

+ // (A) Pay 100-500 APT (scale by 1e8):
+ const OCTAS_PER_APT: u64 = 100_000_000;
+ const MIN_APT: u64 = 100;
+ const MAX_APT: u64 = 500;
+ // Use a secure RNG source (see H2A) returning a u64 'r'
+ let apt_amount: u64 = MIN_APT + (r % (MAX_APT - MIN_APT + 1));
+ let amount_octas: u64 = apt_amount * OCTAS_PER_APT; // 100..500 APT in
+ // octas
- table::add(&mut state.users_claimed_amount, user_addr, random_amount);
+ table::add(&mut state.users_claimed_amount, user_addr, amount_octas);

+ // (B) If the intent is 100-500 OCTAS, explicitly document it and update
+ // UI/spec:
+ // - Rename fields/messages to "octas"
+ // - Show amounts in APT with proper decimal conversion (divide by 1e8) to
+ // avoid confusion

```

Apply an explicit unit conversion when calculating and transferring rewards by multiplying the intended APT amount



by the appropriate octa conversion factor (e.g., `OCTAS_PER_APT = 100_000_000`).

Security Consideration

Important: Correcting the unit mismatch to distribute actual APT values should be performed **in conjunction with fixing the randomness mechanism** described in [\[M-01\]](#). Without addressing predictable timestamp-based randomness, increasing payouts to the intended APT scale would significantly amplify the financial impact of randomness manipulation and enable economically severe exploitation.

[H-02] Unrestricted Registration Enables Sybil Pool Drain

Description

The `get_random_slice` function is exposed as a public entry function with no access control. Participant registration is implicitly inferred from the presence of an entry in `users_claimed_amount`, which is written inside `get_random_slice` without any Owner authorization checks.

As a result, any account can self-register by calling `get_random_slice` and subsequently call `claim_pizza_slice`, fully bypassing the intended "owner-only registration" requirement.

This enables a Sybil attack pattern, where an attacker creates or uses a large number of fresh addresses to repeatedly self-register and claim funds until the pool is exhausted.

[Code below] 



```

#[randomness]
@> entry fun get_random_slice(user_addr: address) acquires ModuleData, State
{ // // externally callable; no owner/ACL → self-registration
  let state = borrow_global_mut<State>(get_resource_address());
  let time = timestamp::now_microseconds();
  let random_val = time % 401;
  let random_amount = 100 + random_val;
  @> table::add(&mut state.users_claimed_amount, user_addr, random_amount);
  // writes "registered" key; since this is public entry, anyone can add for
  // any address (bypasses owner-only registration)
}

public entry fun claim_pizza_slice(user: &signer) acquires ModuleData,
State {
  let user_addr = signer::address_of(user);
  let state = borrow_global_mut<State>(get_resource_address());
  @> assert!(table::contains(&state.users_claimed_amount, user_addr),
E_NOT_REGISTERED); // eligibility = presence in the table that attackers can
// populate via the public entry above
  assert!(!table::contains(&state.claimed_users, user_addr),
E_ALREADY_CLAIMED);
  ...
}

```

Impact

- **Access Control Bypass:** The Owner's exclusive authority to register participants is completely circumvented.
- **Pool Drain:** An attacker can mass-register arbitrary addresses and claim from each, draining the PizzaDrop pool via a Sybil attack.
- **State Bloat:** Unbounded writes to `users_claimed_amount` allow unnecessary global storage growth, increasing operational and maintenance costs.

Likelihood

- `get_random_slice` is a public entry function and can be invoked by any signer once the contract is deployed.
- Registration gating relies solely on the existence of a key in `users_claimed_amount`, which is written without any access control.
- There are no rate limits, identity checks, or economic barriers; creating multiple fresh accounts is inexpensive.



Proof of Concept

A malicious actor creates or controls multiple addresses and calls `get_random_slice` from each address, thereby self-registering. Each address can then successfully invoke `claim_pizza_slice` once, allowing repeated claims until the pool balance is depleted.

```
// 1) Attacker self-registers by calling get_random_slice(attacker_addr).
// 2) Once the pool is funded >= assigned amount, attacker calls
claim_pizza_slice and receives coins.

#[test(deployer = @pizza_drop, attacker = @0xAAA, framework = @0x1)]
fun poc_unrestricted_registration(
  deployer: &signer, attacker: &signer, framework: &signer
) acquires State, ModuleData {
  use aptos_framework::{timestamp, aptos_coin, coin, account};

  timestamp::set_time_has_started_for_testing(framework);
  let (burn_cap, mint_cap) = aptos_coin::initialize_for_test(framework);
  account::create_account_for_test(@pizza_drop);
  account::create_account_for_test(signer::address_of(attacker));
  init_module(deployer);

  // Attacker self-registers (no owner):
  let a = signer::address_of(attacker);
  get_random_slice(a); // <-- unrestricted 'entry'
  assert!(is_registered(a), 1);
  let amt = get_claimed_amount(a);

  // Owner funds at least 'amt' and attacker claims:
  coin::register<aptos_coin::AptosCoin>(deployer);
  let c = coin::mint<aptos_coin::AptosCoin>(amt, &mint_cap);
  coin::deposit<aptos_coin::AptosCoin>(@pizza_drop, c);
  fund_pizza_drop(deployer, amt);
  claim_pizza_slice(attacker); // <-- succeeds
  assert!(has_claimed_slice(a), 2);

  coin::destroy_burn_cap(burn_cap);
  coin::destroy_mint_cap(mint_cap);
}
```

Recommended Mitigation

- Restrict participant registration to an Owner-only function.
- Separate registration logic from reward calculation and claiming.



- Ensure that entry functions performing state mutations enforce explicit access control using authenticated signer checks.
- Optionally, introduce additional safeguards such as explicit registration mappings or claim eligibility flags managed exclusively by the Owner.

```
- #[randomness]
- entry fun get_random_slice(user_addr: address) acquires ModuleData, State {
+ #[randomness]
+ fun get_random_slice_internal(user_addr: address) acquires ModuleData, State
{
    let state = borrow_global_mut<State>(get_resource_address());
    let time = timestamp::now_microseconds();
    let random_val = time % 401;
    let random_amount = 100 + random_val;
    table::add(&mut state.users_claimed_amount, user_addr, random_amount);
}

- public entry fun register_pizza_lover(owner: &signer, user: address) acquires
ModuleData, State {
+ public entry fun register_pizza_lover(owner: &signer, user: address) acquires
ModuleData, State {
    let state = borrow_global_mut<State>(get_resource_address());
    assert!(signer::address_of(owner) == state.owner, E_NOT_OWNER);
-    get_random_slice(user);
+    // Only the owner can trigger registration:
+    get_random_slice_internal(user);
    event::emit(PizzaLoverRegistered { user });
}
```

Medium

[M-01] Predictable Timestamp-Based Randomness Enables Payout Gaming

Description

The slice amount is computed as:

```
100 + (timestamp::now_microseconds() % 401)
```



This value is fully deterministic and derived solely from the current timestamp. As a result, the “randomness” can be predicted and influenced by controlling when the function is invoked.

An attacker can repeatedly time or spam calls to bias the modulo operation toward higher values (e.g., near the upper bound of 500). This allows systematic payout optimization rather than unbiased random distribution.

```
#[randomness]
entry fun get_random_slice(user_addr: address) acquires ModuleData, State {
  let state = borrow_global_mut<State>(get_resource_address());
  @> let time = timestamp::now_microseconds(); // fully deterministic from the
current time
  @> let random_val = time % 401; // not so random
  @> let random_amount = 100 + random_val; // deterministic: 100 + (now %
401)
  table::add(&mut state.users_claimed_amount, user_addr, random_amount);
}
```

Impact

- **Fairness Violation:** Participants who can time calls gain a consistent advantage over honest users, undermining the intended random distribution.
- **Accelerated Pool Depletion:** When combined with mass address creation (*Sybil behavior*), higher average payouts significantly speed up exhaustion of the distribution pool.
- **Forward Escalation Risk:** If reward units are later corrected to distribute actual APT amounts (see [\[H-01\]](#)), this predictability escalates into a financially severe issue.

Likelihood

- The output depends exclusively on the current timestamp, which is observable and partially controllable by the caller.
- There are no additional entropy sources, commit-reveal schemes, or rate limits to prevent timing-based manipulation.



- Even if participant registration were restricted to the Owner, the Owner (or colluding parties) could still bias payouts by timing registrations.

Proof of Concept

By repeatedly invoking the registration or claim path at carefully chosen timestamps, an attacker can bias the modulo result toward higher slice values and consistently receive near-maximum payouts.

```
#[test(deployer = @pizza_drop, user1 = @0x111, framework = @0x1)]
fun poc_timestamp_randomness_is_predictable(
  deployer: &signer,
  user1: &signer,
  framework: &signer
) acquires State, ModuleData {
  use aptos_framework::timestamp;
  use aptos_framework::aptos_coin;
  use aptos_framework::account;
  use aptos_framework::coin;

  // Enable test time and init coin testing caps
  timestamp::set_time_has_started_for_testing(framework);
  let (burn_cap, mint_cap) =
    aptos_coin::initialize_for_test(framework);

  // Accounts + module init
  account::create_account_for_test(@pizza_drop);
  account::create_account_for_test(signer::address_of(user1));
  init_module(deployer);

  // Pick a time where (t % 401) == 400 → expected amount = 500
  timestamp::update_global_time_for_test(4_950_745); // 401*12_345 +
400

  let t = timestamp::now_microseconds();

  let u = signer::address_of(user1);
  register_pizza_lover(deployer, u);
  let amt = get_claimed_amount(u);

  // Determinism checks
  assert!(amt == 100 + (t % 401), 1);
  assert!(amt == 500, 2);

  // Clean up capabilities
  coin::destroy_burn_cap(burn_cap);
  coin::destroy_mint_cap(mint_cap);
}
```



Recommended Mitigation

- Avoid timestamp-based randomness for value-critical decisions.
- Introduce a commit-reveal scheme or use a verifiable randomness source suitable for Aptos.
- Separate randomness generation from user-triggered entry points to reduce caller influence.
- Apply rate limiting or batching mechanisms to mitigate timing and spamming strategies.

```
- // DO NOT derive "randomness" from time.
- let time = timestamp::now_microseconds();
- let random_val = time % 401;
- let random_amount = 100 + random_val;

+ // Use a proper on-chain randomness source or a commit-reveal scheme.
+ // Examples of safer approaches:
+ // 1) Use the chain's randomness API (e.g., a randomness capability
+ //    injected via #[randomness])
+ //    and derive an unbiased u64, then map to [100, 500] carefully.
+ // 2) Use a commit-reveal flow (user commits a salt; reveal happens later
+ //    using a block value).
+ // IMPORTANT: Avoid time-based entropy. Consider anti-Sybil/rate-limiting
+ // if needed.
+ let r /*: u64 from a secure randomness source */;
+ let random_amount = 100 + (r % 401); // or use a uniform mapping to avoid
+ // modulo bias if required
```