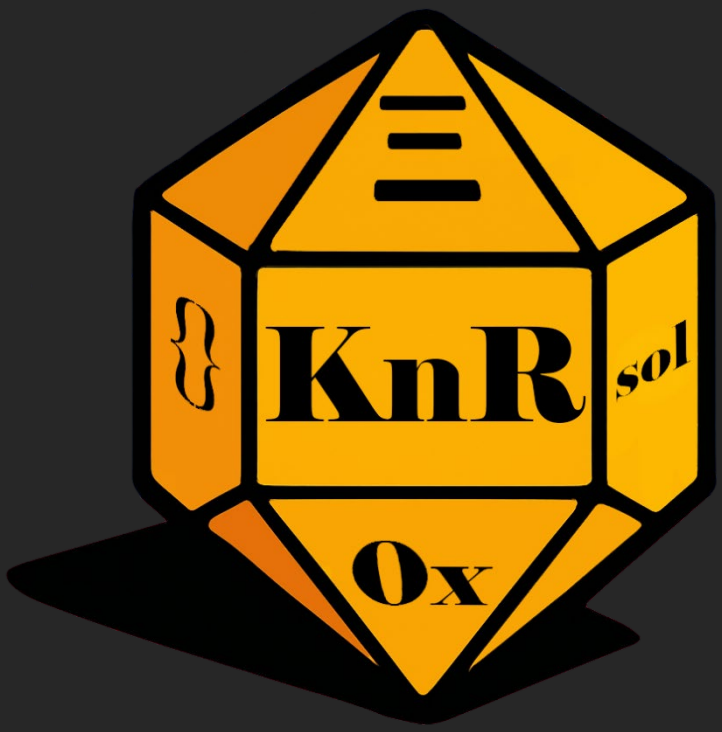

Kode-n-Rolla

One Shot: Reloaded Security Review

Sep 11th 2025 - Sep 18th 2025

Table of Contents

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*:
    - Funds are directly or nearly directly at risk

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

Protocol Summary

RapBattle is an on-chain game protocol deployed on the Aptos blockchain that combines NFT ownership, staking mechanics, and head-to-head battles using a custom fungible token (CRED).

The protocol allows players to mint "*Rapper*" NFTs implemented as Aptos Token V2 objects. Each Rapper has associated logical statistics stored in a resource table, which influence staking rewards and battle outcomes. These NFTs can be staked into the protocol to reduce negative traits ("*vices*") over time and earn CRED rewards based on full days staked.

Battles are initiated by two players matching equal CRED wagers. The protocol computes the battle outcome based on the Rapper statistics and a randomness component, after which the winner receives the combined prize pool and the winning Rapper's state is updated to record the victory.

CRED is implemented as a `*Coin<CRED>*` with *mint* and *burn* capabilities controlled by the module *owner*. The token is used both as a staking reward and as the wagering asset for battles.

From a security perspective, the protocol relies on correct enforcement of access control over minting and custody, accurate state tracking for NFT ownership and statistics,

safe handling of staked assets, and the integrity of the randomness mechanism used in battle resolution. The protocol also introduces centralization assumptions around the module owner, who retains privileged control over token issuance and certain NFT lifecycle operations.

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 "One Shot: Reloaded" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

```
└── sources
|   └── cred_token.move
|   └── one_shot.move
|   └── streets.move
|   └── rap_battle.move
```

Roles

- **Module Owner *(@battle_addr)***

The Module Owner is a privileged account responsible for publishing the protocol modules and controlling critical

administrative capabilities. This role owns the mint and burn capabilities for the CRED token and is exclusively authorized to mint new Rapper NFTs via the *mint_rapper* function.

Due to these privileges, the Module Owner retains full control over the issuance of both fungible (CRED) and non-fungible (Rapper) assets, introducing inherent centralization and trust assumptions.

- **Player / NFT Holder**

A Player is any account that owns or interacts with Rapper NFTs and participates in the protocol's staking and battle mechanics. Players may stake their Rapper NFTs into the protocol to modify internal statistics and earn CRED rewards, and may use CRED tokens to place wagers in battles.

Players are unprivileged accounts and are subject to the access control and state constraints enforced by the protocol.

- **Defender / Challenger**

The Defender and Challenger are contextual roles assumed by Players during a battle. The Defender initiates a battle by committing a CRED wager, while the Challenger matches the wager to trigger battle resolution.

Upon completion of a battle, the combined CRED wager is awarded to the winner, and the winning Rapper's state is updated to record the outcome.

Executive Summary

This security review assessed the RapBattle protocol deployed on the Aptos blockchain, with a focus on NFT custody, staking and battle logic, state consistency, and access control assumptions.

The review identified two high-severity issues and one low-severity issue. The most critical finding results in

permanent asset lock after a completed battle. Due to incorrect state transitions, Rapper NFTs involved in a battle can become irreversibly locked within the protocol, preventing the legitimate owner from reclaiming or transferring the asset.

A second high-severity issue stems from a specification mismatch in the protocol's internal ownership model. After a battle, the winner's address is recorded as the owner of both Rapper NFTs in the internal *RapperStats* registry. As a consequence, the losing player permanently loses ownership of their Rapper at the protocol state level, despite no corresponding on-chain object transfer being executed. This creates an unrecoverable divergence between logical ownership and actual object custody.

Additionally, a low-severity logic flaw was identified in the staking mechanism, where stake records are not properly bound to specific NFT identifiers. While not immediately exploitable for asset theft, this weakens state integrity assumptions and increases the risk of future inconsistencies as the protocol evolves.

Collectively, the identified issues highlight critical flaws in asset custody handling and internal ownership accounting. Without corrective changes, the protocol risks permanent loss of user NFTs and inconsistent state that undermines the reliability and safety of core gameplay mechanics.

Issue found

Severity	Issue
High	Permanent Asset Lock After Battle Due to Missing NFT Transfer
High	Specification Mismatch: Winner Is Assigned Ownership of Both Rapper NFTs

Low	Missing Token Binding in StakeInfo Allows NFT Substitution on Unstake

Findings

High

[H-01] Permanent Asset Lock After Battle Due to Missing NFT Transfer

Description

In the *rap_battle* module, when a Defender or Challenger participates in a battle, their Rapper NFT object is transferred into custody of *@battle_addr*:

```
// rap_battle.move
    one_shot::transfer_record_only(token_id, player_addr, @battle_addr);
@> object::transfer(player, rapper_token, @battle_addr);
```

At the end of the battle, the contract updates the internal ownership state by calling *transfer_record_only* on the stats registry. However, the protocol never performs a corresponding *object::transfer* to return the physical NFT object to the winning player.

As a result, the winner is recorded as the owner in the internal *RapperStats* registry, but the actual NFT object remains permanently held under *@battle_addr*. The winner cannot regain custody or transfer the NFT, despite winning the battle.

```
// rap_battle.move
if (winner == defender_addr) {
    coin::deposit(defender_addr, pool);
    one_shot::increment_wins(arena.defender_token_id);

@>     one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
defender_addr);
```

```
@>     one_shot::transfer_record_only(chall_token_id, @battle_addr,
defender_addr);

} else {
    coin::deposit(chall_addr, pool);
    one_shot::increment_wins(chall_token_id);

@>     one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
chall_addr);
@>     one_shot::transfer_record_only(chall_token_id, @battle_addr,
chall_addr);
}
```

Impact

- **Permanent Asset Lock**: Rapper NFTs become irretrievably locked under protocol custody.
- **Broken Gameplay & Incentives**: Players have no incentive to participate in battles if winning results in loss of NFT control.
- **Asset Loss**: If Rapper NFTs have market or in-game value, users permanently lose their assets despite winning.

Likelihood

- The custody transfer into @battle_addr occurs for every battle.
- No code path exists to transfer the NFT object back to the winning player.
- The issue affects all completed battles consistently.

Proof of Concept

1. Alice and Bob each mint a Rapper NFT.
   → *one_shot::test_mint_to_player_and_return_object*
2. Alice initiates a battle with a wager.
   → *rap_battle::go_on_stage_or_battle*
3. Bob matches the wager, completing the battle.
   → *rap_battle::go_on_stage_or_battle*
4. After the battle:
   - Both Rapper NFTs remain in custody of *@battle_addr*.

- The recorded winner cannot transfer or recover their NFT.
- *@battle_addr* is able to transfer the NFTs, demonstrating retained custody.

```
// one_shot.move
#[test_only]
public fun test_mint_to_player_and_return_object(
    module_owner: &signer,
    player: &signer
): (object::Object<aptos_token_v2::token::Token>, address) acquires
RapperStats {
    // ... helper to mint a Rapper directly to a player and return the object
}

// rap_battle.move
#[test_only]
public fun test_init(arena_owner: &signer) {
    init_module(arena_owner)
}

// asset_lock_stage_poc.move
module battle_addr::asset_lock_stage_poc {
    use std::signer;
    use aptos_framework::object::{Self as object, Object};
    use aptos_token_v2::token::Token;
    use battle_addr::one_shot;
    use battle_addr::rap_battle;

    #[test(battle_addr = @0x42, alice = @0xa11ce)]
    public fun test_asset_lock_after_on_stage(battle_addr: &signer, alice:
&signer) {
        rap_battle::test_init(battle_addr);
        let (alice_obj, alice_token_id) =
one_shot::test_mint_to_player_and_return_object(battle_addr, alice);
        rap_battle::go_on_stage_or_battle(alice, alice_obj, 0);

        // Only @battle_addr can transfer the Rapper back — proves custody
lock
        let obj_from_custody: Object<Token> =
object::address_to_object<Token>(alice_token_id);
        object::transfer(battle_addr, obj_from_custody,
signer::address_of(alice));
    }

    #[test(battle_addr = @0x42, alice = @0xa11ce)]
    #[expected_failure]
    public fun test_player_cannot_transfer_while_in_custody(battle_addr:
&signer, alice: &signer) {
        rap_battle::test_init(battle_addr);
        let (alice_obj, alice_token_id) =
```

```
one_shot::test_mint_to_player_and_return_object(battle_addr, alice);
        rap_battle::go_on_stage_or_battle(alice, alice_obj, 0);

        // Player tries to transfer while not the owner → aborts
        let try_take: Object<Token> =
object::address_to_object<Token>(alice_token_id);
        object::transfer(alice, try_take, signer::address_of(alice));
    }
}
```

Recommended Mitigation

After determining the battle winner, explicitly transfer custody of the winning Rapper NFT back to the winner's address using object::transfer. Ensure that physical object custody and logical ownership recorded in RapperStats remain consistent after battle resolution.

```
if (winner == defender_addr) {
    coin::deposit(defender_addr, pool);
    one_shot::increment_wins(arena.defender_token_id);
    one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
defender_addr);
    one_shot::transfer_record_only(chall_token_id, @battle_addr,
defender_addr);

+   let def_obj: Object<Token> =
object::address_to_object<Token>(arena.defender_token_id);
+   object::transfer(module_owner, def_obj, defender_addr);

+   let chall_obj: Object<Token> =
object::address_to_object<Token>(chall_token_id);
+   object::transfer(module_owner, chall_obj, defender_addr);

} else {
    coin::deposit(chall_addr, pool);
    one_shot::increment_wins(chall_token_id);
    one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
chall_addr);
    one_shot::transfer_record_only(chall_token_id, @battle_addr, chall_addr);

+   let def_obj: Object<Token> =
object::address_to_object<Token>(arena.defender_token_id);
+   object::transfer(module_owner, def_obj, chall_addr);

+   let chall_obj: Object<Token> =
object::address_to_object<Token>(chall_token_id);
```

```
+   object::transfer(module_owner, chall_obj, chall_addr);
}
```

[H-02] Specification Mismatch: Winner Is Assigned Ownership of Both Rapper NFTs

Description

In *rap_battle::go_on_stage_or_battle*, the protocol updates the internal RapperStats registry by assigning the winner's address as the owner of ***both*** Rapper NFTs involved in the battle.

```
if (winner == defender_addr) {
    // Registry update: assign the defender (winner) as the owner of the
DEFENDER's own token
@>  one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
defender_addr);
    // CRITICAL: also assign the defender (winner) as the owner of the
CHALLENGER's token
@>  one_shot::transfer_record_only(chall_token_id, @battle_addr,
defender_addr);
} else {
    // Registry update: assign the challenger (winner) as the owner of the
DEFENDER's token
@>  one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
chall_addr);
    // Registry update: assign the challenger (winner) as the owner of the
CHALLENGER's own token
@>  one_shot::transfer_record_only(chall_token_id, @battle_addr, chall_addr);
}
```

As a result, the loser permanently loses ownership of their Rapper NFT at the protocol state level, despite no corresponding on-chain object transfer being executed. The physical NFT object remains unchanged, but the internal registry enforces a "winner takes all NFTs" rule.

This behavior creates a clear mismatch between the expected battle outcome and the implemented ownership logic.

Impact

- **Specification Violation**: The implemented behavior diverges from the likely intended design of "winner keeps their NFT, loser loses only the wager."
- **Silent Asset Confiscation**: Losing players permanently lose ownership of their Rapper NFTs in protocol state without explicit consent or disclosure.
- **Potential Asset Theft**: An attacker can repeatedly challenge new players and, upon winning a single battle, effectively confiscate their Rapper NFTs through protocol logic.
- **State Inconsistency**: Logical ownership recorded in *RapperStats* diverges from physical object custody, further compounding custody and recovery risks.

Severity Note:

If this behavior is intended game design, the issue represents a user-hostile and highly centralized mechanic. If unintended, it constitutes a high-severity asset ownership vulnerability enabling unauthorized NFT confiscation.

Likelihood

- Ownership reassignment is deterministic and occurs during every battle resolution.
- No conditional checks or opt-in logic exists to preserve the loser's NFT ownership.
- The issue affects all battles consistently.

Proof of Concept

To demonstrate the issue, test-only helper functions were introduced to isolate the battle flow:

```
#[test_only]
public fun test_get_owner(token_id: address): address acquires RapperStats {
    use aptos_std::table;
    let stats = &borrow_global<RapperStats>(@battle_addr).stats;
```

- Test-only mint helpers in *one_shot.move* (above 👆)
- A test-only battle execution path in *rap_battle.move* without coin or timestamp dependencies

```
#[test_only]
    public fun test_go_on_stage_or_battle_no_coin(
        player: &signer,
        rapper_token: Object<Token>,
        bet_amount: u64
    ) acquires BattleArena {
        let player_addr = signer::address_of(player);
        let arena = borrow_global_mut<BattleArena>(@battle_addr);

        if (arena.defender == @0x0) {
            assert!(arena.defender_bet == 0, E_BATTLE_ARENA_OCCUPIED);
            arena.defender = player_addr;
            arena.defender_bet = bet_amount;

            let token_id = object::object_address(&rapper_token);
            arena.defender_token_id = token_id;

            // no coin ops in tests
            one_shot::transfer_record_only(token_id, player_addr,
@battle_addr);
            object::transfer(player, rapper_token, @battle_addr);

        } else {
            assert!(arena.defender_bet == bet_amount, E_BETS_DO_NOT_MATCH);
            let defender_addr = arena.defender;
            let chall_addr = player_addr;

            let chall_token_id = object::object_address(&rapper_token);
            one_shot::transfer_record_only(chall_token_id, chall_addr,
@battle_addr);
            object::transfer(player, rapper_token, @battle_addr);

            // no coin ops in tests

            let defender_skill = one_shot::skill_of(arena.defender_token_id);
            let challenger_skill = one_shot::skill_of(chall_token_id);
            let total_skill = defender_skill + challenger_skill;
            let rnd = Self::test_now_seconds() % (if (total_skill == 0) { 1 }
else { total_skill });
            let winner = if (rnd < defender_skill) { defender_addr } else {
chall_addr };

            // event::emit(Battle { challenger: chall_addr,
challenger_token_id: chall_token_id, winner });

            if (winner == defender_addr) {
                one shot::increment wins(arena.defender token id);
```

```
                one_shot::transfer_record_only(arena.defender_token_id,
@battle_addr, defender_addr);
                one_shot::transfer_record_only(chall_token_id, @battle_addr,
defender_addr);
            } else {
                one_shot::increment_wins(chall_token_id);
                one_shot::transfer_record_only(arena.defender_token_id,
@battle_addr, chall_addr);
                one_shot::transfer_record_only(chall_token_id, @battle_addr,
chall_addr);
            };

            // reset arena
            arena.defender = @0x0;
            arena.defender_bet = 0;
            arena.defender_token_id = @0x0;
        }
```

A unit test (*tests/winner_takes_both_poc.move*) confirms that after a single battle:

- The winner is recorded as the owner of both Rapper NFTs in RapperStats
- The loser permanently loses protocol-level ownership of their NFT

```
module battle_addr::winner_takes_both_poc {
    use battle_addr::one_shot;
    use battle_addr::rap_battle;
    use aptos_framework::object::{Self as object, Object};
    use aptos_token_v2::token::Token;

    #[test(battle_addr = @0x42, alice = @0xa11ce, bob = @0xb0b)]
    public fun test_registry_assigns_both_to_winner(
        battle_addr: &signer,
        alice: &signer,
        bob: &signer,
    ) {
        // Init arena
        rap_battle::test_init(battle_addr);

        // Mint Rapper NFTs
        let (alice_obj, alice_id) =
one_shot::test_mint_to_player_and_return_object(battle_addr, alice);
        let (bob_obj,   bob_id)   =
one_shot::test_mint_to_player_and_return_object(battle_addr, bob);
```

```
        // Run test-only battle (bet=0, deterministic RNG)
        rap_battle::test_go_on_stage_or_battle_no_coin(alice, alice_obj, 0);
        rap_battle::test_go_on_stage_or_battle_no_coin(bob, bob_obj, 0);

        // Both NFT owners in registry are reassigned to winner
        let alice_owner = one_shot::test_get_owner(alice_id);
        let bob_owner   = one_shot::test_get_owner(bob_id);
        assert!(alice_owner == bob_owner, 1001);
    }
}
```

Recommended Mitigation

Update rap_battle::go_on_stage_or_battle to ensure that:

- Only the winning Rapper NFT has its statistics updated
- Ownership records for the losing Rapper NFT remain unchanged
- Logical ownership tracking remains aligned with actual NFT custody semantics

```
if (winner == defender_addr) {
    coin::deposit(defender_addr, pool);
    one_shot::increment_wins(arena.defender_token_id);
-   one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
defender_addr);
-   one_shot::transfer_record_only(chall_token_id, @battle_addr,
defender_addr);
+   one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
defender_addr);
+   // keep challenger's NFT ownership unchanged
} else {
    coin::deposit(chall_addr, pool);
    one_shot::increment_wins(chall_token_id);
-   one_shot::transfer_record_only(arena.defender_token_id, @battle_addr,
chall_addr);
-   one_shot::transfer_record_only(chall_token_id, @battle_addr, chall_addr);
+   one_shot::transfer_record_only(chall_token_id, @battle_addr, chall_addr);
+   // keep defender's NFT ownership unchanged
}
```

Low

[L-01] Missing Token Binding in *StakeInfo* Allows NFT Substitution on *Unstake*

Description

The *StakeInfo* structure does not store the *token_id* (or equivalent unique identifier) of the staked Rapper NFT. During *streets::unstake*, the function trusts the caller-supplied *rapper_token* object and returns that object from custody, without verifying that it corresponds to the originally staked NFT.

```
public entry fun stake(staker: &signer, rapper_token: Object<Token>) {
    let staker_addr = signer::address_of(staker);
    let token_id = object::object_address(&rapper_token);

    move_to(staker, StakeInfo {
        start_time_seconds: timestamp::now_seconds(),
        owner: staker_addr,
    });

    one_shot::transfer_record_only(token_id, staker_addr, @battle_addr);
    object::transfer(staker, rapper_token, @battle_addr);
}

public entry fun unstake(staker: &signer, module_owner: &signer,
rapper_token: Object<Token>) acquires StakeInfo {
    let staker_addr = signer::address_of(staker);
    let token_id = object::object_address(&rapper_token);

    assert!(exists<StakeInfo>(staker_addr), E_TOKEN_NOT_STAKED);
    let stake_info = borrow_global<StakeInfo>(staker_addr);
    assert!(stake_info.owner == staker_addr, E_NOT_OWNER);

    // @CRITICAL: StakeInfo doesn't bind a token_id, so the function
    // uses the caller-provided object to decide what to return.
@>  let StakeInfo { start_time_seconds: _, owner: _ } =
move_from<StakeInfo>(staker_addr);

@>  one_shot::transfer_record_only(token_id, @battle_addr, staker_addr);
// reassigns the supplied token_id
@>  object::transfer(module_owner, rapper_token, staker_addr); // returns the
supplied object
}
```

As a result, *unstaking* is not bound to the specific NFT that was previously staked.

Why This Matters

A user who originally staked Rapper **A** can call *unstake* while supplying the object handle of a different Rapper **B** that is currently held in protocol custody (for example, belonging to another player). The function will return **B** to the caller instead of **A**.

This constitutes a token substitution flaw: the protocol does not guarantee that the asset returned during unstaking is the same asset that was initially staked.

Impact

- **NFT Theft or Mix-Up**: Callers may retrieve a different NFT than the one they staked.
- **State Integrity Violation**: Ownership and staking records can diverge from actual asset movements.
- **Trust Degradation**: Even without malicious intent, users may receive the wrong NFT back, breaking expected protocol behavior.

Practical Consideration:

If the Module Owner's signer is automatically co-signed by backend infrastructure (a common pattern in custody-based flows), this issue can become practically exploitable on-chain rather than purely theoretical.

Likelihood

- The logic flaw is unconditional; no checks bind a stake record to a specific token identifier.
- The issue is exploitable whenever two or more Rapper NFTs are simultaneously held in custody.

Proof of Concept

A unit test demonstrates the issue by staking one Rapper NFT and then calling *unstake* with a different Rapper object currently held in custody. The function returns the caller-

supplied NFT rather than the originally staked one, confirming the absence of token binding.

Test-only helper functions were introduced to stub unrelated framework dependencies (timestamp and coin handling) while preserving the exact staking and ownership logic. These helpers do not alter production behavior and are used solely to isolate and demonstrate the flaw.

Test-only helpers (added to sources)

In *one_shot.move* – read logical owner from the registry

```
#[test_only]
public fun test_get_owner(token_id: address): address acquires RapperStats {
    use aptos_std::table;
    let stats = &borrow_global<RapperStats>(@battle_addr).stats;
    let s = table::borrow(stats, token_id);
    s.owner
}
```

In *streets.move* – *stake/unstake* clones without timestamp/coin, but with the same ownership logic

```
#[test_only]
public fun test_stake(staker: &signer, rapper_token: Object<Token>) {
    let staker_addr = signer::address_of(staker);
    let token_id = aptos_framework::object::object_address(&rapper_token);

    // same as stake, but fixed time (no timestamp) and no CRED logic
    move_to(staker, StakeInfo { start_time_seconds: 0, owner: staker_addr });
    battle_addr::one_shot::transfer_record_only(token_id, staker_addr,
@battle_addr);
    aptos_framework::object::transfer(staker, rapper_token, @battle_addr);
}

#[test_only]
public fun test_unstake_vulnerable(
    staker: &signer,
    module_owner: &signer,
    rapper_token: Object<Token>
) acquires StakeInfo {
    let staker_addr = signer::address_of(staker);
    let token_id = aptos_framework::object::object_address(&rapper_token);
```

```
    assert!(exists<StakeInfo>(staker_addr), E_TOKEN_NOT_STAKED);
    let si = borrow_global<StakeInfo>(staker_addr);
    assert!(si.owner == staker_addr, E_NOT_OWNER);

    // identical flaw: uses the caller-provided object/address
    let StakeInfo { start_time_seconds: _, owner: _ } =
move_from<StakeInfo>(staker_addr);
    battle_addr::one_shot::transfer_record_only(token_id, @battle_addr,
staker_addr);
    aptos_framework::object::transfer(module_owner, rapper_token,
staker_addr);
}
```

PoC test in *tests/stakeinfo_poc.move*

```
module battle_addr::stakeinfo_poc {
    use std::signer;
    use aptos_framework::object::{Self as object, Object};
    use aptos_token_v2::token::Token;

    use battle_addr::one_shot;
    use battle_addr::streets;

    /// PoC: Unstake returns whatever token handle the caller passes.
    #[test(battle_addr = @0x42, alice = @0xa11ce, bob = @0xb0b)]
    public fun test_unstake_with_wrong_token_returns_other(
        battle_addr: &signer,
        alice: &signer,
        bob: &signer,
    ) {
        // Mint Rapper A for Alice, B for Bob
        let (alice_obj, alice_id) =
one_shot::test_mint_to_player_and_return_object(battle_addr, alice);
        let (bob_obj,   bob_id)   =
one_shot::test_mint_to_player_and_return_object(battle_addr, bob);

        // Both stake into custody (test helper: no timestamp/coin)
        streets::test_stake(alice, alice_obj); // A → custody
        streets::test_stake(bob,   bob_obj);   // B → custody

        // Alice crafts a handle for Bob's token (which is in custody)
        let fake_b: Object<Token> = object::address_to_object<Token>(bob_id);

        // Exploit: Alice unstakes while passing Bob's token handle
        streets::test_unstake_vulnerable(alice, battle_addr, fake_b);

        // Registry now shows Bob's Rapper belongs to Alice
        let owner_of_B_after = one_shot::test_get_owner(bob_id);
```

```
        assert!(owner_of_B_after == signer::address_of(alice), 1001);
    }
}
```

Recommended Mitigation

Bind each stake record to a specific token identifier at stake time. During *unstake*, ignore caller-supplied object handles and instead return the NFT that matches the recorded token identifier associated with the stake.

```
- struct StakeInfo has key, store {
-     start_time_seconds: u64,
-     owner: address,
- }
+ struct StakeInfo has key, store {
+     token_id: address,            // bind to the exact token
+     start_time_seconds: u64,
+     owner: address,
+ }

 public entry fun stake(staker: &signer, rapper_token: Object<Token>) {
     let staker_addr = signer::address_of(staker);
     let token_id = object::object_address(&rapper_token);

-    move_to(staker, StakeInfo { start_time_seconds:
timestamp::now_seconds(), owner: staker_addr });
+    move_to(staker, StakeInfo {
+        token_id,
+        start_time_seconds: timestamp::now_seconds(),
+        owner: staker_addr,
+    });

     one_shot::transfer_record_only(token_id, staker_addr, @battle_addr);
     object::transfer(staker, rapper_token, @battle_addr);
 }

-public entry fun unstake(staker: &signer, module_owner: &signer,
rapper_token: Object<Token>) acquires StakeInfo {
+public entry fun unstake(staker: &signer, module_owner: &signer, _ignored:
Object<Token>) acquires StakeInfo {
     let staker_addr = signer::address_of(staker);
-    let token_id = object::object_address(&rapper_token);
+    let info = borrow_global<StakeInfo>(staker_addr);
+    let token_id = info.token_id;

     assert!(exists<StakeInfo>(staker_addr), E_TOKEN_NOT_STAKED);
```

```
-     let stake_info = borrow_global<StakeInfo>(staker_addr);
-     assert!(stake_info.owner == staker_addr, E_NOT_OWNER);
+     assert!(info.owner == staker_addr, E_NOT_OWNER);

      // ... rewards & stats ...

-     let StakeInfo { start_time_seconds: _, owner: _ } =
move_from<StakeInfo>(staker_addr);
+     let StakeInfo { start_time_seconds: _, owner: _, token_id: _ } =
move_from<StakeInfo>(staker_addr);

      one_shot::transfer_record_only(token_id, @battle_addr, staker_addr);
-     object::transfer(module_owner, rapper_token, staker_addr);
+     // Rebuild the exact object by id; do not trust caller's handle
+     let obj =
object::address_to_object<aptos_token_v2::token::Token>(token_id);
+     object::transfer(module_owner, obj, staker_addr);
}
```