Audit Report

Stargaze Infinity Pool Re-audit

v1.0

July 20, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Stargaze Foundation to perform a security audit of Stargaze's Infinity Pool.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/public-awesome/infinity
Commit	`1ee941e876382ef444800b0150911b55ebeebc70`
Scope	All contracts were in scope.

Note that this audit was conducted prior to a significant redesign of the infinity protocol.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Infinity Pool is an AMM protocol enabling the trade of NFT assets using a specified fungible token. The NFT assets' buy and sell prices are determined by the pool's parameters, the bonding curve, and the assets held by the pool.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium-High	The infinity pool codebase integrates with other Stargaze contracts, increasing the overall complexity.
Code readability and clarity	Medium-High	-
Level of documentation	Medium-High	Detailed documentation was provided in the README file.
Test coverage	Medium	`cargo tarpaulin` reports a test coverage of `72.65%` with `1498/2062` lines covered.

Summary of Findings

No	Description	Severity	Status
1	Malicious users can transact NFTs with zero deltas, causing a loss of profit for pool owners	Critical	Acknowledged
2	Swap fees are not applied throughout the codebase	Major	Acknowledged
3	Payment distribution will fail for zero seller amount	Major	Acknowledged
4	Iterating through best-priced pools may fail due to an out-of-gas error	Major	Acknowledged
5	Pool activation is possible before depositing the required assets, resulting in an empty pool	Minor	Acknowledged
6	Trades will fail if the delta decreases the spot price to zero	Minor	Acknowledged
7	Possible incorrect denom emitted during contract instantiation	Minor	Acknowledged
8	Pools can be deactivated through the `SetActivePool` message, leaving them in an inconsistent state	Minor	Acknowledged
9	Inefficient pool type validation	Informational	Acknowledged
10	Unhandled zero-amount withdrawal	Informational	Acknowledged
11	Redundant parameter when depositing NFTs	Informational	Acknowledged
12	Misleading parameter naming in `swap_tokens_for_any_nfts` function	Informational	Acknowledged
13	Developer address is not emitted during contract instantiation	Informational	Acknowledged
14	Inconsistent attribute name for pool identifiers	Informational	Acknowledged

Detailed Findings

1. Malicious users can transact NFTs with zero deltas, causing a loss of profit for pool owners

Severity: Critical

In `contracts/infinity-swap/src/swap_processor.rs:560-571` and `718-730`, the `break` statement will be entered if the swap fails. This is problematic because reused pools will not be stored in `self.pools_to_save`, which is responsible for saving the pool into storage.

As a result, a malicious user could potentially exploit this by intentionally causing the second swap to fail when using the `SwapNftsForTokens` and `SwapTokensForAnyNfts` messages. This effectively allows them to buy or sell NFTs without affecting the deltas, resulting in a potential loss of profit for pool owners.

Please see the `unsaved_pools` test case to reproduce the issue.

Recommendation

We recommend saving reused pools in storage even when the swap fails to correctly update the pool's spot price.

Status: Acknowledged

2. Swap fees are not applied throughout the codebase

Severity: Major

In `contracts/infinity-swap/src/pool.rs:127-133`, the trade pool expects the caller to provide a swap fee percentage value, which represents the percentage of the swap that will be paid to the pool owner. However, the fee is not collected anywhere in the contract. Consequently, pool owners will not receive their share of swap fees when users trade on their pool.

Recommendation

We recommend applying swap fees on trades.

Status: Acknowledged

3. Payment distribution will fail for zero seller amount

Severity: Major

In `contracts/infinity-swap/src/swap_processor.rs:410-416`, the `transfer_token` function is used to distribute tokens to all intended recipients. If the seller amount becomes zero after deducting network, finder, and royalty fees, the transaction will revert because Cosmos SDK prevents sending zero amounts. As a result, trades will not be executed as intended.

Recommendation

We recommend only distributing the seller amount if it is greater than zero.

Status: Acknowledged

4. Iterating through best-priced pools may fail due to an out-of-gas error

Severity: Major

The protocol enables NFTs to be exchanged for tokens while allowing multiple identical pools to coexist. When buying NFTs, it is possible to specify the maximum amount of tokens to spend in exchange for the lowest NFT price.

To achieve this, it is necessary to iterate through existing pools to find the highest price when selling NFTs or the lowest price when purchasing NFTs. The `load_next_pool` function in `contracts/infinity-swap/src/swap_processor.rs:439-452` serves this purpose, iterating through all pools and sorting them to find the optimal one based on the transaction type.

However, the implementation does not account for the potential issue of excessive computational power consumption when many pools are created over time, either organically or by a malicious actor. In such cases, both operations, `DirectSwapNftsForTokens` and `SwapTokensForAnyNfts`, may consistently return errors related to exceeding the maximum gas limit, leading to a denial of service for the protocol.

We classify this issue as major instead of critical because users can still trade using other entry points.

Recommendation

We recommend limiting the number of pools to fetch depending on the `nfts_to_swap` and `min_expected_token_input` vector length.

Status: Acknowledged

5. Pool activation is possible before depositing the required assets, resulting in an empty pool

Severity: Minor

The project's technical documentation outlines the process of creating a pair for one of the three types of pools. It begins with calling the `CreatePool` message with appropriate parameters, followed by depositing tokens or NFTs via the `DepositTokens` or `DepositNfts` messages. Lastly, the pool is activated by calling the `SetActivePool` message.

However, the `execute_set_active_pool` function in `contracts/infinity-swap/src/execute.rs:544` does not verify that a deposit has been made. Consequently, creating and activating a pool without making any deposit is possible, resulting in an empty pool.

Please see the `test_activate_without_deposit` test case to reproduce this issue.

Recommendation

We recommend implementing validation during pool activation in order to check whether the deposit has already been made.

Status: Acknowledged

6. Trades will fail if the delta decreases the spot price to zero

Severity: Minor

In `contracts/infinity-swap/src/pool.rs:364-377`, linear and exponential bonding curves update the spot price by deducting their delta with respective calculations. In a scenario where the spot price becomes zero, `Tokens` and `Nft` pools will fail to update, as seen in lines `68-72` and `101-105`. Consequently, pools that misconfigured their delta will fail to purchase NFTs from users.

Please see the `error_updating_spot_price` test case to reproduce this issue.

Recommendation

We recommend adding validation to ensure the spot price will not result in zero before purchasing NFTs.

Status: Acknowledged

7. Possible incorrect denom emitted during contract instantiation

Severity: Minor

In `contracts/infinity-swap/src/instantiate.rs:37`, the denom attribute value is emitted as `msg.denom`. This is problematic because the contract uses `NATIVE_DENOM`, which is hardcoded to `ustars`. If the contract instantiator provides another type of denom, the contract will emit it, which may mislead users.

Recommendation

We recommend modifying the denom attribute value to `NATIVE_DENOM`.

Status: Acknowledged

8. Pools can be deactivated through the `SetActivePool` message, leaving them in an inconsistent state

Severity: Minor

The `is_active` parameter, changeable via the `SetActivePool` message or directly through the `set_active` function, is set and used during creation of a pool in `contracts/infinity-swap/src/execute.rs:556` and is negated when removing it in `contracts/infinity-swap/src/helpers.rs:214`.

However, the documentation does not mention that it is possible to deactivate a pool by sending the `SetActivePool` message with a `false` value while it has deposits and is open for swaps. This deactivation prevents any further swaps.

Recommendation

We recommend modifying the `SetActivePool` message not to take a boolean value from user input but rather only use the message for pool activation. This change will not affect the `RemovePool` message, as the `remove_pool` function directly calls `set_active(false)`.

Status: Acknowledged

9. Inefficient pool type validation

Severity: Informational

In `contracts/infinity-swap/src/pool.rs:165-173`, the `deposit_nfts` function performs a validation check to ensure that the pool is of the NFT type. However, during the deposit of NFTs in `contracts/infinity-swap/src/execute.rs:346`, this validation is called after the NFT transfer is performed (see lines `336-344`).

If the validation was performed earlier, such as at the beginning of the `execute_deposit_nfts` function, the transaction could consume less gas before being reverted in case of an incorrect pool type.

Recommendation

We recommend performing the pool type validation at the beginning of the `DepositNfts` call.

Status: Acknowledged

10. Unhandled zero-amount withdrawal

Severity: Informational

Using the `WithdrawTokens` and `WithdrawAllTokens` messages, a user can withdraw tokens from the specified pool. The funds are transferred from the contract to the recipient using the `transfer_token` function defined in `contracts/infinity-swap/src/helpers.rs:335`.

However, these messages do not validate whether the amount is greater than zero. If a user attempts to withdraw zero tokens, an error will be returned by the Cosmos SDK, as `BankMsg::Send` does not support zero amount transfers.

Recommendation

We recommend adding a validation step when withdrawing funds to skip the transfer if the amount is greater than zero.

Status: Acknowledged

11. Redundant parameter when depositing NFTs

Severity: Informational

In `contracts/infinity-swap/src/execute.rs:327-332`, the `execute_deposit_nfts` function requires the caller to provide a `collection` parameter, which is then checked to ensure the collection address matches the pool's collection address. This parameter can be removed, as the pool's collection address can be used directly instead, reducing gas consumption and code complexity.

Recommendation

We recommend removing the collection parameter and using the pool's collection address instead.

Status: Acknowledged

12. Misleading parameter naming in `swap_tokens_for_any_nfts` function

Severity: Informational

In `contracts/infinity-swap/src/swap_processor.rs:686`, the `swap_tokens_for_any_nfts` function contains a parameter called `min_expected_token_input`, implying it represents a minimum expected input amount.

Since the function is used in `contracts/infinity-swap/src/execute.rs:873` and `contracts/infinity-swap/src/query.rs:428`, where the `min_expected_token_input` parameter is fed with `max_expected_token_input`, the naming is misleading as the function processes it as the maximum expected input amount.

Recommendation

We recommend renaming the parameter from `min_expected_token_input` to `max_expected_token_input`.

Status: Acknowledged

13. Developer address is not emitted during contract instantiation

Severity: Informational

In `contracts/infinity-swap/src/instantiate.rs:33-39`, the response emits all provided parameters as attributes, excluding the developer address. This is inconsistent as the provided developer address is not emitted for off-chain listeners to index.

Recommendation

We recommend emitting the developer's address if it is provided as `Some(_)`.

Status: Acknowledged

14. Inconsistent attribute name for pool identifiers

Severity: Informational

In `contracts/infinity-swap/src/execute.rs:353`, `445` and `contracts/infinity-swap/src/swap_processor.rs:83`, the attribute value for pool identifiers is used as `pool_id`. On the other hand, other functions use `id` to indicate the pool identifier, as seen in `contracts/infinity-swap/src/pool.rs:453`.

Recommendation

We recommend using `create_event` to emit events in the `execute_deposit_nfts` and `execute_withdraw_nfts` functions and modifying `contracts/infinity-swap/src/swap_processor.rs:83` to use `id` as the attribute name.

Status: Acknowledged

Appendix A: Test Cases

1. Test case for "Malicious users can transact NFTs with zero deltas"

The test case should fail if the issue is patched.

```
#[test]
fn unsaved_pools() {
    use cosmwasm_std::{Addr, Uint128, Timestamp};
    use cw721::OwnerOfResponse;
    use infinity_swap::msg::{ExecuteMsg, NftSwap, SwapParams};
    use sg721_base::{QueryMsg as NFTQueryMsg};

    // reproduced in test/unit/src/tests/pool_tests/token_pool_tests.rs

    let vt = standard_minter_template(5000);
    let (mut router, minter, creator, user1) = (
        vt.router,
        vt.collection_response_vec[0].minter.as_ref().unwrap(),
        vt.accts.creator,
        vt.accts.bidder,
    );

    let collection = vt.collection_response_vec[0].collection.clone().unwrap();
    let _asset_account = Addr::unchecked(ASSET_ACCOUNT);

    setup_block_time(&mut router, GENESIS_MINT_START_TIME, None);
    let marketplace = setup_marketplace(&mut router, creator.clone()).unwrap();
    let infinity_swap = setup_infinity_swap(&mut router, creator.clone(),
marketplace).unwrap();

    // 1. create two pools with same configurations

    // create pool_one
    let pool_one = create_pool(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        ExecuteMsg::CreateTokenPool {
            collection: collection.to_string(),
            asset_recipient: None,
            bonding_curve: BondingCurve::Linear,
            spot_price: Uint128::from(1000_u64),
            delta: Uint128::from(100_u64),
            finders_fee_bps: 0,
        },
    )
    .unwrap();
```

```
    deposit_tokens(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        pool_one.id,
        Uint128::from(5000u64),
    ).unwrap();

    let msg = ExecuteMsg::SetActivePool {
        is_active: true,
        pool_id: pool_one.id,
    };
    router.execute_contract(creator.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // create pool_two

    let pool_two = create_pool(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        ExecuteMsg::CreateTokenPool {
            collection: collection.to_string(),
            asset_recipient: None,
            bonding_curve: BondingCurve::Linear,
            spot_price: Uint128::from(1000_u64),
            delta: Uint128::from(100_u64),
            finders_fee_bps: 0,
        },
    ) .unwrap();

    deposit_tokens(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        pool_two.id,
        Uint128::from(5000u64),
    )
    .unwrap();

    let msg = ExecuteMsg::SetActivePool {
        is_active: true,
        pool_id: pool_two.id,
    };
    router.execute_contract(creator.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // mint nfts for user1
    let token_id_1 = mint(&mut router, &user1, minter);
    approve(
```

```
        &mut router,
        &user1,
        &collection,
        &infinity_swap.clone(),
        token_id_1,
    );

    let token_id_2 = mint(&mut router, &user1, minter);
    approve(
        &mut router,
        &user1,
        &collection,
        &infinity_swap,
        token_id_2,
    );

    let token_id_3 = mint(&mut router, &user1, minter);
    approve(
        &mut router,
        &user1,
        &collection,
        &infinity_swap.clone(),
        token_id_3,
    );

    let token_id_4 = mint(&mut router, &user1, minter);
    approve(
        &mut router,
        &user1,
        &collection,
        &infinity_swap,
        token_id_4,
    );

    // snapshot user balance
    let prev_balance = router.wrap().query_balance(user1.clone(),
NATIVE_DENOM).unwrap().amount;

    // 2. sell token_id_1
    let msg = ExecuteMsg::DirectSwapNftsForTokens {
        pool_id: pool_one.id,
        nfts_to_swap: vec![
            NftSwap { nft_token_id: token_id_1.to_string(), token_amount:
Uint128::new(1000) },
        ],
        swap_params: SwapParams {
            deadline:
Timestamp::from_nanos(GENESIS_MINT_START_TIME).plus_seconds(1_u64),
            robust: false,
            asset_recipient: None,
```

```
            finder: None,
        },
    };
    router.execute_contract(user1.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // 3. sell token_id_2
    let msg = ExecuteMsg::DirectSwapNftsForTokens {
        pool_id: pool_one.id,
        nfts_to_swap: vec![
            NftSwap { nft_token_id: token_id_2.to_string(), token_amount:
Uint128::new(900) },
        ],
        swap_params: SwapParams {
            deadline:
Timestamp::from_nanos(GENESIS_MINT_START_TIME).plus_seconds(1_u64),
            robust: false,
            asset_recipient: None,
            finder: None,
        },
    };
    router.execute_contract(user1.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // 4. calc profit amount for compare ltr
    let new_balance = router.wrap().query_balance(user1.clone(),
NATIVE_DENOM).unwrap().amount;
    let expected_amount = new_balance - prev_balance;

    // resnapshot balance
    let prev_balance = router.wrap().query_balance(user1.clone(),
NATIVE_DENOM).unwrap().amount;

    // 5. exploit
    let msg = ExecuteMsg::SwapNftsForTokens {
        collection: pool_two.collection.to_string(),
        nfts_to_swap: vec![
            NftSwap { nft_token_id: token_id_3.to_string(), token_amount:
Uint128::new(1000) },
            NftSwap { nft_token_id: token_id_4.to_string(), token_amount:
Uint128::new(1000) }
        ],
        swap_params: SwapParams {
            deadline:
Timestamp::from_nanos(GENESIS_MINT_START_TIME).plus_seconds(1_u64),
            robust: true,
            asset_recipient: None,
            finder: None,
        }
    };
```

```
    router.execute_contract(user1.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // token_id_3 has been transferred to creator
    let msg = NFTQueryMsg::OwnerOf { token_id: token_id_3.to_string(),
include_expired: None };
    let res : OwnerOfResponse =
router.wrap().query_wasm_smart(collection.clone(), &msg).unwrap();
    assert_eq!(res.owner, creator);

    // token_id_4 fails to transfer
    let msg = NFTQueryMsg::OwnerOf { token_id: token_id_4.to_string(),
include_expired: None };
    let res : OwnerOfResponse =
router.wrap().query_wasm_smart(collection.clone(), &msg).unwrap();
    assert_eq!(res.owner, user1);

    // 6. sell token_id_4 manually
    let msg = ExecuteMsg::DirectSwapNftsForTokens {
        pool_id: pool_two.id,
        nfts_to_swap: vec![
            NftSwap { nft_token_id: token_id_4.to_string(), token_amount:
Uint128::new(900) },
        ],
        swap_params: SwapParams {
            deadline:
Timestamp::from_nanos(GENESIS_MINT_START_TIME).plus_seconds(1_u64),
            robust: false,
            asset_recipient: None,
            finder: None,
        },
    };
    router.execute_contract(user1.clone(), infinity_swap.clone(), &msg,
&[]).unwrap();

    // token_id_4 has been transferred to creator
    let msg = NFTQueryMsg::OwnerOf { token_id: token_id_4.to_string(),
include_expired: None };
    let res : OwnerOfResponse = router.wrap().query_wasm_smart(collection,
&msg).unwrap();
    assert_eq!(res.owner, creator);

    // 7. calc excess profit
    let new_balance = router.wrap().query_balance(user1.clone(),
NATIVE_DENOM).unwrap().amount;
    let received_amount = new_balance - prev_balance;
    assert_eq!(received_amount, expected_amount);

}
```

2. Test case for "Pool activation is possible before depositing the required assets, resulting in an empty pool"

The test case should fail if the issue is patched.

```
#[test]
fn activate_without_deposit_token_pool() {
    // reproduced in test/unit/src/tests/pool_tests/token_pool_tests.rs

    let vt = standard_minter_template(5000);
    let (mut router, _minter, creator, user1) = (
        vt.router,
        vt.collection_response_vec[0].minter.as_ref().unwrap(),
        vt.accts.creator,
        vt.accts.bidder,
    );

    let collection = vt.collection_response_vec[0].collection.clone().unwrap();
    let _asset_account = Addr::unchecked(ASSET_ACCOUNT);

    setup_block_time(&mut router, GENESIS_MINT_START_TIME, None);
    let marketplace = setup_marketplace(&mut router, creator.clone()).unwrap();
    let infinity_swap = setup_infinity_swap(&mut router, creator.clone(),
marketplace).unwrap();

    let pool = create_pool(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        ExecuteMsg::CreateTokenPool {
            collection: collection.to_string(),
            asset_recipient: None,
            bonding_curve: BondingCurve::Linear,
            spot_price: Uint128::from(2400u64),
            delta: Uint128::from(100u64),
            finders_fee_bps: 0,
        },
    )
    .unwrap();

    // Owner of pool can activate pool
    let msg = ExecuteMsg::SetActivePool {
        is_active: true,
        pool_id: pool.id,
    };
    let res = router.execute_contract(creator, infinity_swap, &msg, &[]);
    let is_active = &res.as_ref().unwrap().events[1].attributes[2].value;
    assert_eq!(is_active, &"true");
}
```

3. Test case for "Trades will fail if the delta decreases the spot price to zero"

The test case should pass if the issue is patched.

```
#[test]
fn error_updating_spot_price() {
    let vt = standard_minter_template(5000);
    let (mut router, minter, creator, user1) = (
        vt.router,
        vt.collection_response_vec[0].minter.as_ref().unwrap(),
        vt.accts.creator,
        vt.accts.bidder,
    );

    let collection = vt.collection_response_vec[0].collection.clone().unwrap();
    let asset_account = Addr::unchecked(ASSET_ACCOUNT);

    setup_block_time(&mut router, GENESIS_MINT_START_TIME, None);
    let marketplace = setup_marketplace(&mut router, creator.clone()).unwrap();
    let infinity_swap = setup_infinity_swap(&mut router, creator.clone(),
marketplace).unwrap();

    let pool = create_pool(
        &mut router,
        infinity_swap.clone(),
        creator.clone(),
        ExecuteMsg::CreateTokenPool {
            collection: collection.to_string(),
            asset_recipient: Some(asset_account.to_string()),
            bonding_curve: BondingCurve::Linear,
            spot_price: Uint128::new(1000),
            delta: Uint128::new(1000),
            finders_fee_bps: 0,
        },
    ).unwrap();

    // deposit tokens
    let deposit_amount = 10_000_u128;
    let msg = ExecuteMsg::DepositTokens { pool_id: pool.id };
    router.execute_contract(
        creator.clone(),
        infinity_swap.clone(),
        &msg,
        &coins(deposit_amount, NATIVE_DENOM),
    ).unwrap();

    // activate pool
    let msg = ExecuteMsg::SetActivePool { is_active: true, pool_id: pool.id };
    router.execute_contract(
```

```
        creator.clone(),
        infinity_swap.clone(),
        &msg,
        &[],
    ).unwrap();

    // mint nfts
    let token_id_1 = mint(&mut router, &user1, minter);
    approve(
        &mut router,
        &user1,
        &collection,
        &infinity_swap.clone(),
        token_id_1,
    );

    let token_id_2 = mint(&mut router, &user1, minter);
    approve(
        &mut router,
        &user1,
        &collection,
        &infinity_swap,
        token_id_2,
    );

    // swap nfts for tokens
    let msg = ExecuteMsg::SwapNftsForTokens {
        collection: pool.collection.to_string(),
        nfts_to_swap: vec![
            NftSwap { nft_token_id: token_id_1.to_string(), token_amount:
Uint128::new(0) },
            NftSwap { nft_token_id: token_id_2.to_string(), token_amount:
Uint128::new(0) },
        ],
        swap_params: SwapParams {
            deadline:
Timestamp::from_nanos(GENESIS_MINT_START_TIME).plus_seconds(1_u64),
            robust: false,
            asset_recipient: None,
            finder: None,
        }
    };
    router.execute_contract(
        user1.clone(),
        infinity_swap.clone(),
        &msg,
        &[],
    ).unwrap();
}
```