BlockApex

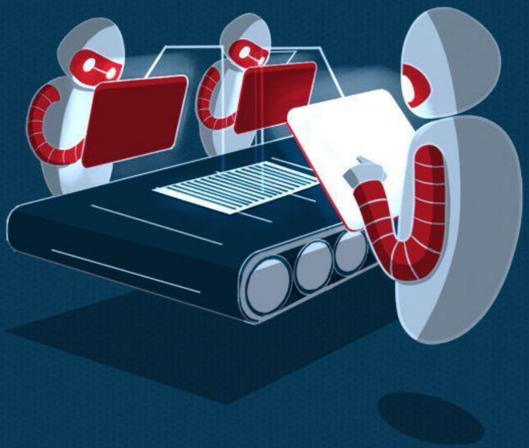
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Powered by XORD

PREFACE

Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings

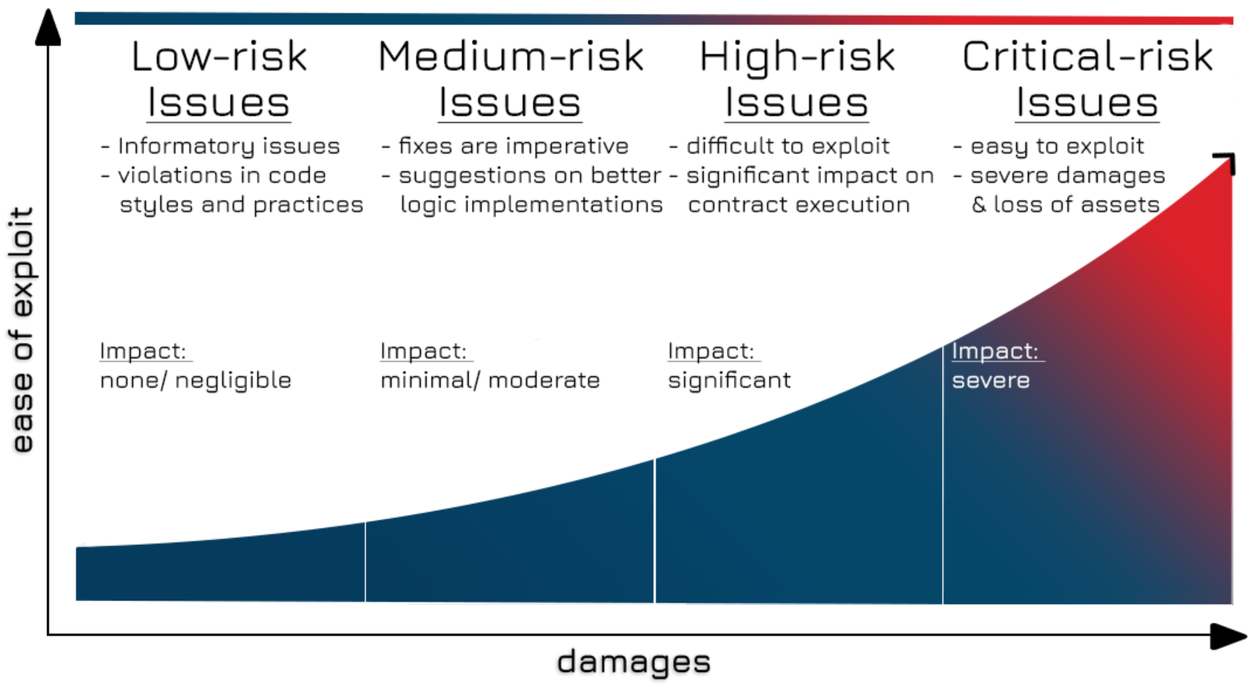


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Jump Defi (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which started on 05 Oct 2022.

Name
Jump Defi Decentralized Finance Platform
Auditor
BlockApex
Platform
NEAR Protocol / Rust
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository/ Commit Hash
ddb92dda6eb779ac854471eeda817abeacfc054e
White paper/ Documentation
Docs
Document log
Initial Audit Completed: Oct. 19th, 2022
Final Audit: (Nov. 4th, 2022)

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulnerabilities**. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	FT token API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Jump Defi infrastructure built on NEAR Protocol, a reliable and scalable L1 solution. Jump Defi is a one-stop solution for all core Defi needs on NEAR. Jump ecosystem has a diverse range of revenue-generating products which makes it sustainable.

Jump Defi's diverse range of protocols increases the visibility of other projects on NEAR which potentially increases the mass adoption of NEAR.

System Architecture

Jump Dex (Out of Scope)

is a fully permissionless automated market maker in which a user can trade or become a liquidity provider on the NEAR protocol to earn platform rewards.

Jump Dex AMM is powered by two constant product functions, inspired by Uniswapv2 and Curve Finance:

- Swap function: $x * y = k$
- StableSwap function: $\chi Dn-1 * \Sigma xi + \prod xi = \chi Dn + (D / n)n$

Jump Pad

Provides a manner for cryptocurrency projects to raise liquidity via Initial DEX Offering (IDO) and private sales.

Jump Pad IDO has two rounds, round one is for JUMP token holders and round two is open to the public to purchase any unsold tokens on a first come first serve basis. 1% of fees on IDO raises which uses to buy and deposit JUMP into the xJUMP pool

JUMP Pad private sales are for VC Investors only. The fees on private sales are decided on a project-by-project basis which also uses to buy and deposit JUMP into the xJUMP pool

Jump Token Laboratory

Anyone can create a NEP-141 fungible token with the standard using Jump token laboratory with customizable code and intelligent tokenomics.

Jump NFT Staking

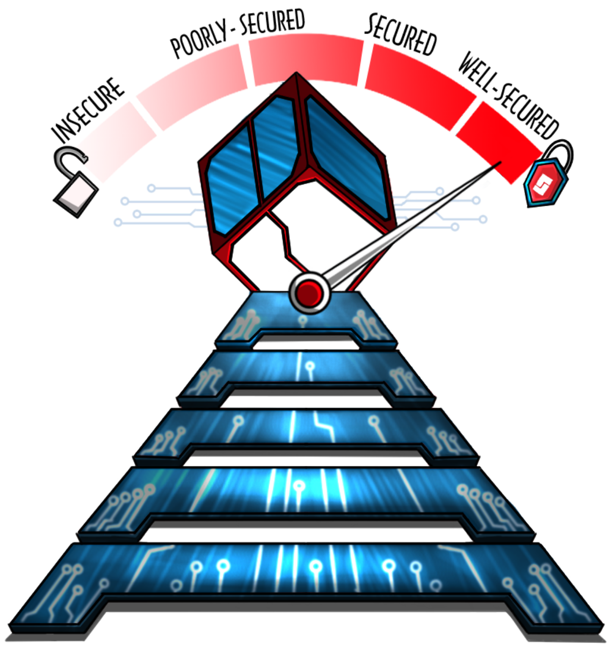
It provides a staking-as-a-service infrastructure for NFT collection to efficiently structure fungible token rewards for their NFT holders. Approved NFT collection can have triple token rewards.

Methodology & Scope

The codebase was audited using a filtered audit technique. A band of four (4) auditors scanned the codebase in an iterative process spanning over a time of two (2) weeks.

Starting with the recon phase, a basic understanding was developed and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives that were detected by automated analysis tools.

AUDIT REPORT

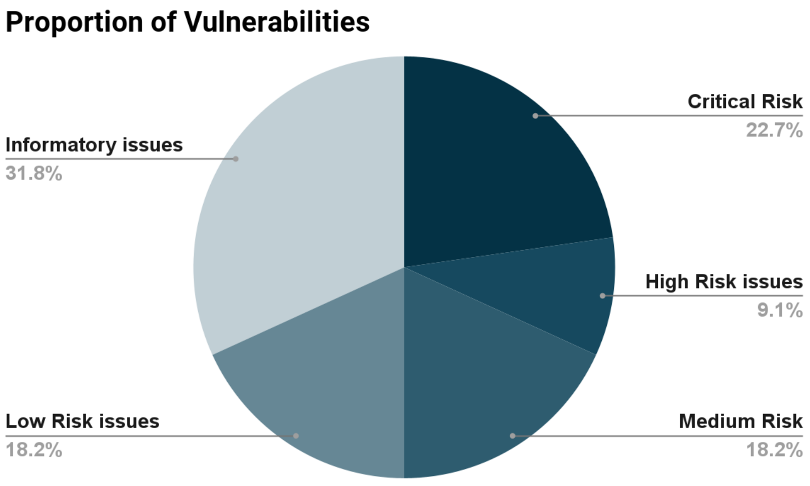


Executive Summary

Our team performed a technique called "Filtered Audit", where the contract was separately audited by four individuals. After a thorough and rigorous process of manual testing, an automated review was carried out using cargo audit & cargo-tarpaulin for static analysis and cargo-fuzz for fuzzing invariants. All the flags raised were manually reviewed and re-tested to identify the false positives.

Our team found:

# of issues	Severity of the risk
5	Critical Risk issue(s)
2	High Risk issue(s)
4	Medium Risk issue(s)
4	Low Risk issue(s)
7	Informatory issue(s)



Key Findings

#	Findings	Risk	Status
1	Improper access controls leads to liquidity theft	Critical	Not Applicable
2	Inexistent logic to unregister storage	Critical	Fixed
3	Inadequate implementation leads to loss of rewards	Critical	Fixed
4	Insufficient Access Controls on NFT Minting	Critical	Not Applicable
5	Users will not be able to withdraw their locked tokens even after they are unlocked	Critical	Fixed
6	Improper Accumulation of Rewards	High	Fixed
7	Inefficient NFT staking reward distribution	High	Fixed
8	Improper evaluation of `dex_lock_time`	Medium	Not Applicable
9	Collision of Guardians to steal contract tokens	Medium	Fixed
10	Inadequate state roll back will lead to reward loss	Medium	Acknowledged
11	Potential “million cheap data additions” attack	Medium	Fixed
12	Inadequate function parameters	Low	Acknowledged
13	Insufficient logic for storage cost	Low	Fixed
14	Potential avoidance of launchpad fee	Low	Fixed

15	Inadequate validation on AccountId	Low	Fixed
16	Improper validation of deploy address	Informational	Fixed
17	Inadequate checks on unregister storage	Informational	Fixed
18	Optimization in withdraw	Informational	Fixed
19	Standard Conformity and Inconsistencies in the requirements	Informational	Fixed
20	Incomplete Functions in the Contracts	Informational	Fixed
21	Redundant code in calculate_vested_investor_withdraw function	Informational	Fixed
22	Optimization in withdraw_rewards	Informational	Fixed

Detailed Overview

Critical-risk issues

1. Improper access controls leads to liquidity theft

File: launchpad/src/actions/dex_launch_actions.rs

Description:

Users can add liquidity to partner_dex once the `LOCK_PERIOD` has expired. Although this ensures the public launch on a specified date, it introduces a sandwich attack where attackers can steal the liquidity.
To add liquidity, a user has to perform the following steps

1. Tx1: `launch_on_dex` → matches `ListingStatus::SaleFinalized` → cross-contract call to create pool → update the status to `ListingStatus::PoolCreated`
2. Tx2: `launch_on_dex` → matches `ListingStatus::PoolCreated` → cross-contract call to send Project token → update the `ListingStatus::PoolProjectTokenSent`
3. Tx3: `launch_on_dex` → matches `ListingStatus::PoolProjectTokenSent` → cross-contract call to send price token → update `ListingStatus::PoolPriceTokenSent`
4. Tx4: `launch_on_dex` → matches `ListingStatus::PoolPriceTokenSent` → cross-contract call to add liquidity → update `ListingStatus::LiquidityPoolFinalized`

Background: We assume the attack scenario by building the underlying assumption of DEX like Uniswap V2. Uniswap V2 adds liquidity into the pool in three steps in a single transaction.

1. It takes token A then & sends it to the pool.
2. It takes token B then & sends it to the pool.
3. It calls the `mint` function which actually mints the LP shares to the caller

These steps are necessary to be called in an atomic transaction, uniswap achieves atomicity through its periphery contract. See the code below from uniswapV2 periphery contract.
Code Reference: Uniswap V2 repository

```
function addLiquidity(address tokenA, address tokenB, uint
amountADesired,
        uint amountBDesired, uint amountAMin, uint amountBMin,address
to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint
amountA, uint amountB, uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB,
amountADesired, amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA,
tokenB);
              // Below are the 3 steps in a single tx
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair,
amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair,
amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Since launch_on_dex needs 4 transactions to add liquidity, the attacker can create the following smart contract to steal the liquidity.

Code:

```
pub fn steal_liquidity(victim: AccountId, listingId: u64,
partner_dex:AccountId) -> Promise {
  ext_dex::ext(victim)
          .with_static_gas(DEX_INTERACTION_GAS)
          .with_attached_deposit(deposit)
          .launch_on_dex(
            listingId
          ); // call this 3 time, so that both tokens are transferred
to dex, instead calling Tx:4 attacker directly adds liquidity to the
dex
 ext_dex::ext(partner_dex)
          .with_static_gas(DEX_INTERACTION_GAS)
          .with_attached_deposit(deposit)
          .add_liquidity(
            listing.dex_id.unwrap(),
            vec![
              U128(listing.dex_project_tokens.unwrap()),
              U128(listing.dex_price_tokens.unwrap()),
            ],
            None,
          )
}
```

Recommendation: Ensure the atomicity in all 4 transactions by following the UnswapV2 periphery logic

Dev's Response #1:
Not Applicable -> alternative measures taken

The auditors claim that liquidity can be stolen in case the attacker does not complete the add liquidity lifecycle and that the transactions should be atomic in order to preserve security. However, these claims are based on the model followed by Uniswap V2 code on EVM based blockchains.
The Near Protocol has some properties that disallow such atomicity from being achieved, given that cross contract calls are performed across different shards through promises processed in different blocks, hence cross contract calls CANNOT preserve atomicity.
Moreover, the reference dex implementation, on which we are basing the code comes from ref.finance. In this model, it is necessary to perform 4 transactions in order to create a pool:

1. *create pool*
2. *Add token 1 to launchpad's internal account in ref.finance*
3. *Add token 2 to launchpad's internal account in ref.finance*
4. *Take tokens from launchpad's internal balance and add them to liquidity in the pool created in (1)*

In this model, there is no way to steal the liquidity deposited in steps 2 and 3 since it is locked in an internal account inside ref's contract that can only be accessed by its owner (the launchpad contract). ***If any other account tries to call add_liquidity their transaction will fail since they do not have the necessary internal balance deposited.***
However, given that the JUMP DEX is currently not operational and that until it is the automatic generation of liquidity pools should not be used inside listings, ***we have changed the launch_on_dex method to always panic and made it impossible to set automatic IDOs for new sales created.***
The code will be updated to revert the changes as soon as the JUMP DEX is launched and a full integration can be built.

Auditors' Response #1: *After reviewing the ref.finance dex code, auditors agree with the devs. However, there is no functionality to remove liquidity from dex in Jump which might result in liquidity being permanently locked in the dex contract.*

Dev's Response #2:

This is precisely the idea, the project would be committing part of their capital permanently to providing liquidity on the DEX. This is equivalent to burning LP shares

Auditors' Response #2:

The auditors agree with the devs.

2. Inexistent logic to unregister storage

File: Nft_Staking/src/actions/storage_impl.rs

Description:

In `Nft_Staking/src/actions/storage_impl.rs` the `storage_unregister` function will always fail at assertion forbidding investors to unregister storage.

Code:

```
#[payable]
  fn storage_unregister(&mut self, force: Option<bool>) -> bool {
    assert_one_yocto();
    let account_id = env::predecessor_account_id();
    if let Some(account_deposit) =
self.internal_get_investor(&account_id) {
      // TODO: figure out force option logic.
      // assert!(account_deposit.allocation_count.is_empty(), "{}",
ERR_203);
      assert!(false); // will always revert
      self.investors.remove(&account_id);

Promise::new(account_id.clone()).transfer(account_deposit.storage_depo
sit);
```

```
      true
    } else {
      false
    }
```

Recommendation: Remove assertion.
Alleviation: This issue is **fixed.**

3. Inadequate implementation leads to loss of rewards

File: Staking/src/staking.rs

Description:

Stakers can claim $JUMP as their staking rewards by calling the claim function. The function pulls the user's data and calculate the latest rewards as follows

```
user.unclaimed_rewards + ((user.balance * (contract_rps - user.user_rps)) /
FRACTION_BASE)
```

Apparently at `line#45`, the function then updates the unclaimed amount as `user.unclaimed_rewards = 0`;before making the transfer call at `line#51`. Although setting unclaimed rewards to 0 is required, the current implementation leads to a critical flaw where the user gets nothing as the `token_contract::ft_transfer` function uses the latest amount for unclaimed rewards. See the code snippet below

```
pub fn claim(&mut self) -> Promise {
  ...
   user.unclaimed_rewards = 0; //@audit

   self.user_map.insert(&account_id, &user);
```

```
    token_contract::ft_transfer(
      account_id.to_string(),
      U128(user.unclaimed_rewards), //@audit sends 0
      "Claimed #{amount}".to_string(),
      &self.token_address,
      1,
      BASE_GAS,
    )
  }
```

Recommendation: Use a temporary variable to store unclaim rewards

Dev's Response:

The issue is acknowledged, however the staking contract will be discontinued as the team opted to go along only with the xJUMP model for token staking.
The staking contract has been excluded from the repository.

Alleviation: The developers removed the staking contracts from the repository making it inapplicable.

4. Insufficient Access Controls on NFT Minting

File: locked_token/src/actions/user.rs

Description:

The current implementation uses the internal_mint function to mint new NFTs by calling `nft_mint function`. Internal functions generally do not implement validations of any kind and rely on public wrappers to ensure security controls. The following code snippet exhibits the lack of proper access controls.

```
pub fn nft_mint(&mut self, receiver_id: Option<AccountId>) -> Token {
    let account_id =
receiver_id.unwrap_or(env::predecessor_account_id());
```

```
    ...

    let token = self
      .tokens
      .internal_mint(token_id, account_id, Some(metadata)); //@audit
no-auth checks

    token
  }
```

Recommendation: Use mint functionality from NEP-171 which has a vetted codebase containing appropriate access controls.

Dev's Response:

Issue refers to testnet only contract, no need to implement

Alleviation: Out-of-Scope as it is used for testing.

5. Users will not be able to withdraw their locked tokens even after they are unlocked

File: locked_token/src/actions/user.rs

Description: Gas for `FT_TRANSFER_GAS` & `FT_TRANSFER_CALLBACK_GAS` is set to `zero` in `withdraw_locked_tokens` function. This will always result in cross-contract call failure.

Recommendation: Set FT_TRANSFER_GAS & FT_TRANSFER_CALLBACK_GAS to standard cross-contract gas.

Alleviation: The issue is **fixed** by allocating enough gas for a successful transaction.

High-risk issues

6. Improper Accumulation of Rewards

File: Nft_Staking/src/actions/staker.rs

Description:

Users can earn rewards by staking their NFTs. It is worth noting that code documentation says about claiming rewards

```
The existence of these transactions is purely for a technical reason.
When combined they allow a staker to withdraw their rewards. While
inner_withdraw transfers a Staked NFT Balance to the respective Staker
Balance, outer_withdraw allows the staker to withdraw a specific
fungible token from their Staker Balance to their Staker Wallet.

As the Near API allows for batch calls, this won't be noticed by the
user in the frontend. However CLI users need to be aware of these
details
```

Although the above statements informs CLI users to be careful, in the current implementation rewards would be lost if users called the `claim_reward` function more than once before calling the `withdraw_reward`.

The issue arises due to the way the balances are updated in `claim_reward`. `withdraw _reward` utilizes the `StakingProgram::stakers_balances` to get the claimable amount. `StakingProgram::stakers_balances` is updated by adding `new_rewards` and `StakedNFT::balance`. However when the claim function is called `StakedNFT::balance` is set. Therefore when a claim function is called again, `StakingProgram::stakers_balances` is updated only with `new_rewards`, losing the previous rewards

Code

```
pub fn withdraw(&mut self) -> FungibleTokenBalance {
    let withdrawn_balance = self.balance.clone();
    for (_, v) in self.balance.iter_mut() {
      *v = 0; //@audit, balance set to zero
    }
    withdrawn_balance
  }
```

```
pub fn inner_withdraw(&mut self, token_id: &NonFungibleTokenID) ->
FungibleTokenBalance {
    let mut staked_nft = self.claim_rewards(token_id);
    let withdrawn_balance = staked_nft.withdraw(); //@audit updates the
states to zero
    let owner_id = &staked_nft.owner_id;
...
    self.stakers_balances.insert(&owner_id, &balance); //@audit only updated
with new balance
    self.staked_nfts.insert(&token_id, &staked_nft);

    balance
  }
```

Recommendation:

We suggest to implement one of the following

1. Instead of replacing `self.stakers_balances.insert(&owner_id, &balance)`, it should add the newer rewards to the old.
2. There should be a mutex here, which would not allow the `inner_reward` to be called again until `outer_reward` is called.

Alleviation: The issue is **fixed** by following recommendation #1.

7 Inefficient NFT staking reward distribution

File: Nft_Staking/src/models/farm.rs

Description:
The function for reward calculations for NFT Staking is flawed in many ways.

1. When a user stakes NFT, `Farm::nfts_rps` should be set to updated value `Farm::distribution.rps`. So when the rewards are claimed, the user gets his fair share. But in current implementation, the `Farm::nfts_rps` is set to the older value of `Farm::distribution.rps`. This implementation is flawed because this enables the user to claim extra rewards. As we see in the code mentioned below `Farm::nfts_rps` is updated before updating `Farm::distribution.rps`

CODE:

```
pub fn add_nft(&mut self, nft_id: &NonFungibleTokenID) {
    let mut balance = HashMap::new();

    for (ft_id, dist) in self.distributions.iter() {
      balance.insert(ft_id.clone(), dist.rps);
    }

    self.nfts_rps.insert(nft_id, &balance);
    self.distribute(); // TODO: confirm this is a bug, and
should in fact happen before RPS assignment
  }
```

Consider the following scenario for better understanding

```
// at round 2 rps was 10
round = 2
Farm::distribution.rps = 10

// at round 5, the user stakes his NFT
round = 5
Farm::distribution.rps = 10 + 3 rounds rewards which greater than 10
Farm::nfts_rps = 10 // set to older value

//at round 5, the user immediately claims his rewards
Farm::distribution.rps = 10 + 3 rounds rewards which is greater than
10
reward = Farm::distribution.rps - Farm::nfts_rps  = 10 + 3 rounds
rewards - 10 > 0

expected_rewards = 0 //since claimed rewards in same transaction
actual_rewards > 0
```

Recommendation:

There is already a TODO left. Changing the call order would fix the issue

```
//current implementation
self.nfts_rps.insert(nft_id, &balance);
self.distribute();
```

```
//fix
self.distribute();
self.nfts_rps.insert(nft_id, &balance);
```

Alleviation: The issue is fixed

2. The staker can claim the reward for the entire round even staking for a very short duration discouraging users to stake for the entire round. This is due to the fact the reward is calculated on a per round basis.

```
pub fn claim(&self, token_rps: U256) -> (Self, u128) {
    let mut dist = self.clone();
    let claimed = denom_convert(self.rps - token_rps); // see
below

    dist.unclaimed -= claimed;

    (dist, claimed)
  }
```

Consider the following scenario for better understanding.
Let suppose the following variables

```
round_interval = 10 mins
reward_per_round = 10 tokens
Duration passed = 29 mins // from farm.start_at
NFTs staked (seed) = 3
```

AT `t=29mins` the user(attacker) stakes an NFT increasing the `total_seeds` to 4. The global variables will be updated as follows

```
total_passed_rounds = delat_t/round_interval = 29/10 = 2 round
```

```
added_reward = tolata_passed_rounds * reward_per_round = 10
tokens
Prev_rsp = 2 * 10 / 3 = 6.66
Farm::distributions.rps = prev_rps + added_reward/total_seeds =
6.66 + 10/4 = 9.1
Farm::nfts_rps_<nft_id> = 9.1
```

At `t=31 mins`, the attacker claims the reward. Although only 2 mins have passed, the attacker will get the entire round's reward. The claimable reward is calculated as

```
total_passed_rounds = delat_t/round_interval = 31/10 = 3 rounds
added_reward = tolata_passed_rounds * reward_per_round = 10 tokens
//total_passed_round is 1, since the last action was at t = 29 mins

Farm::distributions.rps = 9.1 + 2.5 =11.6
claimableReward = 11.6 - 9.1 = 2.5
```

Recommendation: We recommend implementing a linear reward schedule independent of rounds. Or making the round intervals short to avoid rounding errors

Dev's Response #1:
The interval between rounds is determined when creating a Staking Program.
Note that the shorter the round interval, the more closely to linear on time the reward distribution function ends up as. ***It's recommended to set it to a couple of seconds at most.***

Auditors' Response #1: *We suggest introducing an upper cap for the round interval as described in the dev's response. The documentation suggests short round intervals but* ***this is not reflected in the smart contract****.*

Dev's Response #2:
This has been included in the user documentation. We don't feel like there should be a hard limit since different projects might have different ideas on how to use the contract. For instance, a NFT project might want to reward all skaters that have stakes before a

certain date. This would be done by implementing a large round interval and distributing all rewards in a single round.

Auditors' Response #2:
The auditors agree with the devs.

Medium-risk issues

8. Improper evaluation of `dex_lock_time`

File: launchpad/src/actions/dex_launch_actions.rs

Description: When the lauch_on_dex function is called, it checks if `dex_lock_time` is less than current timestamp & sets it's value to `timestamp + LOCK_PERIOD` but the callback always resets the value back to zero.

Code:

```
pub fn launch_on_dex(&mut self, listing_id: U64) -> Promise {
    let timestamp = env::block_timestamp();
    let mut listing = self.internal_get_listing(listing_id.0);
    assert!(listing.dex_lock_time <= timestamp, "{}", ERR_403); //here
    listing.update_treasury_after_sale();
    listing.dex_lock_time = timestamp + LOCK_PERIOD; //here
 ...
}
```

Calling this function updates the status of a listing & attaches a callback. Every callback always sets the value of dex_lock_time to zero. See the code snippet below.

Code:

```
pub fn callback_dex_launch_create_pool(
 ...
  ) -> PromiseOrValue<bool> {
    let mut listing = self.internal_get_listing(listing_id.0);
    listing.dex_lock_time = 0; //here    ...
```

This will always allow the function to be called without LOCKED_PERIOD applied. This can be found at the following functions. i.e `listing.dex_lock_time = 0;`

1. In `Callback_dex_add_liquidity`, callback_actions.rs
2. In `Callback_dex_deposit_project_token`, callback_actions.rs
3. In `Callback_dex_deposit_price_token`, callback_actions.rs
4. In `Callback_dex_add_liquidity`, callback_actions.rs

Recommendation: Introduce proper validations

Dev's Response

Not Applicable

The idea behind it is that if the `launch_on_dex` method is called more than once in the same block, all calls but the first will revert. The lock will only be lifted after the execution of the callback to allow the users to proceed with the next phase of the launch.
The reason it is implemented as a timestamp instead of a boolean value is that in case some unintended bug affects the callback, funds do not get locked forever on the contract.
Therefore, no modifications were done to the contract on this issue's account.

Auditor's repsonse: *The auditors agree with the developers.*

9. Collision of Guardians to steal contract tokens

File: nft_staking/src/funds/transfer.rs

As this is only exploitable by the guardian, and currently guardians are only created by the owner, but this could be changed in the future when DAO is implemented, as we have seen in the countless hacks in the DeFI ecosystem where the attacker was able to become a guardian, moreover in the specification of JUMPDeFi, it was written that only owner is able to interact with the `contract_treasury` but in the code implementation only the guardians are able to interact with it. The attack scenario is a bit complex but highly likely and very easy if a malicious user is able to become a guardian.

Description:

When the `nft_staking` is deployed the owner is set at the time of deployment, the nft_staking will have a single `contract_treasury`. Now the new staking-programs can be deployed and they will have their own collection_treasury. Each staking program will have a farm in which the tokens to be distributed will be set, although the `contract_tokens` and `programs_tokens` will have to be different, `contract_tokens` can be added in the distribution which will eventually be distributed among the Stakers.

The other thing that we need to consider in it, is that inside the `reallocate_funds` function the guardian is able transfer `contract_token` from `contract_treasury` into `collection_treasury`, (the guardian should not be able to interact with `contract_treasury` only the owner should have done it).

After the `contract_tokens` are transferred into `collection_treasury`, it can be moved to distribution by using `TransferOperation::CollectionToDistribution` in `reallocate_funds`. Once the contract tokens are in distribution it would be distributed to the stakers.

Attack Scenario:
Now the attack pattern will look like, once the attacker is able to become a guardian.

1. Guardian will create a new `staking_program` and add the contract token as a distribution reward here in `create_staking_program` in guardian.rs.
2. Staking program is created with distribution tokens set similar to `contract_token`. Guardian will transfer the contract token from contract treasury into collection treasury. And then transfer them from the collection treasury into distribution. All in the `reallocate_funds` function.
3. Attacker will only stake himself in the `staking_program`, and set the staking time very short. In this way they will be able to take out all the contract tokens as reward because they will be the only staker and the way `distribute` function works.

Code:

```
fn reallocate_funds(
 ...
 )
match operation {
     TransferOperation::ContractToCollection => {
       let contract_treasury =
self.contract_treasury.entry(token_id.clone()).or_insert(0);
       let amount = amount.unwrap_or(*contract_treasury);
       assert!(
         amount <= *contract_treasury,
         "{ERR_INSUFFICIENT_CONTRACT_TREASURY}"
       );
       *contract_treasury -= amount;
       *collection_treasury += amount;
     }
...
```

```
    }
```

Recommendation: As per the specifications, only the owner should be able to transfer funds to the Collection Treasury instead of guardians. That is the quickest way to ensure safety here.

Alleviation: The issue is **fixed** by placing an onlyOwner check.

10. Inadequate state roll back will lead to reward loss

File: Nft_Staking/src/actions/staker.rs

Description:

Users can un-stake their NFT through the contract. During the call to transfer NFT it attaches the callback `compensate_unstake` function. If the promise fails it restakes the NFT which in turn calls `redistribute` function which will result in loss of reward for the user.

Code:

```
pub fn compensate_unstake(
   &mut self,
   token_id: NonFungibleTokenID,
   owner_id: AccountId,
   staked_timestamp: U64,
   balance: SerializableFungibleTokenBalance,
 ) {
   if is_promise_success() {
     events::unstake_nft(&token_id, balance);
   } else {
     let collection = token_id.0.clone();
     let staked_nft = StakedNFT::new(token_id, owner_id,
staked_timestamp.0);
     let mut staking_program =
self.staking_programs.get(&collection).unwrap();
```

```
    staking_program.insert_staked_nft(&staked_nft);
    self.staking_programs.insert(&collection, &staking_program);
```

Recommendation:

Introduce a mechanism to only revert the changes that were caused rather than making a new entry

Dev's Response:

Essentially, the problem is that once you un-stake the total amount of staked NFTs gets reduced. This means rewards during the blocks that have the NFT unstaked are increased for every present NFT. In order for us to give rewards for the unstaked NFT in that period, we would need to recalculate the rewards given away, since we'd have to include 1 more NFT for the distribution. This cannot be done safely as other users might have claimed their rewards in the period

This would introduce a lot of race conditions in the code essentially. The callback is already a failsafe for cases where the NFT transfer fails, the user missing a couple blocks of rewards is a small loss considering that we already implemented a mechanism for him to not have to wait for the lock period again And in practice there is no reason for a nft_transfer to fail, since the contracts do not require storage deposits. In case this happens, it would probably be necessary to contact the technical team as the NFT's contract might have a bug.

Auditor's Response:

The auditors agree with the devs.

11. Potential "million cheap data additions" attack

File: launchpad/src/actions/guardian_actions.rs

Description:

Since the investors have to pay storage costs. The guardians can create fake project listings on the launchpad with the project owner being any investor (i.e victim). `track_storage_usage` will increase the `storage_used` and when investors call `storage_withdraw` he/she will get less amount.

Since the guardians have to undergo strict KYC and other criteria, this is very unlikely that guardians will perform this attack. But we have to consider a case when the platform softens the policies for guardians, this attack makes more sense

Code:

```
pub fn create_new_listing(&mut self, listing_data: ListingData) ->
u64 {
   self.assert_owner_or_guardian();
   let initial_storage = env::storage_usage();
   let project_owner_account_id = listing_data.project_owner.clone();
   let mut project_owner_account = self
     .internal_get_investor(&project_owner_account_id)
     .expect(ERR_010);
   let listing_id = self.internal_create_new_listing(listing_data);
   project_owner_account.track_storage_usage(initial_storage);
   self.internal_update_investor(&project_owner_account_id,
project_owner_account);
   listing_id
 }
```

Recommendation:
Implements a consent/permission mechanism for investors to be project owners. The following are the approaches.

- if the owner is EOA, the signature should be provided
- if the owner is a smart contract, then add a callback `verify_permission_for_listing` on SC while creating a listing

Dev's Response:

The auditors found that a malicious guardian, either by malicious intent or by hacking of their private keys, might spam the contract with multiple unasked for listings and assign random investors as their owners, which would cause the investors to pay for the storage needed and would make them unable to recover their storage fees forever.

The suggested fix was to require a signature of the project_owner for a specific message

(if address is a user wallet) or to do a callback to verify the intent (if address is a smart contract).
However, we found that such methods significantly deteriorate the ease of use of the contract, especially when setting up a new sale for a project.
As a solution, we implemented the boolean attribute authorized_listing_creation in the investor struct. It is default false and the user must call the new public method ***toggle_authorize_listing_creation*** *to toggle its value to true.*
Guardians can only call create_new_listing using the address of the project_owner if that address is registered as an investor with the authorized_listing_creation attribute set to true.
After creating a new listing, ***authorized_listing_creation*** *is set back to false to prevent the creation of multiple undesired listings.*

Alleviation: The issue is **fixed** by introducing a boolean attribute \``authorize_listing_creation`\`. The fix introduced by the developers is very efficient.

Low-risk issues

12. Inadequate function parameters

File: vesting_contract/src/investment.rs

Description:

In `vesting_contract/src/investment.rs` the new function the `date_in` parameter should not be optional.

Here when the user will go on to create an investment on a specific schema in the farm, the user will have the option to keep the `initial_timestamp` of the investment which is date_in parameter as optional. If it is optional the `initial_timestamp` of the investment will be the schema's `initial_timestamp`. Now this could cause a lot of issues. The main one is if the schema was created way back in time and investment was created recently, the user can easily increase their reward and could bypass the needed cliff period. As in the `calculate_available_withdraw` on these lines calculates whether the user has passed the cliff and vesting period and the rewards are calculated accordingly.

Only one schema is created for one category and there cannot be more than one schema for a single category, moreover the schema can only be created by the token contract means it is only callable by owner. So we can expect that new schemas will not be created everyday but new investments by user will be created all the time.

Code:

```
pub fn new(account: AccountId, total_value: u128, date_in: 
Option<u64>) -> Self {
      ...
}
```

Recommendation: The parameter `` `date_in` `` shouldn't be made optional.

Dev's Response

This is the intended behavior, as the method can only be called by the owner, they must have the autonomy to actually select the date in which the investor entered the investment agreement.

The main reason for the behavior is that, in practice, many investors already put funds into Jump DeFi, even before the contracts were deployed as they were early investors. Therefore, their vesting schemas have already been started and must be faithfully reproduced, which might require the owner to set a previous start_date.

Auditors' Response: We agree with the dev's response.

13. Insufficient logic for storage cost

File: token_launcher/src/lib.rs

Description:

In the lib.rs contract of token_launcher, the user can register as many cratract and the storage cost for them will be borne by the contract instead of users paying for the storage. This would lead to a griefing attack where the attacker can mass register and try to put as much storage cost on the contract as possible.

Code:

```
pub fn register_contract(
   &mut self,
   contract_name: String,
   contract_hash: Base58CryptoHash,
   contract_cost: U128,
   init_fn_name: String,
   init_fn_params: String,
 ) {
   assert!(
  ...
   self.binaries.insert(&contract_name, &new_binary);//storage cost
   event_new_contract_registered(contract_name.as_str(),
contract_hash);
 }
```

Recommendation: A unregister_contract function should be created that would aid the user/contract owner in removing registration and freeing up the storage when necessary. Furthermore the storage cost should be paid by the user instead of the contract.

Alleviation: The issue is **fixed** by introducing onlyOwner check.

14. Potential avoidance of launchpad fee

File: launchpad/src/actions/guardian_actions.rs

Description:

The guardian can cancel the listing just before its ending period or phase 2 start time i.e an hour before. This will allow investors to withdraw the project tokens after vesting period. The project owners will be able to withdraw their project & price tokens without paying a fee to the launchpad.

Code

```
#[payable]
 pub fn cancel_listing(&mut self, listing_id: U64) {
   self.assert_owner_or_guardian();
   self.internal_cancel_listing(listing_id.0);
 }
}
```

Recommendation: Add fees for launchpad regardless of listing cancellation or not allow to cancel listing during the phase1 & phase2

Alleviation: The issue is **fixed**.

15. Inadequate validation on AccountId

File: mintable_token_contract.rs

Description:

All the functions in this contract that takes `AccountId` as a parameter does not validate that the AccountId value passed to it actually contains a valid AccountId following the NEAR's account ID rules. As a result, an owner who wishes to burn or mint tokens to another user can mistakenly call the function with a string pointing to an invalid NEAR account ID, resulting in a complete and irreversible loss of control over the contract.

Code

```
#[near_bindgen]
impl Contract {
 #[payable]
 pub fn ft_mint(&mut self, quantity_to_mint: U128, recipient:
AccountId) {
    assert_one_yocto();
    self.only_owner();
    self.token.internal_deposit(&recipient, quantity_to_mint.0);
    self.on_tokens_minted(recipient, quantity_to_mint.0);
 }
}
```

In the context of the mint function, the owner can mint tokens to an invalid account ID.

Note

This vulnerability is present in all the functions of the following contracts which takes accountId as a parameter

- `modified_contract_standards.rs`
- `staking.rs`
- `token_contract.rs`
- `vesting_contract.rs`

Recommendation:

The functions should check that the passing argument is in the form of correct AccountID. Update near-sdk version to 4.0.0+ or use env::is_valid_account_id from near-sdk for validating accountId.

Alleviation: The issue is **fixed** by the auditors recommendation.

Informatory issues and Optimizations

16. Improper validation of deploy address

File: token_launcher/src/actions/deploy.rs

Description:

In `deploy_new_contract` function there is a check that deploy_address should not exceed the maximum size allowed by NEAR is 64 characters but the assertion is not implemented correctly because it limits the address to always be less than 64 characters. The correct logic should be less than and equal to 64 characters.

Code

```
#[payable]
 pub fn deploy_new_contract(
 ...
 ) {
  ...
   //verify if contract size does not exceed max allowed on near
   assert!(
     deploy_address.len() < 64, //here it should <=64
     "{}{}",
     ERR_102,
     env::current_account_id()
   );
```

Recommendation: verify if contract size does not exceed max allowed on near

```
assert!(
    deploy_address.len() <= 64
```

Alleviation: The issue is **fixed** by the auditors' recommendation.

17. Inadequate checks on unregister storage

File: launchpad/src/actions/storage_impl.rs

Description:

Project owners can `unregister_storage` . There is a check `!account_deposit.is_listing_owner`. When a new Listing is created, `investor's is_listing_owner` is never set to true.

Code

```
#[allow(unused_variables)]
 #[payable]
 fn storage_unregister(&mut self, force: Option<bool>) -> bool {
   assert_one_yocto();
   let account_id = env::predecessor_account_id();
   if let Some(account_deposit) = self.internal_get_investor(&account_id) {
     // TODO: figure out force option logic.
     assert!(account_deposit.allocation_count.is_empty(), "{}", ERR_203);
     assert!(!account_deposit.is_listing_owner, "{}", ERR_210);
     self.investors.remove(&account_id);

Promise::new(account_id.clone()).transfer(account_deposit.storage_deposit);
     true
   } else {
     false
```

Recommendation: Update `is_listing_owner` field of investor

Alleviation: The issue is **fixed** by updating the `is_listing_owner` value to `true` in `create_new_listing` function.

18. Optimization in withdraw

File: locked_token/src/actions/user.rs

Description:

In the `withdraw_locked_token` function there is a variable called `value_to_withdraw` which saves the value the user can withdraw. But that value can be 0, which eventually leads to the loss of users' gas fees.

Code

```
 #[payable]
 pub fn withdraw_locked_tokens(&mut self, vesting_id: U64) -> Promise {
...

   let value_to_withdraw =
vesting.withdraw_available(env::block_timestamp());
   vesting_vector.replace(vesting_id, &vesting);

...

   ext_token_contract::ext(self.contract_config.get().unwrap().base_token)
     .with_static_gas(FT_TRANSFER_GAS)
     .with_attached_deposit(1)
     .ft_transfer(
       account_id.to_string(),
       U128(value_to_withdraw), //here it can be 0
       "locked token withdraw".to_string(),
     )
     ...
}
```

Recommendation:

Assertion on this variable can save users a lot of gas and avoid entire transaction execution.

Alleviation: The optimization recommended by the auditors is **implemented.**

19. Standard Conformity and Inconsistencies in the requirements

Description:
In the documentation it is saying that `program_token` and `contract_token` can be the same tokens. But in the current implementation, when the collection owner will try to make one of the `contract_token` as `program_token` the contract will stop it and throw an error.

Documentation also says that the `contract_token` in `contract_treasury` can only be transferred by owner but in the code the guardian is able to transfer the `contract_token` into the collection treasury and so on.

Recommendation: The code should be according to the requirements and documentation.

Alleviation: The issue is **Fixed** by replacing the only guardian check with only owner check.

20. Incomplete Functions in the Contracts

Description:
The contracts contain many incomplete functions and contracts, many of these contain core functionality of the protocol. These functions should be completed for the protocol to perform as intended.
Some of these incomplete functions are the following:

1. `add_contract_token in nft_staking/src/actions/owner.rs`
2. `remove_contract_token in nft_staking/src/actions/owner.rs`

3. `storage_unregister` in `nft_staking/src/actions/storage_impl.rs`
4. `new` in `nft_staking/src/models/staking_program.rs`

There are also some missing checks where anyone can deposit `contract_tokens` to `contract_treasury`, instead this should be restricted to partners.

Recommendation: These functions should be completed as they are defined in the documentation.

Dev's Response for `add_contract_token`:

This is an intentional technical debt. Currently there's no way to delete a staking program, so if we were to block the addition of a new contract token while it was being used as a program token, it would result in the impossibility of ever using any token which was at any point in time a program token, as a contract token.

It's likely this whole edge-case will remain irrelevant forever, as there's little reason to make a token a contract token, even more so a program token, as it likely isn't even owned by the contract owner.

Note that in the current contract behavior, what happens is that the collection owner authorization always has precedence over the guardians/owner. Meaning, if for some reason, in a given staking program, a program token is also a contract token, its role as a program token supersedes its role as a contract token, and only the collection owner may operate on it. We found this to be a reasonable trade-off with little downsides to an unlikely problem with a costly solution.

Alleviation: The issue is **partially fixed.** The auditors agree with the devs' explanation for not fixing 1 and 4.

21. Redundant code in `calculate_vested_investor_withdraw` function

File: launchpad/src/listing/mod.rs

Description: In the `calculate_vested_investor_withdraw` function, the same logic is checked twice which raises redundancy in the code. As mentioned in the below code the condition `timestamp >= self.cliff_timestamp` is checked twice

Code

```
pub fn calculate_vested_investor_withdraw(&self, allocations: u64,
timestamp: u64) -> u128 {
   let allocations = allocations as u128;
   let initial_release =
     ((self.token_allocation_size * self.fraction_instant_release) /
FRACTION_BASE) * allocations;
   let cliff_release =
     ((self.token_allocation_size * self.fraction_cliff_release) /
FRACTION_BASE) * allocations;
   let final_release = self.token_allocation_size * allocations -
initial_release - cliff_release;
   let mut total_release = initial_release;
   if timestamp >= self.cliff_timestamp // redundant
     && timestamp < self.end_cliff_timestamp
     && timestamp >= self.cliff_timestamp //here redundant
   {
...
}
```

Recommendation: It is redundant and should be removed.

Alleviation: The developers implemented the auditor's **recommendation**.

22. Optimization in withdraw_rewards

File: Nft_Staking/src/actions/staker.rs

Description:

The function `withdraw_reward` has a `withdrawn_amount` variable that can be zero. An assertion can be introduced to save user gas

Code

Recommendation: revert early if amount is zero

```
#[payable]
 pub fn withdraw_reward(
   &mut self,
   collection: NFTCollection,
   token_id: FungibleTokenID,
   amount: Option<U128>,
 ) -> Promise {
...
     ext_fungible_token::ext(token_id.clone())
     .with_static_gas(FT_TRANSFER_GAS)
     .with_attached_deposit(1)
     .ft_transfer(caller_id.clone(),
U128(withdrawn_amount), //here can be zero
 None)
```

Recommendation: revert early if amount is zero

Alleviation: The developers implemented the auditor's **recommendation**.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.