hack·a·chain

SHARD LABS

SECURITY AUDIT REPORT

in favor of KUUTAMO

kuutamo

Summary

This report has been prepared for Kuutamo in the source code of the project as well as in project dependencies that are not part of an officially recognized library. The audit has been conducted by combining static and dynamic code analysis with a manual review of the source code by Hack-a-Chain's research team.

The audit process analyses:

1) Adherence to widely recognized best practices and industry standards;
2) Vulnerability to most common attack vectors;
3) Thorough line-by-line review of the code base;
4) Ensuring that contract logic meets the specifications of the project's whitepaper.

The security audit result is composed of different findings, whose vulnerabilities are classified from critical to informational, according to the following impact versus likelihood matrix:

Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low / Informational

Likelihood

After presenting the findings to the client, they are granted a 7 days period to fix the vulnerabilities. This report will specify all vulnerabilities found and whether they were fixed by the team.

Overview

Project Summary

Project Name	Kuutamo Staking Pool
Description	Delegation contract
Platform	NEAR Protocol
Language	Rust
Codebase	https://github.com/Shard-Labs/staking-farm
Initial Commit	23d2342523f40f112ddbd945dbf5a2c564db7049
Commit after fixes	9de6229b08d62856203e3950a50116b4450aa2ec

Audit Summary

First delivery date	08/26/2022
Final delivery date	10/13/2022

Code Inventory

FAC	Factory contract
STK	Staking and farming contract

Findings

ID	Title	Category	Severity	Status
FAC-1	Child contracts data visibility	Usability	Informational	Acknowledged
GEN-2	Migrate to near-sandbox	Testing	Low	Not implemented
GEN-3	Improve testing modules documentation	Testing/Documentation	Informational	Acknowledged
FAC-4	Dynamically calculate MIN_ATTACHED	Usability	Informational	Acknowledged

	_BALANCE			
GEN-5	Assert one yocto near in privileged methods	Permissioning	Medium	Fixed
STK-6	Undocumented distribution algorithm	Documentation	Informational	Acknowledged
STK-7	Migration execution	Bug	High	Fixed
STK-8	Upgrade version	Bug	Low	Fixed
STK-9	Unhandled cross contract error	Usability	Medium	Fixed
STK-10	Farm recycling	Usability	Low	Fixed
STK-11	Separating register and deposit functionality	Usability	Informational	Not implemented
STK-12	Locked withdrawal of rewards	Fund management	Low	Not implemented
STK-13	Full balance lockup	Fund management	Low	Acknowledged
STK-14	Use of public views in change methods	Code organization	Informational	Fixed

*FAC refers to issues in the Factory contract, STK to issues in the stake-farm contract and GEN to issues applicable to both.

Table of contents

Audit report

The audited code consists of 2 separate smart contracts which are upgrades of the staking-farm contract by referencedev, available at https://github.com/referencedev/staking-farm. There is a factory contract responsible for compliance with NEAR foundation's whitelisting process for delegation contracts and a farm contract which is deployed and upgraded by the factory contract.

Sections 1 and 2 perform a thorough overview of the contracts logic, architecture and implementation.

Section 3 provides specific description on each audit recommendation.

The disclaimers section provides the legal terms of responsibility for the auditor.

1 Factory

The factory contract has 2 main responsibilities:

1. Deploying and upgrading its children staking-farm contracts and;
2. Handling the whitelisting process for its children staking-farm contract.

NEAR protocol's token delegation contracts must be included in a whitelist to be able to operate. This whitelist is controlled by the NEAR Foundation through a smart contract. This smart contract has a list of approved factories which may themselves call the whitelist contract to approve their children contracts. To be included in the whitelist, the factory must faithfully implement the NEP-27 standard for its children contracts.

Below follows a review of user roles and application flow, test coverage and a method by method commentary of the smart contract audit.

1.1 User Roles

Users of the factory contract can be divided into 2 categories: **owner** and **regular user**.

Owner

The owner is responsible for administrative tasks, which are approving new delegation contract models that can be deployed or disapproving old models that cannot be used anymore.

This prevents an adversary from loading malicious bytecode into the factory and tricking unaware users to deploy it, leveraging the whitelisting permissions of the factory contract.

Regular User

Regular users can perform any action besides managing the authorized bytecodes in the factory. Users can even load a new bytecode in the factory's state, although it cannot be used by anyone until the owner approves it.

The main feature for regular users is the ability to deploy their own instance of the staking-farm contract through the factory.

1.2 Children contracts' bytecode

The factory contract stores raw WASM bytecode that is used to deploy its children contracts. The contract makes use of the unsafe low-level near_sys library to be able to efficiently process and store the bytecode without running out of gas for the transaction.

The bytecode is stored directly in the contract's storage trie with the sha256 hash of the bytecode in binary format as its key. For the owner to approve the contract for deployment, they must know the bytecode's hash value. The authorization is also stored in the trie with a bytes representation of the base58 representation of the bytecode's hash.

The contract makes extensive use of low-level features which are necessary for the storage and retrieval of the bytecode but are not necessary for the authorization of the bytecode by the owner. In the current setup all users of the contract need to know the hash of the bytecode they want to deploy/authorize to be able to interact with the contract [FAC-1].

It is advisable to move part of this data into the contract in a more visible way, such as allowing view methods to show the available authorized bytecodes, add a description to them and pointing to the repository containing their source code.

The use of unsafe rust also makes the contract very error prone. Fortunately the code is well tested and shows no bugs or vulnerabilities in its handling of raw pointers and low level NEAR libraries.

1.3 Dependencies

The Cargo.toml file for the project highlights all dependencies used in the contract.

```
[package]
name = "staking-factory"
version = "1.0.0"
authors = ["referencedev <goi65io3903d@protonmail.com>"]
edition = "2018"
publish = false

[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
uint = { version = "0.9.0", default-features = false }
near-sdk = { version = "4.0.0-pre.4", features = ["unstable"]  }
near-contract-standards = "4.0.0-pre.4"

[dev-dependencies]
lazy_static = "1.4.0"
quickcheck = "0.9"
quickcheck_macros = "0.9"
log = "0.4"
env_logger = { version = "0.7.1", default-features = false }
near-sdk-sim = "4.0.0-pre.4"
```

The code relies on a small quantity of dependencies for its build which is a good security practice.

All versions are also pinned and the repository contains a Cargo.lock file, which forces builds to use the same package versions even if the dependencies do not pin their sub dependencies. Cargo.lock also contains a checksum of every used library, which prevents adversaries from tampering with the libraries repository and injecting malicious code into the contract.

near-sdk and near-contract-standards libraries are pinned to the “4.0.0-pre.4” version, which is a pre-release version for the already published “4.0.0” version. Even though the used version does not have any known vulnerabilities, the team should consider an upgrade to the most recent version which will have longer term support.

The unstable feature in the near-sdk library is a point of concern. However, the only unstable feature used is the import of near-sys, which is itself a tested and

actively supported library, thus the use of unstable features is not a point of concern for the current contract.

1.4 Testing coverage

Total test coverage is evaluated as 93.27% using the tarpaulin tool for measuring cargo tests coverage. Tarpaulin was created to evaluate unit tests, but it actually evaluates coverage for all tests run by the cargo package manager, which includes the simulation tests for this contract.

Simulation tests are currently run using the near_sdk_sim library. This library implements a simulator of the NEAR and does not run an actual instance of it. For a long time this library has been the preferred method for unit testing code for NEAR smart contracts.

Nevertheless the more recent near sandbox coupled with the workspaces library (available both in Rust and JavaScript) provides a better testing environment since it runs an actual instance of the blockchain locally which is much more similar to the actual production environment. Since the sandbox instance is an actual RPC node, it is also possible to test using the same code base straight on testnet and even mainnet.

Therefore, it is recommended to migrate simulation tests to the newer near-sandbox framework and its supporting libraries for a more comprehensive testing environment [GEN-2].

We find the level of testing satisfactory both in statistical terms and after a manual review of the testing concepts. However testing code does not adhere to the documentation standards used in standard code which makes it hard to read and maintain. Testing code documentation is very important to ensure that future developers and auditors understand which concepts are being tested in each test function so that they can update and evaluate their relevance and completeness. It is thus advisable to improve documentation for the testing modules [GEN-3].

1.5 Smart contract review and issues

This section overviews the functionality of every public method in the contract and links issues found in them whenever present.

new

Initializes the contract by setting the account_id for its owner and for the NEAR foundation owned whitelist contract.

We found no issue with this method.

get_min_attached_balance (view)

Returns the MIN_ATTACHED_BALANCE constant value to the caller. This value is the minimum that must be attached to deploy a contract using the factory.

MIN_ATTACHED_BALANCE is used to prevent end users from wrongly transferring less NEAR than necessary to cover deployment costs. We consider this approach error prone and inefficient since balance requirements can change in further upgrades of the children contracts, it would be better to dynamically calculate the required amount based on the size of the source code in memory for each different bytecode available [FAC-4].

get_number_of_staking_pools_created (view)

returns the number of staking-pool contracts deployed from the factory in its history.

We found no issue with this method.

create_staking_pool

Deploys a new staking pool. Asserts that all information for initializing the new contract is correct, that user deposited more than MIN_ATTACHED_BALANCE NEAR and that the selected bytecode hash is valid.

This will trigger a cross contract call chain that:

1. Creates a batch actions to the new contract
    1. Create new sub account
    2. Deposit NEAR to cover storage costs
    3. Deploy bytecode to the account
    4. Call initializing method (”new”)
2. Creates a callback to the factory
    1. If 1 succeeded, sends cross contract call to whitelist contract to add the newly created account to the whitelist
    2. If 1 failed, returns the deposited NEAR to the caller

We found no issue with this method.

allow_contract

Function can only be called by the owner. It is recommended to assert a deposit of one yocto near to avoid phishing attacks against the owner [GEN-5].

Caller passes the hash of the bytecode they want to authorize.

disallow_contract

Function can only be called by the owner. It is recommended to assert a deposit of one yocto near to avoid phishing attacks against the owner [GEN-5].

Caller passes the hash of the bytecode they want to remove authorization from.

is_contract_allowed (view)

Receives the hash of the bytecode as argument, returns true if this bytecode has been authorized by the owner and false if not.

We found no issue with this method.

store

Stores a new bytecode in the contract's state. In fact it can store any data, but if the data is not a valid WASM contract the create_stake_pool method would break when using the bytecode and revert the transaction.

Can be called by a regular user, but the caller must deposit enough NEAR to pay storage cost of the bytecode.

The bytecode is passed as an argument - here it must be passed as a raw argument, not in the regular JSON serialized form that normal methods take.

The method returns the hash of the stored bytecode.

We found no issue with this method.

1.6 User flow

This section showcases the sequence by which methods are called to perform the intended functionality and links issues found in them if present.

Initialize contract

1. The contract account deploys the contract and calls **new** in a batch action;
2. The contract account burns all its access keys;
3. The owner contacts the NEAR Foundation to request the whitelisting of the contract's address in the delegator whitelist contract.

Add new stake-farm bytecode

1. The author of the stake-farm variant calls **store** by passing the bytecode as an argument;
    1. This process can not be performed with the standard NEAR CLI, it can either use the nearcall tool built by the Aurora team or directly encode a RPC call through code;
2. The author contacts the owner of the factory contract, who reviews the contract and reproduces the build to ensure that the deployed bytecode matches the reviewed source code. The owner then calls **allow_contract** with the hash of the bytecode as an argument;
3. In case the new bytecode is an upgrade/patch of a previous version, the owner calls **disallow_contract** with the hash of the previous version, to ensure that from now on only the new version will be used to deploy new stake-farms.

Risks in this flow:

- The owner restricted methods **allow_contract** and **disallow_contract** can theoretically be called using a FunctionCallKey instead of the regular FullAccessKey for the owner account. There is a risk that an online application might maliciously create such a key in a phishing attempt. Even though the owner account will probably only be used in a secure environment, it is better to eliminate this risk by requiring a 1 yoctoNear deposit to call owner restricted methods - these can only be called by a FullAccessKey [GEN-5]

Deploy a stake-farm contract

1. Users must retrieve the hash of the bytecode they want to deploy.
    1. This can be done by accessing previous blocks through an indexing server and finding out which bytecodes where stored and allowed;
    2. To facilitate this step the contract could implement methods that display all available contract codes to the user [FAC-1];

2. User calls **create_staking_pool** and sends an attached deposit greater than or equal to the required amount.

2 Stake-Farm

Stake-farm is a delegation contract. The purpose of the contract is to allow a **validator** - the owner of the contract - to increase their validation stake for NEAR's Proof of Staking validation system.

Regular users who want to earn validation fees without actually running a node and staking the minimum value for validation can simply stake their tokens into the stake-farm contract and become **delegators**. The validator will use the delegated stake to increase their total staked share for block validation and will proportionally share the validation rewards with all stakers (minus a fee).

The contract is called stake-farm because it also allows the distribution of other assets as rewards, besides the validation rewards. The owner of the pool can add rewards in NEP-141 tokens to all stakers, therefore attracting more capital to the delegation pool.

The main novelties of the contract considering the standard reference stake-farm contract are:

1. The ability to NOT auto compound staking rewards and immediately withdraw them
2. The ability to withdraw funds to any account, not only to the account that staked them

Below follows a review of user roles and application flow, test coverage and a method by method commentary of the smart contract audit.

2.1 User roles

The contract has differentiate permissions for 4 different kinds of users:

factory

The factory contract that originally deployed the contract, it is the only account that can reduce the burn fee on rewards for the staking pool.

owner

The owner of the contract has permissions to:

- assign a new owner
- change the staking key used for validation
- change the reward fees charged from delegators
- pause and resume staking
- add or remove authorized_users
- add or remove authorized farm tokens
- stop active farms
- upgrade the contract to a newer version added to the factory

authorized_user

The authorized user can simply add or remove authorized farm tokens to help the owner manage the different rewards given out in farms.

regular user

Any user can:

- deposit, stake and withdraw funds from the staking pool
- claim farm rewards
- Call the ping method to update the contract's data
- Call burn and unstake_burn to execute burn fee functionality

2.2 Staking pools

The contract implements two separate internal pools with different rules for automatically restaking rewards.

InnerStakingPool

Mimics the functioning of the staking pool in the reference implementation. The contract tracks the amount of "shares" owned by each account. Each share represents an ideal fraction of all funds in the pool, whenever a user deposits funds they gain shares representing the proportion of the pool that they own.

This causes a compounding effect, where all rewards received remain in the pool and continue to be staked, all that changes is that the user's previously owned shares are worth more NEAR with time.

Users can redeem their shares at any time, when that happens, their funds get removed from the pool and placed in their internal account. However, they can not withdraw these funds from the contract for the 4 next validation epochs

InnerStakingPoolWithoutRewardsRestaked

A different staking pool in which received rewards are not restaked, they are instead sent to a rewards treasury. All rewards received can be withdrawn after 4 epochs.

In this model the concept of shares is not present, the resources in the pool are divided into:

1. total_staked_balance → the amount of tokens effectively staked
2. total_rewards → total accumulated rewards in the pool
3. total_buffered_rewards → the amount of tokens that are ready for immediate withdraw (4 epochs have passed)

The way rewards are distributed between users is by using a modified version of the scalable reward distribution algorithm (available at https://solmaz.io/2019/02/24/scalable-reward-changing/). This version presents a way to distribute rewards between multiple stakers in a blockchain environment in O(1) complexity - that is, without having to loop through all stakers.

The distribution algorithm keeps track of 4 base variables:

1. total_staked_balance in the pool
    1. This represents the current sum of all funds staked to this pool
2. rewards_per_token
    1. This represents the historical rewards distributed to the pool
    2. Calculated at every distribution event by adding rewards / total_staked_balance

3. stake_shares
    1. Tracked at the Account level
    2. Represents how many tokens each user has in the pool
4. reward_tally
    1. Tracked at the account level
    2. Dynamically adjusted value to track the distribution of tokens
    3. can be any positive or negative amount

Deposit/withdraw stake events

At any point in time, when a user joins the pool, they deposit N tokens into it:

- total_staked_balance is increased by N;
- rewards_per_token remains constant;
- stake_shares for the user's account is increased by N;
- reward_tally for the user's account is increased by N * rewards_per_token;

At any point in time, when a user withdraws N tokens from their stake in the pool:

- total_staked_balance is decreased by N;
- rewards_per_token remains constant;
- stake_shares for the user's account is decreased by N;
- reward_tally for the user's account is decreased by N * rewards_per_token;

Reward distribution event

Whenever the contract distributes N rewards for its stakers, all that must be done is:

- rewards_per_token is increased by N / total_staked_balance;

Rewards withdrawing

Since the reward distribution event does not directly send each staker their share of the rewards, they must be claimed individually by each staker through a withdraw rewards action, which:

- computes available reward as: stake_shares * rewards_per_token - reward_tally;
- contract releases available reward to user;
- reward_tally is updated to stake_shares * rewards_per_token.

We found that the algorithm used in the **InnerStakingPoolWithoutRewardsRestaked** is implemented in a different non intuitive way.

Instead of only tracking reward_tally, the implementation chooses to track both reward_tally and payed_rewards.

- reward_tally is increased on stakes by staked_amount * rewards_per_token;
- reward_tally is decreased on unstakes by unstaked_amount * rewards_per_token;
- payed_rewards is increased by the amount of paid out rewards at every reward claim;

This choice of implementation must allow reward_tally to go negative since rewards_per_token only increases over time. A deposit that received any rewards would make the reward_tally negative after a withdrawal.

The implementation makes this choice to account for the 4 epochs delay between earning a reward and actually being able to withdraw it. Nevertheless it is recommended to better document the intentions of the novel implementation in the code or to implement the standard algorithm [STK-6] as it might improve code maintainability and readability.

2.3 Farms

Farms work in a very similar mechanic to the InnerStakingPoolWithoutRewardsRestaked pool, but use the original version of the scalable reward distribution algorithm.

Each farm keeps track of:

- total farm amount;
- undistributed amount;
- rewards_per_share (used for shares of InnerStakingPool);
- reward_per_share_for_accounts_not_staking_rewards (used for shares of InnerStakingPoolWithoutRewardsRestaked);
- current reward_round

The farm is created when farm tokens are first deposited into the contract. The creator of the farm selects its start and end dates.

Whenever an internal distribution method is called the farm:

- checks the current reward_round by calculating how many seconds passed between start_date and the current block timestamp
- calculates amount to be distributed based on current reward_round - previous_reward_round
- distributes the new rewards between the InnerStakingPool and InnerStakingPoolWithoutRewardsRestaked according to how many percent of the contract's staked funds are in each
    - Inside each pool divides distributed reward by internal number of shares and adds the result to rewards_per_share/reward_per_share_for_accounts_not_staking_rewards

2.4 Ping method

Both the staking-pool logic described in 2.2 and the farm logic described in 2.3 make use of lazy evaluations to distribute rewards. This is indeed the preferred method for reward distribution in a blockchain environment since it only performs calculations when necessary to distribute funds to a specific user, therefore not consuming large amounts of gas to iterate over all stakers.

When considering the stake pools, it is necessary to evaluate new validation rewards received at every new epoch and distribute them among the pools. This can be done by anyone that calls the ping method. That logic is also executed every time anyone interacts with the contract through staking methods.

2.5 Upgrades

The contract includes a self upgrading logic that works with the support of the factory contract. The owner of the stake-farm contract can call the upgrade method with the hash of the new bytecode they want to upgrade to as an argument.

The contract then fetches the bytecode from the factory and deploys it, substituting the previous bytecode. After deployment, the method **migrate** is called in the newly deployed contract to perform any necessary state migrations.

A bug has been found in the upgrade method, more specifically in the migrate callback that it calls.

Currently the contract implements **migrate_state** at lib.rs as a method for migrating the old state to the new one, but that method is never called in the upgrade workflow. It should be included in the **migrate** method defined in owner.rs [STK-7].

Also, the **migrate** method checks the previous version of the contract to ensure that there are no intermediary versions being skipped and that the migration method is adequate. However, the **migrate** method currently expects “staking-farm:2.0.0” which is the same name as the version being audited. Either the authors forgot to change the version of the code submitted for audit or hardcoded the wrong version as the expected value in the method [STK-8].

2.6 Dependencies

The Cargo.toml file for the project highlights all dependencies used in the contract.

```
[package]
name = "staking-factory"
version = "1.0.0"
authors = ["referencedev <goi65io3903d@protonmail.com>"]
edition = "2018"
publish = false

[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
uint = { version = "0.9.0", default-features = false }
near-sdk = { version = "4.0.0-pre.4", features = ["unstable"]  }
near-contract-standards = "4.0.0-pre.4"

[dev-dependencies]
lazy_static = "1.4.0"
quickcheck = "0.9"
quickcheck_macros = "0.9"
log = "0.4"
env_logger = { version = "0.7.1", default-features = false }
near-sdk-sim = "4.0.0-pre.4"
```

The code relies on a small quantity of dependencies for its build which is a good security practice.

All versions are also pinned and the repository contains a Cargo.lock file, which forces builds to use the same package versions even if the dependencies do not pin their sub dependencies. Cargo.lock also contains a checksum of every used library, which prevents adversaries from tampering with the libraries repository and injecting malicious code into the contract.

near-sdk and near-contract-standards libraries are pinned to the “4.0.0-pre.4” version, which is a pre-release version for the already published “4.0.0” version. Even though the used version does not have any known vulnerabilities, the team should consider an upgrade to the most recent version which will have longer term support.

The unstable feature in the near-sdk library is a point of concern. However, the only unstable feature used is the import of near-sys, which is itself a tested and

actively supported library, thus the use of unstable features is not a point of concern for the current contract.

2.7 Testing coverage

Total test coverage is evaluated as 89.49% using the tarpaulin tool for measuring cargo tests coverage. Tarpaulin was created to evaluate unit tests, but it actually evaluates coverage for all tests run by the cargo package manager, which includes the simulation tests for this contract.

The current test coverage for the owner.rs module is critically low. Tests do not cover any part of the upgrade workflow, which might explain the errors found in such methods.

Test coverage for the farm.rs module is also very low and should be increased.

Even though the view.rs module does not contain state altering calls it could benefit from more comprehensive testing to avoid disruption of applications that rely on them for data feeding.

Simulation tests are currently run using the near_sdk_sim library. This library implements a simulator of the NEAR protocol and does not run an actual instance of it. For a long time this library has been the preferred method for unit testing code for NEAR smart contracts.

Nevertheless the more recent near sandbox coupled with the workspaces library (available both in Rust and JavaScript) provides a better testing environment since it runs an actual instance of the blockchain locally which is much more similar to the actual production environment. Since the sandbox instance is an actual RPC node, it is also possible to test using the same code base straight on testnet and even mainnet.

Therefore, it is recommended to migrate simulation tests to the newer near-sandbox framework and its supporting libraries for a more comprehensive testing environment [GEN-2].

Also, testing code does not adhere to the documentation standards used in standard code which makes it hard to read and maintain. Testing code documentation is very important to ensure that future developers and auditors understand which concepts are being tested in each test function so that they can update and evaluate their relevance and completeness. It is thus advisable to improve documentation for the testing modules [GEN-3].

2.8 Smart contract review and issues

This section presents a review of all public methods of the contract, their functionality and possible vulnerabilities concerning them.

2.8.1 lib.rs

Code is clean, well documented and extensively tested in this module. It contains 3 relevant methods:

new

Initializes the contract with the standard arguments from the factory contract.

migrate_state

Takes the previous state of the contract - considering the reference implementation - and migrates it to the new data format.

There's an issue with this method. It is marked as private so it can only be called by the contract itself, however no other method ever calls it It should be called inside the **migrate** method from the owner.rs module to execute its intended functionality [STK-7].

ping

Checks whether a validation epoch has passed since the last time ping was called.

- If no does nothing;
- If yes distributes rewards among pools and restakes the contract's delegated balance;

We did not find any vulnerabilities in this method

2.8.2 farm.rs

Code is clean and well documented, however this module could use more comprehensive testing. We did not find any vulnerability in it.

claim

Distributes pending rewards to the caller and withdraws rewards earned in a selected token.

If the beneficiary is a smart contract, check that the caller is the owner of the smart contract before executing the functionality.

If the transfer of the tokens fails the internal balance of the user is established.

We did not find any vulnerabilities in this method.

stop_farm

This method can only be called by the owner. It calculates and distributes rewards for the farm and finalizes it.

After a farm is finalized, it does not distribute rewards anymore and all remaining funds are transferred to the owner.

It is recommended to assert a deposit of one yocto near to avoid phishing attacks against the owner.

It is also advisable to create a fallback in case the token transfer fails. Such a feature is already implemented for users and can avoid relevant financial losses by the owner in case of contract misuse [STK-9].

Also, no method exists in the contract to remove one of the active farms to give space to a new one. Even if the farm has been stopped. We recommend adding the functionality in this method to also remove the farm_id from the active_farms Vec [STK-10].

2.8.3 owner

This module is implemented very similarly to the reference code and is well documented. However the code is very poorly tested and needs improvement in its testing coverage to avoid unexpected behavior.

set_owner_id

Changes the owner of the contract to the newly indicated account.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

update_staking_key

Changes the staking key used to stake the contract's funds.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

update_reward_fee_fraction

Changes the reward fee charged by the contract.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

decrease_burn_fee_fraction

Lowers the burn fee charged by the contract.

Can only be called by the factory contract.

This method is not relevant since the current burn fee setup in the factory is 0.

pause_staking

Unstakes the entire staked balance and sets the contract to a paused state, in which funds are not going to be restaked until the paused state is removed.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

resume_staking

Removes the previously set paused state in the contract and restakes the contract's balance.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

add_authorized_user

Adds a new account_id as an authorized user in the contract.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

remove_authorized_user

Adds a new account_id as an authorized user in the contract.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

add_authorized_farm_token

Adds a new NEP-141 token to the list of whitelisted tokens, which allows the creation of a farm for the token.

This method can only be called by the owner or an authorized_user. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

remove_authorized_farm_token

Removes a previously authorized NEP-141 token from the list of whitelisted tokens.

This method can only be called by the owner or an authorized_user. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

upgrade

Fetches a new contract version from the factory contract, redeploys it substituting the current contract and migrates the state of the account to support the new contract's code.

This method can only be called by the owner. However it is recommended to assert a deposit of one yocto near to avoid phishing attacks [GEN-5].

In the present setup the **migrate** callback executed by the method is not effectively performing the migration. It should call the **migrate_state** method defined in lib.rs to execute the intended migration logic [STK-7].

Moreover the **migrate** callback also performs an assertion on the previous version of the contract. Currently the expected version hardcoded in the contract is the same version as the contract itself. This hardcoded value should point to the previous version of the contract instead [STK-8].

2.8.4 stake.rs

This module implements the interface for end users to interact when depositing, withdrawing, staking and unstaking tokens. It is very well documented and tested. However we find that many implementations can perform non intuitive actions that might confuse the end user and highlight such behaviors.

deposit

Deposits the attached NEAR amount into the predecessor's internal balance. If the user does not yet have an internal balance one is created for them in the InnerStakingPool pool.

We find the behavior of this method odd, since it is supposed to be used by accounts registered in the InnerStakingPool but if a user is already registered it will deposit the funds to whichever pool the user is registered in. We suggest separating the functionality of creating a pool registration from other calls to remove ambiguities in this flow [STK-11].

deposit_rewards_not_stake

Deposits the attached NEAR amount into the predecessor's internal balance. If the user does not yet have an internal balance one is created for them in the InnerStakingPoolWithoutRewardsRestaked pool.

We find the behavior of this method odd, since it is supposed to be used by accounts registered in the InnerStakingPoolWithoutRewardsRestaked but if a user is already registered it will deposit the funds to whichever pool the user is registered in. We suggest separating the functionality of creating a pool registration from other calls to remove ambiguities in this flow [11].

deposit_and_stake

Deposits the attached NEAR amount into the predecessor's internal balance. If the user does not yet have an internal balance one is created for them in the InnerStakingPool pool. Immediately stakes the entire deposited balance.

We find the behavior of this method odd, since it is supposed to be used by accounts registered in the InnerStakingPool but if a user is already registered it will deposit the funds to whichever pool the user is registered in. We suggest separating the functionality of creating a pool registration from other calls to remove ambiguities in this flow [STK-11].

deposit_and_stake_rewards_not_stake

Deposits the attached NEAR amount into the predecessor's internal balance. If the user does not yet have an internal balance one is created for them in the InnerStakingPoolWithoutRewardsRestaked pool. Immediately stakes the entire deposited balance.

We find the behavior of this method odd, since it is supposed to be used by accounts registered in the InnerStakingPoolWithoutRewardsRestaked but if a user is already registered it will deposit the funds to whichever pool the user is registered in. We suggest separating the functionality of creating a pool registration from other calls to remove ambiguities in this flow [STK-11].

withdraw_all

Withdraws the predecessor’s entire internal balance.

This action can only be performed if the user did not unstake assets in the previous 4 epochs, in this case funds are locked from any withdrawal for that duration.

We did not find any issues with this method.

withdraw

Withdraws the predecessor’s internal balance by the amount of tokens passed as argument.

This action can only be performed if the user did not unstake assets in the previous 4 epochs, in this case funds are locked from any withdrawal for that duration.

We did not find any issues with this method.

withdraw_rewards

Withdraws rewards received for users registered in the InnerStakingPoolWithoutRewardsRestaked. Users registered in the InnerStakingPool will perform no mutations by calling this method.

There is an edge case in which this method's behavior might confuse the user. If the user unstaked their entire balance this method will not allow them to withdraw any rewards, even if there are available rewards to withdraw.

That happens because when the user has staked tokens the contract calculates their balance based on the already buffered rewards (rewards that have been received prior to the previous 4 epochs), however when the user has no balance the contract tries to withdraw all received rewards and panics in case any of them has not been buffered yet.

We recommend that in case there still are locked rewards the contract falls back to just calculating the buffered rewards and returning them to the user [STK-12].

stake_all

Stakes the entire internal balance of the user.

We did not find any issues with this method.

stake

Stakes the internal user's balance in the amount passed as argument.

We did not find any issues with this method.

unstake_all

Unstakes the entire staked balance of the user and sends it to their internal balance.

After unstaking no amount of the user balance can be withdrawn for the next 4 epochs.

However there is no need to actually lock the entire internal balance for 4 epochs, just the balance that has been unstaked. Though we recognize this makes the code simpler it would be ideal to not lock the user's balance unless it is needed [STK-13].

unstake

Unstakes the staked balance of the user in the amount passed as argument and sends it to their internal balance.

After unstaking no amount of the user balance can be withdrawn for the next 4 epochs.

However there is no need to actually lock the entire internal balance for 4 epochs, just the balance that has been unstaked. Though we recognize this makes the code simpler it would be ideal to not lock the user's balance unless it is needed [STK-13].

unstake_burn

Unstakes shares directed to the burn account.

This method is irrelevant since the factory is implementing a 0 burn fee.

burn

Burns the internal balance of the burn account.

This method is irrelevant since the factory is implementing a 0 burn fee.

2.8.5 token_receiver.rs

The token_receiver.rs module is meant to hold methods called by other contracts when transferring tokens. Even though its test coverage is low, that is not significant considering the simplicity of its code.

ft_on_transfer

Method to receive deposited tokens, only accepts authorized_farm_tokens sent by the owner or authorized_users.

This method can either create a new farm or specify an existing farm to deposit to and update its start_date and end_date.

In case the user tries to create a new farm the contract checks that the maximum quantity of active farms has not yet been reached and panics otherwise.

However no method exists in the contract to remove one of the active farms to give space to a new one. Even if the farm has been stopped by calling **stop_farm**. We recommend the inclusion of this action in the **stop_farm** method [STK-10].

2.8.6 views.rs

The views module generally does not constitute a point of failure for a smart contract since all state information is already public, therefore the view methods can not leak any unintended data that was not already public.

However, if public view methods are called inside the routines of change methods unexpected return values might cause unintended effects in the application. In the current case we found that 3 view methods are used inside change methods:

1. get_total_staked_balance
2. get_account_staked_balance
3. get_account_unstaked_balance

We recommend removing any calls to public view methods from the flow of change methods. This should be implemented by having an internal function that returns the desired value and is called by both the view and change methods [STK-14].

That is a good practice so that view methods can be more iterated upon for front end rendering or data availability purposes without risk of breaking internal contract functionality.

2.9 User flow

This section showcases the sequence by which methods are called to perform the intended functionality and links issues found in them if present.

Initialize contract

1. The factory contract creates the account, deploys the bytecode and calls the **new** method.

Create farm

1. Owner appoints authorized_user;
2. authorized_user calls add_authorized_farm_token;
3. authorized_user calls ft_transfer_call on the authorized_token with msg containing the parameter for the creation of a new farm;

*step 1 is optional, the owner can perform the next steps by themselves.

Issues:

1. If the contract has already reached the limit of active farms, there's no way to end those and create new ones [STK-10].

User stakes receives rewards and unstakes (InnerStakingPool)

1. User calls deposit_and_stake with attached amount;
2. User waits intended time period;
3. User calls unstake_all;
4. User calls get_farms to figure out all possible farms to claim from;
5. User calls claim on all existing farms;
6. User waits for the period of 4 epochs to pass;
7. User calls withdraw all;

User stakes receives rewards and unstakes (InnerStakingPoolWithoutRewardsRestaked)

1. User calls deposit_and_stake_rewards_not_stake with attached amount;
2. User waits intended time period;
3. User calls withdraw_rewards;
4. User calls unstake_all;
5. User calls get_farms to figure out all possible farms to claim from;
6. User calls claim on all existing farms;
7. User waits for the period of 4 epochs to pass;
8. User calls withdraw_rewards;
9. User calls withdraw_all;

Issues:

1. This flow might confuse users since they need to call withdraw_rewards before unstake_all to be able to withdraw their unlocked rewards. If they unstake first, they'll only be able to withdraw after 4 epochs (or have to stake again, call withdraw_rewards and then unstake one more time) [12].

3 Detailed issue report

FAC-1 - Child contracts data visibility

Category	Severity	Location	Status
Usability	Informational	lib.rs	Acknowledged

Description

There is no way to know the available contract model's stored in the contract without indexing previous transactions and checking their success. Whenever the owner wants to approve a new bytecode, they need to know its hash, whenever a user wants to deploy a new stake-farm contract they must know its value.

Recommendation

We advise the implementation of a persistent data structure that stores contract IDs, descriptions and pointers to the hash of the bytecode together with a view call to check this information.

Alleviation

The team acknowledged the issue but chose not to implement fixes immediately as it does not pose a security threat.

GEN-2 - Migrate to near-sandbox

Category	Severity	Location	Status
Testing	Low	entire code base	Not implemented

Description

The code base currently relies on simulation tests performed using the deprecated near-sdk-sim library, which uses a simulator of the NEAR protocol to perform tests. The library has been deprecated in favor of the near-sandbox which runs an actual NEAR node, emulating the real behavior of the contract much more realistically.
Moreover, the move to sandbox based tests also allows tests to be run on testnet and even mainnet since the sandbox shares the API of the real production environment - although it is not possible to use special features such as state patching and time fast-forwarding outside of the sandbox.

Recommendation

We advise the refactoring of simulation tests to use the near-sandbox in combination with a library such as workspaces-rs.

Alleviation

The team chose not to implement the near-sandbox as its features would not allow some precision calculation tests being done with near-sdk-sim.

GEN-3 - Improve testing modules documentation

Category	Severity	Location	Status
Testing/Documentation	Informational	All testing modules	Fixed

Description

The entire code base is very well documented through the use of rust's standard documenting features. Nevertheless, the testing modules do not reproduce the high documentation standards seen in the base code.

Testing code, as much as regular code, needs to be read, understood and refactored through the applications lifecycle being handled by developers and auditors. A clean documentation stating the functionalities being tested, the expected results and the intent behind the code can dramatically increase testing quality and speed up future refactorings and audits.

Recommendation

We advise the inclusion of comprehensive documentation for each individual test created in the test modules. The regular modules can be used as a reference since their documentation is very well written.

Alleviation

The team fixed the issue by adding internal code comments to the testing suite that facilitate reading and updating the code.

FAC-4 - Dynamically calculate MIN_ATTACHED_BALANCE

Category	Severity	Location	Status
Usability	Informational	lib.rs	Acknowledged

Description

The contract has a constant `MIN_ATTACHED_BALANCE` which is used in the `create_staking_pool` method to assert a minimum deposit by the user.
However, the size of the child contract's bytecode is dynamic and the `create_staking_pool` method already calls the `on_staking_pool_create` callback, which returns all transferred funds to the initial caller if the deploy fails.

The only reason for enforcing a specific minimum deposit then is to force the user to deposit more than the minimum necessary value to pay storage costs for the deployment of the contract. In that regard the constant value can break future child contract implementations in case they require more gas.

Recommendation

We advise the calculation of the necessary minimum balance based on the actual size of the bytecode being deployed instead of a hardcoded value.

Alleviation

The team acknowledged the issue but chose not to implement fixes immediately as it does not pose a security threat.

GEN-5 - Assert one yocto near in privileged methods

Category	Severity	Location	Status
Permissioning	Medium	All restricted methods	Fixed

Description

The NEAR protocol allows accounts to have multiple different key pairs. Each key pair can be of type `FullAccess` or `FunctionCall`. Whenever a user interacts with a front end application integrated with the near protocol the website can create a FunctionCall key allowing it to call the contract without the user's explicit authorization.
An adversary could implement a phishing attack by tricking an authorized user into authorizing a FunctionCall key for a contract in which they have management permissions. That key could then be used to effectively hack the smart contract.

To avoid this scenario it is necessary to enforce that privileged methods can only be called through a `FullAccess` key. The only way to enforce that is to require a deposit of NEAR - which can only be authorized by a `FullAccess` key.

Recommendation

We advise the requirement of a deposit worth 1 yocto NEAR to call any privileged method in the contracts.

Alleviation

The team has fixed the issue by adding a 1 yocto NEAR assertion to all privilege checking methods and turning all exposed permissioned methods into payable.

STK-6 - Undocumented distribution algorithm

Category	Severity	Location	Status
Documentation	Medium	staking_pool.rs	Acknowledged

Description

The contract's documentation refers to a version of the scalable reward distribution algorithm (available at https://solmaz.io/2019/02/24/scalable-reward-changing/) in the implementation of `InnerStakingPoolWithoutRewardsRestaked`. However the code implementation differs significantly from the proposed scheme and is not documented anywhere.
A more clear explanation of the code will facilitate code maintenance and auditing in the future.

Recommendation

We advise the documentation of the actual implemented algorithm explaining its functioning and the reason for its different implementation.

Alleviation

The team acknowledged the issue and chose to implement changes in a leter time since it poses no security risk.

STK-7 - Migration execution

Category	Severity	Location	Status
Bug	High	owner.rs and lib.rs	Fixed

Description

The contract implements a self upgrading mechanism that can be called by the owner to fetch the bytecode for the new version from the factory contract and substitute the current bytecode deployed for the new version.
The entire upgrade flow works as expected, however it never performs the data migration necessary to upgrade the contract from the previous version. The upgrade workflow calls the `migrate` method which curiously does not call any migration functionality:

```
#[no_mangle]
pub extern "C" fn migrate() {
    env::setup_panic_hook();
    assert_eq!(
        env::predecessor_account_id(),
        env::current_account_id(),
        "{}",
        ERR_MUST_BE_SELF
    );

    // Check that state version is previous.
    // Will fail migration in the case of trying to skip the versions.
    assert_eq!(
        StakingContract::internal_get_state_version(),
        "staking-farm:2.0.0"
    );
    StakingContract::internal_set_version();
}
```

There is a migrate_state method defined in lib.rs that could be called by the migrate method.

Recommendation

We advise the inclusion of migration logic inside the `migrate` method.

Alleviation

The team fixed the issue by a migrate_state call in the migrate method.

STK-8 - Upgrade version

Category	Severity	Location	Status
Bug	Medium	owner.rs	Fixed

Description

The contract's self upgrading workflow performs a check in the migrate callback to assert that the contract version is compatible with the current upgrade.

```
assert_eq!(
       StakingContract::internal_get_state_version(),
       "staking-farm:2.0.0"
   );
```

The contract must assert that it is being deployed on top of the previous contract, to guarantee that its migration procedure will be able to effectively migrate the contract.
However, the audited contract is requiring its own version. Either the reference to the previous version is wrong or the authors wrongly assigned the contract version in its Cargo.toml file:

```
[package]
name = "staking-farm"
version = "2.0.0"
authors = ["referencedev <goi65io3903d@protonmail.com>"]
edition = "2018"
publish = false
```

Recommendation

We advise the investigation of the issue to understand which of the versions must be altered.

Alleviation

The team fixed the issue by setting the version assertion to 1.1.

STK-9 - Unhandled cross contract error

Category	Severity	Location	Status
Usability	Medium	farm.rs stop_farm	Fixed

Description

The `stop_farm` method implements no handling for a failed token transfer to the contract's owner. Even though that is documented we find it to be a non optimal implementation considering the ease of implementing a fail safe logic and the potential asset losses that it may cause to the owner of the contract.
A similar logic is already implemented for end users in the `claim` method.

Recommendation

We advise the implementation of a failsafe method to avoid permanent fund losses inside `stop_farm`.

Alleviation

The team fixed the issue by adding a callback method that reverts the state alterations in case of a failed token transfer, thus eliminating the possibility of asset losses.

STK-10 - Farm recycling

Category	Severity	Location	Status
Usability	Low	farm.rs	Fixed

Description

The contract can only support `MAX_NUM_ACTIVE_FARMS` farms. This is prudent considering that a large number of farms would trigger a series of serialization and deserialization that could easily overflow NEAR's transaction gas limits.
However, the contract implements no method to remove one of the active farms and substitute it for a new one. The `stop_farm` method does not perform this operation neither does any other method in the contract.

Although it is not a vulnerability in the contract, the entire setup and architecture of the farms gives the impression that the functionality of changing active farms should be supported.

Recommendation

We advise the implementation of a method to remove already stopped farms from the `active_farms` Vec.

Alleviation

The team fixed the issue by adding the `remove_stoped_farm` method, which allows the owner to remove farms that have been stopped.

STK-11 - Separating register and deposit functionality

Category	Severity	Location	Status
Usability	Informational	stake.rs	Not implemented

Description

The contract supports two different staking pools `InnerStakingPool` and `InnerStakingPoolWithoutRewardsRestaked`. Each user can only be registered in one of the pools.
To segregate the pools in the API, the contract exposes `deposit` and `deposit_and_stake` as methods to deposit assets in the `InnerStakingPool` and `deposit_rewards_not_stake` and `deposit_and_stake_rewards_not_stake` to deposit assets in the `InnerStakingPoolWithoutRewardsRestaked`.

However, we find the behavior of these methods very odd. They will only enforce the pool type if the user is not registered yet (their first deposit). After the user has been registered they can call any of the methods and they'll all behave the same way, depositing the funds into the pool that the user is registered to.

There are two different functionalities being handled by these methods that should be split. There should be a method for the user to choose which pool to register to and a deposit/deposit_stake method that will deposit to their inner account in the registered pool. This approach would better encapsulate different functionality in different methods and be easier to explain to an end user.

Recommendation

We advise the implementation of a different method for user registration and to make the deposit methods totally pool type agnostic.

Alleviation

The team chose not to alter the code since wallet applications are already built with a standard interface to delegation contracts and the team wants to be able to have a seamless integration with such tools.

STK-12 - Locked withdrawal of rewards

Category	Severity	Location	Status
Fund management	Low	stake.rs	Not implemented

Description

All staked funds are subject to a 3 epoch lock after being unstaked - the contract actually enforces a 4 epoch lock because of NEAR's promise flow. In the `InnerStakingPoolWithoutRewardsRestaked` all rewards earned are immediately unstaked, but are still subject to the 4 epochs lock.

The `withdraw_rewards` method behaves in 2 different ways depending on the current staked balance of the user:

- **If the user has no staked balance**, the method checks whether 4 epochs have passed since the last reward was received, only allowing withdrawals after the entire rewards balance is unlocked;
- **If the user still has staked balance**, the method evaluates which rewards are already unlocked and sends them to the user, keeping the locked rewards in the contract.

We find this behavior unintuitive, 2 different action flows by the user will result in different outcomes.

- If a user first withdraws their rewards and later unstakes, they end up with all unlocked rewards in their wallet;
- However, if they unstake first and then withdraw their rewards their second call panics and they end up with no rewards withdrawn.

Recommendation

We advise the implementation of a fallback in case the user has no staked balance but not all funds are unlocked, so that the call behaves the same way it would if the user still had funds staked.

Alleviation

The team chose not to implement any changes since it is an edge case that poses no security threat.

STK-13 - Full balance lockup

Category	Severity	Location	Status
Fund management	Low	stake.rs	Acknowledged

Description

All staked funds are subject to a 3 epoch lock after being unstaked - the contract actually enforces a 4 epoch lock because of NEAR's promise flow.
The contract enforces this lock by not enabling any withdrawals from the user's inner account if they have unstaked any funds in the previous 4 epochs.

```
assert!(
            account.unstaked_available_epoch_height <= env::epoch_height(),
            "The unstaked balance is not yet available due to unstaking delay"
        );
```

However, the lockup is only necessary for the funds that have been unstaked. In case the user had any extra funds sitting in their internal balance those funds are also locked for 4 epochs in the current implementation.

Recommendation

We advise a refactor in the withdrawal logic to allow the withdrawal of already unlocked funds even if a part of the funds is still locked.

Alleviation

The team acknowledged the issue but chose to implement changes in a later project as it poses no security risk.

STK-14 - Use of public view in change methods

Category	Severity	Location	Status
Fund management	Low	stake.rs	Fixed

Description

The views module generally does not constitute a point of failure for a smart contract since all state information is already public, therefore the view methods can not leak any unintended data that was not already public.

However, if public view methods are called inside the routines of change methods unexpected return values might cause unintended effects in the application. In the current case we found that 3 view methods are used inside change methods:

1. get_total_staked_balance
2. get_account_staked_balance
3. get_account_unstaked_balance

It is a good practice to never utilize view public view methods inside the code's logic so that view methods can be more iterated upon for front end rendering or data availability purposes without risk of breaking internal contract functionality.

Recommendation

We recommend removing any calls to public view methods from the flow of change methods. This should be implemented by having an internal function that returns the desired value and is called by both the view and change methods.

Alleviation

The team fixed the issue by implementing internal view methods.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Hack-a-Chain's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Hack-a-Chain to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intended to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Hack-a-Chain's position is that each company and individual are responsible for their own due diligence and continuous security.

Hack-a-Chain's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Hack-a-Chain are subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives,

and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS,OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, HACK-A-CHAIN HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, HACK-A-CHAIN SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, HACK-A-CHAIN MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, HACK-A-CHAIN PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED. WITHOUT LIMITING THE FOREGOING, NEITHER HACK-A-CHAIN NOR ANY OF HACK-A-CHAIN’S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. HACK-A-CHAIN WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER’S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED “AS IS” AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS

AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT HACK-A-CHAIN'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST HACK-A-CHAIN WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF HACK-A-CHAIN CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST HACK-A-CHAIN WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

SECURITY AUDIT CERTIFICATE

Company Name

We, Hack-a-Chain, Blockchain Specialist Software Development and Audit Company, in this act represented by our Chief Technology Officer, João Antônio Schmidt da Veiga, grant this **Security Audit Certificate** in favor of **Kuutamo**, recognizing that they underwent the security audit process and corrected all the issues that have been found in their smart contract.

The full security audit report and it's disclaimer can be found in the following link:

https://github.com/hack-a-chain/security-audits

Devoted to enhancing security in the Blockchain Ecosystem and to provide the best quality service for our clients and the community, we sign this Certificate:

João Antônio Schmidt da Veiga
Chief Technology Officer