2023-02-26

Coinecta

Staking Contracts

π Lanningham

Sundae Labs

Contents

1 - Audit Manifest

Please find below the list of pinned software dependencies and files that were covered by the audit.

Software	Version	Commit
Staking Contracts	`0.0.0`	`7122206784b1ea36214393e7775aadc96928a141`
Aiken Compiler	`aiken v1.0.24-alpha+982eff4`	`982eff449e02c0346e3db66223d983c90cd6ee9c`
SundaeSwap-finance/ aicone	`1.0.0`	`a02ce943a2c76c0e683f327af8ad6f28d8b7cfd4`

Filename	Hash (SHA256)
validators/stake_pool_mint.ak	`17f720076db6a08e6f2dc9f9b1dd681ec7d15178d8e4177a000f41a837873841`
validators/stake_proxy.ak	`3a66082dd1773121b3b1e15b2be84ed3f04e6e49335757d29598abe5d2503c9b`
validators/time_lock.ak	`0c29fd62516c174d44338bc97167f847b4850b97cc345af80a39bd717bee24da`
lib/…/datums.ak	`68d2069017fec9940181c403d96e4ca542119016c4a3849edddba8a42ec56085`
lib/…/stake_nft_mint.ak	`c3c49fc62709a956da8773cc0a937c38d43451a5080696cc65157fdf876babdf`
lib/…/stake_pool.ak	`be685002cc4941cb3c444b759d904b02bc40b2c7c384f8d41e7cf958f9fe0a72`
lib/…/utils.ak	`b3cef511def286bbe5bff520c9c753b0f0cd6de10c1a97487eb31f4f56f2f665`

Parameter	Value
`time_lock_hash`	`f5daa8c5b5fc29f5bdcd18931269f0763ca6a7e85fc83c4d2b0117d5`
`batcher_certificate`	`f9c811825adb28f42d82391b900ca6962fa94a1d51739fbaa52f4b06434e43545f434552544946`

Validator	Method	Hash (Blake2b-224)
stake_pool_mint	mint	`61b3802ce748ed1fdaad2d6c744b19f104285f7d318172a5d4f06a4e`
stake_pool_mint	spend	`61b3802ce748ed1fdaad2d6c744b19f104285f7d318172a5d4f06a4e`
stake_proxy	stake_proxy	`eaeeb6716f41383b1fb53ec0c91d4fbb55aba4f23061b73cdf5d0b62`
time_lock	time_lock	`f5daa8c5b5fc29f5bdcd18931269f0763ca6a7e85fc83c4d2b0117d5`

2 - Specification

The first part of any audit that Sundae Labs performs involves developing a deep and intimate understanding for what the client is trying to accomplish.

We first work with the client to develop a clear high level mission statement capturing the business goals of the project. Then, we translate that into an informal specification, which loosely outlines how to achieve that goal. Finally, we translate that, often with the help of the code they've written, into a detailed specification of how the protocol works.

As we make recommendations and findings, we update this specification.

We present the final version of each of these as part of our audit report.

2.a - High Level Objectives

The chief objective of the Coinecta staking contracts is to allow users to lock tokens for a set period of time, earn rewards for longer lockups, while preserving the liquidity of the users position.

2.b - Informal Specification

- Allow a project to configure a "stake pool", with a specific "reward" token, and a set of tiers called the `reward_settings`.
  - These settings include a list of `(percentage, time_period)` tiers.
  - Users who lock for at least `time_period` gain a bonus `percentage` of the `reward` token
  - This `percentage` is applied to the quantity of reward token itself.
  - The pool also has an "open time", before which no rewards can be claimed.
- Allow users to lock an arbitrary collection of assets for fixed time, from a list of allowed tiers.
  - To avoid contention on the "stake pool", a user can lock their tokens in a "proxy contract"; this can then be locked and mint the token by an automated off-chain process
  - Batching should be a permissioned process, to limit gaming of the system
  - Users can reclaim from the proxy contract if it never gets processed.
- At the time of lockup, disburse some amount of "rewards" from the "stake pool" into the lock alongside the user tokens, according to the `reward_settings` tier and the quantity of locked tokens.
- Mint an NFT that allows the eventual claiming of these locked tokens.
  - For example, if this NFT is sold, the new owner of this NFT will be able to unlock the tokens at the end of the lock period.
  - The NFT is burned as part of the unlock.

  - These NFTs should be CIP-68 compatible to provide a nice user experience, and enable other contracts to depend on the quantity of locked tokens.
    - We should be able to safely unlock multiple positions at once.
- The "stake pool" has an owner that can update the reward settings.
  - If the reward settings are updated, it is acceptable that a small number of pending proxy contracts are no longer processes-able

The following goals were discussed, and are explicitly out of scope:

- Rewarding a separate token from the locked token.
- Rewarding a token for a disparate collection of tokens.
- The ability to "unlock early" and forfeit rewards.
- The ability to split or aggregate time locked positions.

2.c - Detailed

2.c.i - Definitions

- **Reward Token**
  - A token distributed by a **Project Owner** to a **User** in return for locking that same token for a set **Locking Term** .

- **Project Owner**
  - Someone who wants to offer a bonus percentage of a **Reward Token** to a **User** in return for locking that token for a set **Locking Term**
  - "Project Owner" is an informality; in theory, anyone can create the **Stake Pool** , but in practice it will often be the team behind a specific **Reward Token** .

- **Stake Pool**
  - A pot of **Reward Token** in a UTXO locked by the `Staking Validator` with one or more configured **Locking Term** s, created by a **Project Owner** .

- **User**
  - An end user who locks an amount of a particular **Reward Token** for a set **Locking Term** in return for a bonus percentage of the **Reward Token** from the **Stake Pool** . The user receives an **Stake NFT** to unlock the position at the end of the **Locking Term** , which they can transfer or sell to another user.

- **Time Lock**
  - A UTXO containing **Reward Token** s locked for a specific **Locking Term** by a **User** , and unlocked with a specific **Stake NFT** .

- **Locking Term**
  - One entry in the configuration on a **Stake Pool** that defines a length of time and a percentage. A **User** who locks their **Reward Token** for at least this length of time will receive a bonus percentage of the **Reward Token** from the **Stake Pool** .

- **Stake NFT**
  - A non-fungible token that tracks ownership of a specific **Time Lock** position, that can be used by a **User** to unlock the position after the **Locking Term** has elapsed.

- **Stake Proxy**
  - A script the **User** can lock **Reward Token** s into, instead of interacting directly with the **Stake Pool** , to reduce contention. Ultimately processed by a **Sequencer** to open the actual **Time Lock** .

- **Sequencer**
  - An actor who processes **Stake Proxy** positions to open **Time Lock** s on behalf of an end user.

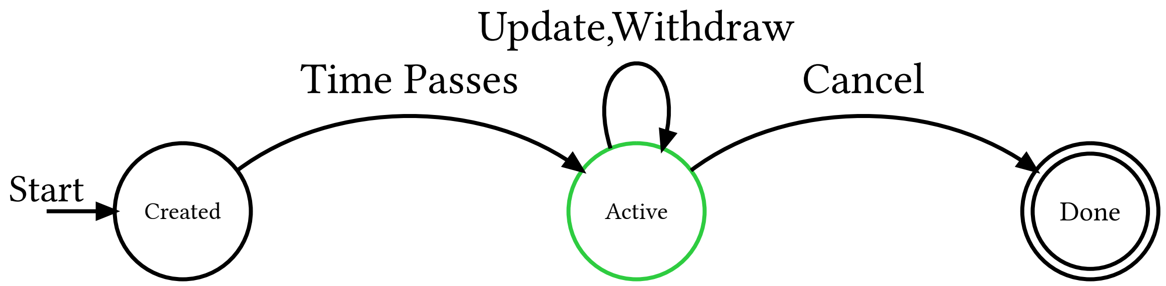
- **Batcher Certificate**
  - A semi-fungible asset passed as a parameter to the **Stake Proxy** to ensure that the **Sequencer** role is only performed by authorized parties

- **Multisig Script**
  - A set of conditions, such as a threshold count of signatures, or an on-chain script that must be run.
  - Provided by the open source library `SundaeSwap-finance/aicone`, which (due to conflict of interest), is not covered by this audit.

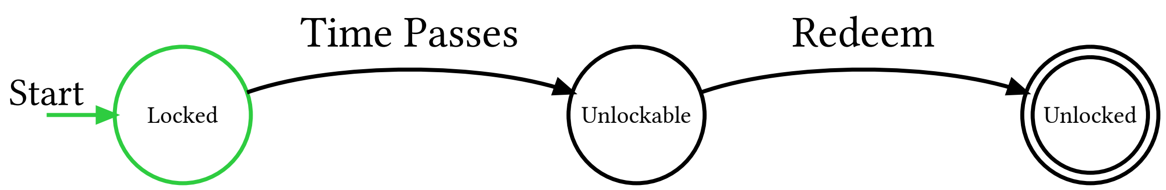
2.d - State Machine

Before digging into each specific transaction, we outline the high-level state machines in the protocol. Colors indicate that a transition can only happen during a state of the same color in a previous diagram.

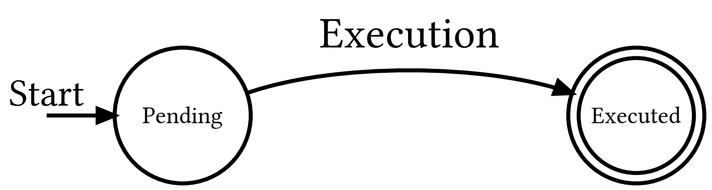
The **Stake Pool** UTXO implements the following state machine:



A **Time Lock** UTXO implements the following state machine:



A **Stake Proxy** UTXO implements the following state machine:

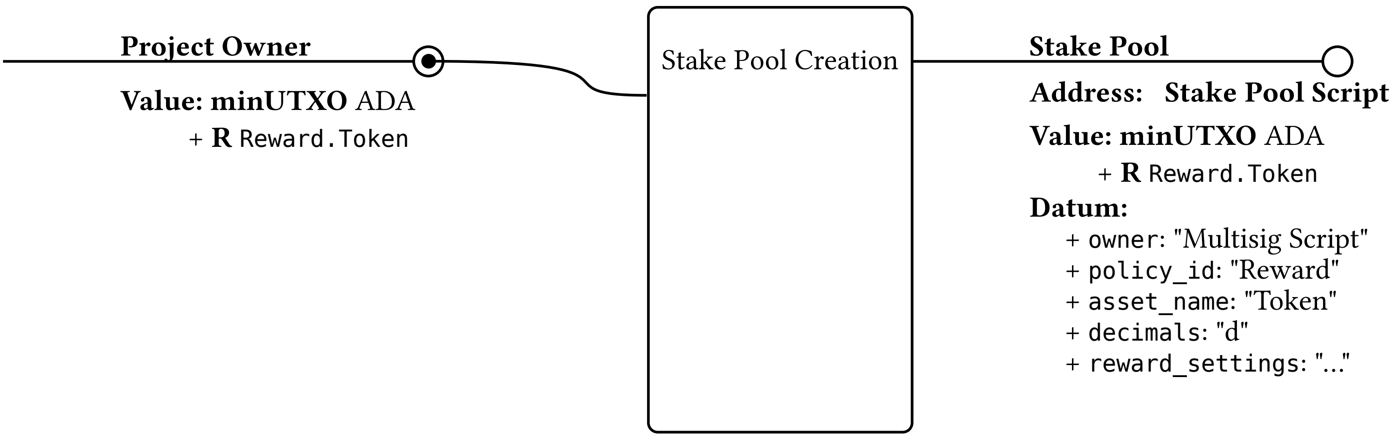


2.d.i - Transactions

There are 7 relevant transaction archetypes. Here is a brief overview of each.

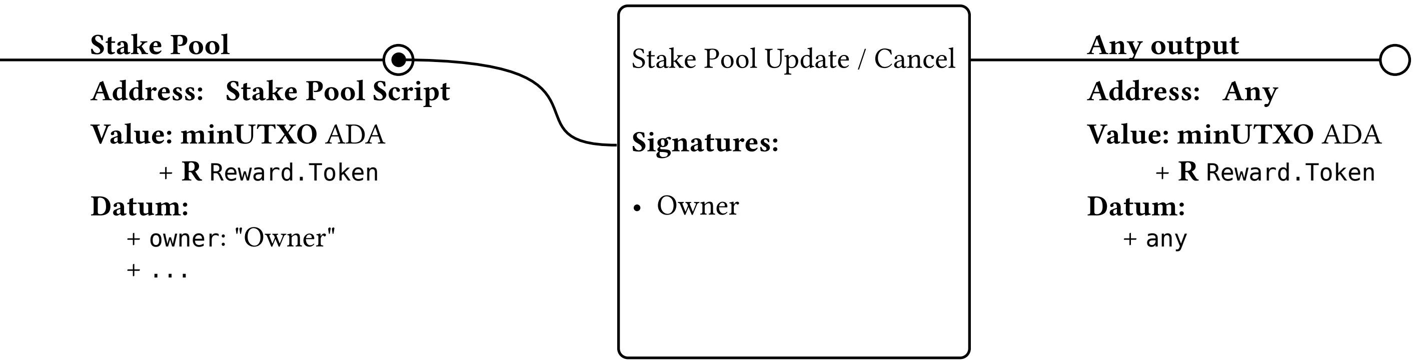
Stake Pool creation

1. This runs no scripts, and consists of the **Project Owner** paying into a UTXO locked by the **Stake Pool** script
2. The validity of the datum is not enforced by any mechanism
3. Nevertheless, a valid datum consists of:
   1. `owner`, the **Project Owner** , a **Multisig Script**
   2. `policy_id`, The Policy ID of the **Reward Token**
   3. `asset_name`, the Asset Name of the **Reward Token**
   4. `reward_settings`, a list of **Locking Term** s, each with the following fields:
      1. `ms_locked`, the minimum number of `Int` milliseconds a position must remain locked
      2. `reward_multiplier`, a `Rational` percentage of rewards earned by a **Time Lock** position at least this long



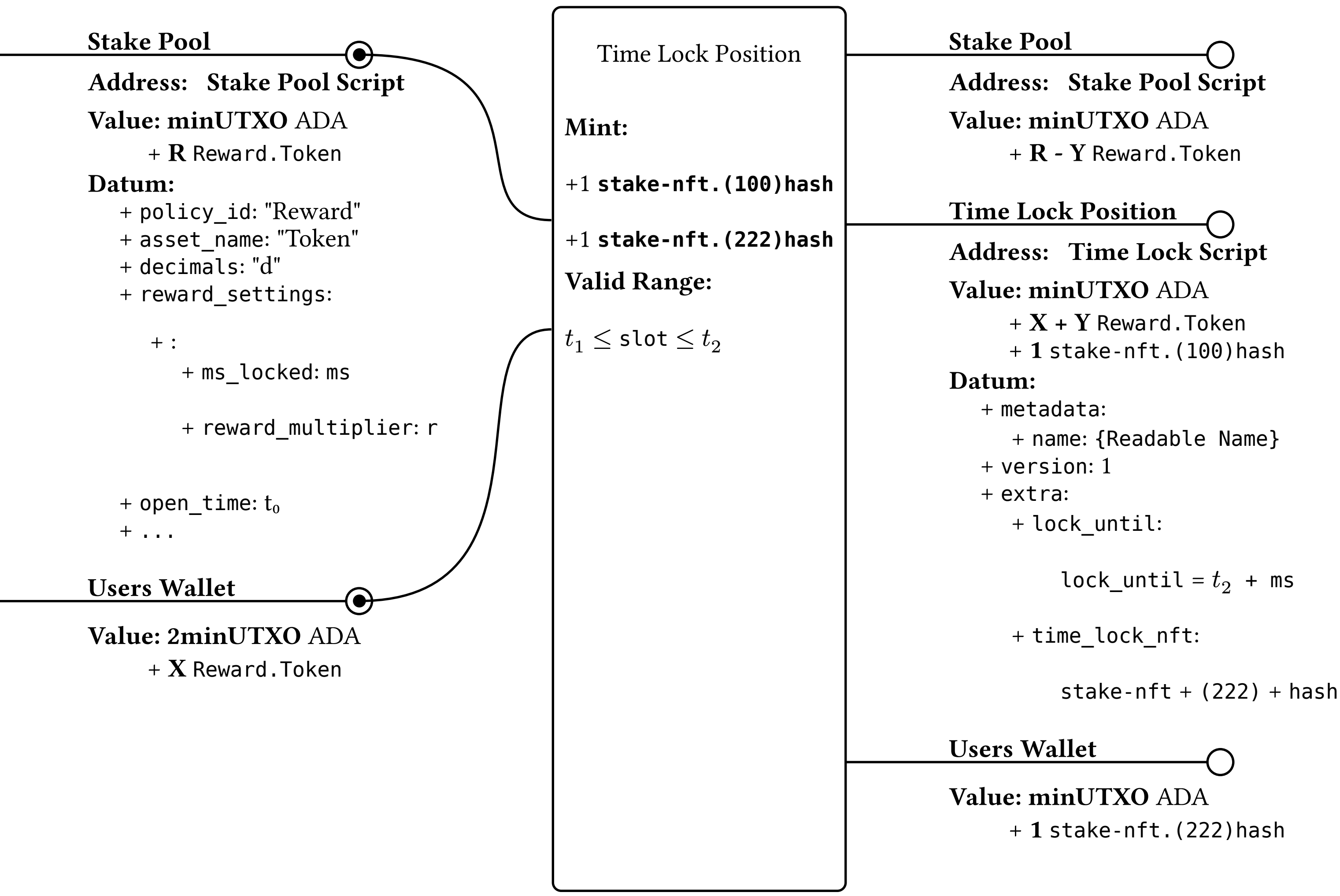
Stake Pool Update or Cancellation

1. This involves spending a **Stake Pool** UTXO in a transaction that satisfies the `owner` **Multisig Script**
2. The assets may be paid back into the script with new settings, or back to the **Project Owner** 's wallet, or any other set of outputs.



Time Lock Position creation

1. A **User** opens a **Time Lock** position directly by spending the **Stake Pool** UTXO and paying some **Reward Token** into the **Time Lock** script address
2. Additionally, the user mints a **Stake NFT** token and the corresponding CIP-68 `(100) reference token`
3. Additionally, depending on the chosen **Locking Term** , the user must pay some **Reward Token** from the **Stake Pool** UTXO into the **Time Lock** output UTXO
4. This transaction cannot happen until the pool is considered "open"



Note:

Transaction lower bound must be after t_0

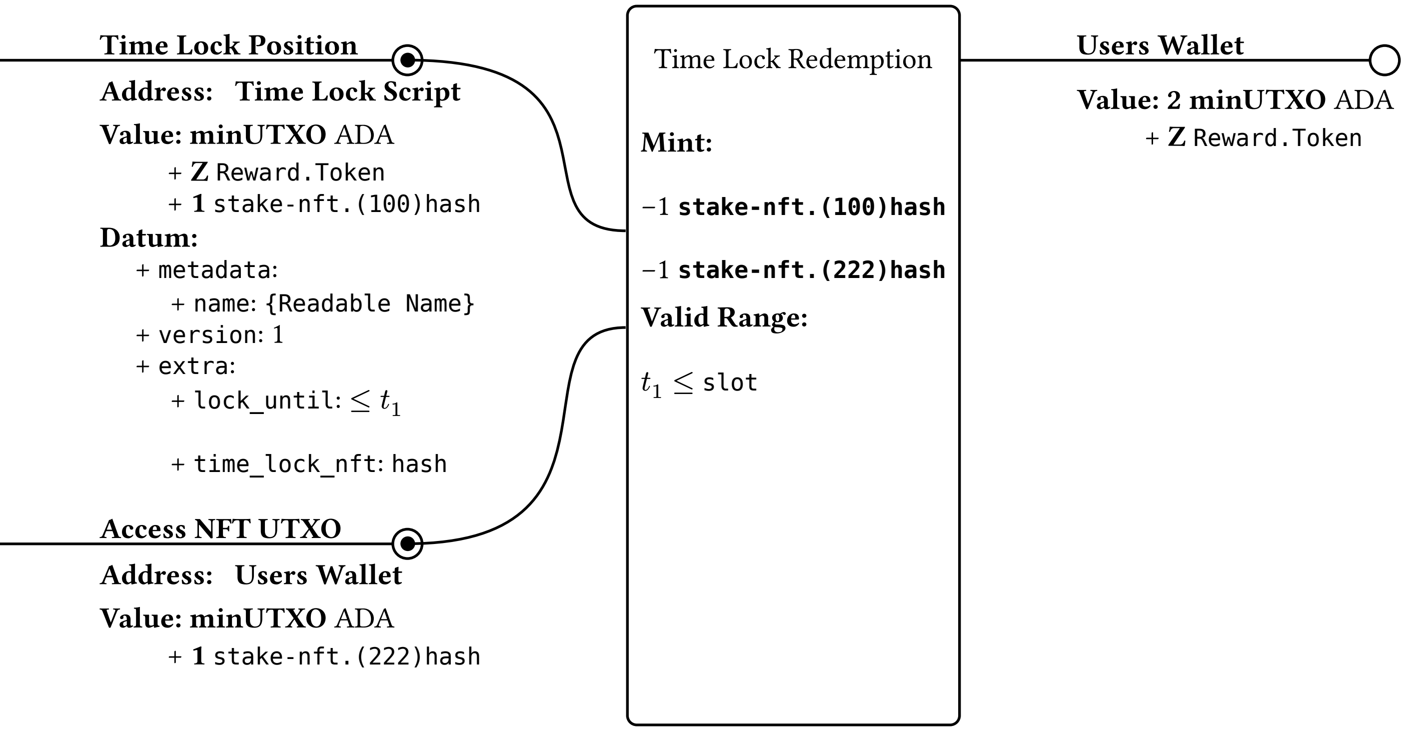
Transaction upper bound and transaction lower bound must be within 1 hour of eachother.

`Y = X * reward_multiplier`

`hash` refers to the first 28-bytes of the `blake2b-256` hash of the **Stake Pool** input TxRef.

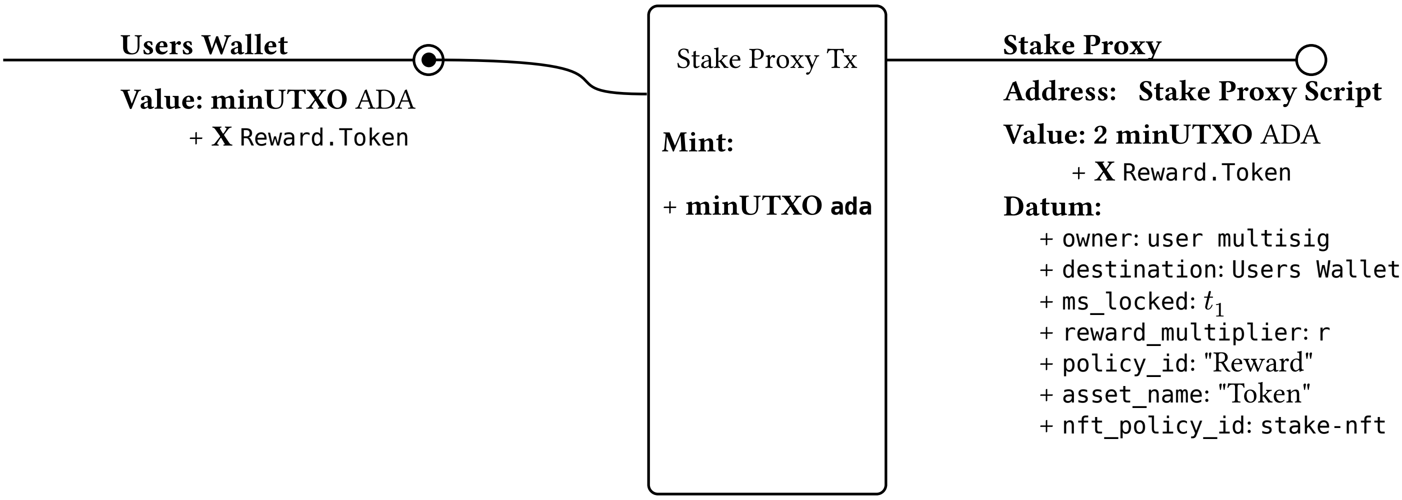
A Time Lock redemption

1. A **User** with a **Stake NFT** token spends the appropriate **Time Lock** UTXO after the **Locking Term** has expired, burns the **Stake NFT** token, and receives the locked **Reward Token** back to their wallet



A Stake Proxy order

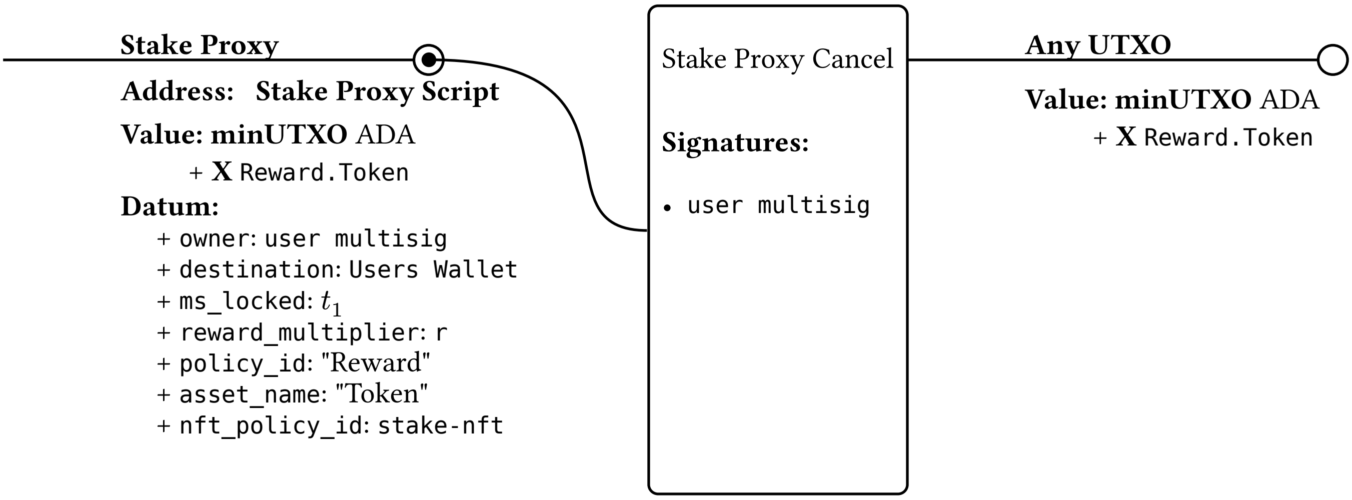
1. A **User** pays some **Reward Token** into a **Stake Proxy** script address, to be processed by a **Sequencer**



Note: The **Stake Proxy** order needs at least enough ADA to create the **Time Lock** position and return the **Stake NFT** to the **User** wallet.

A Stake Proxy Cancellation

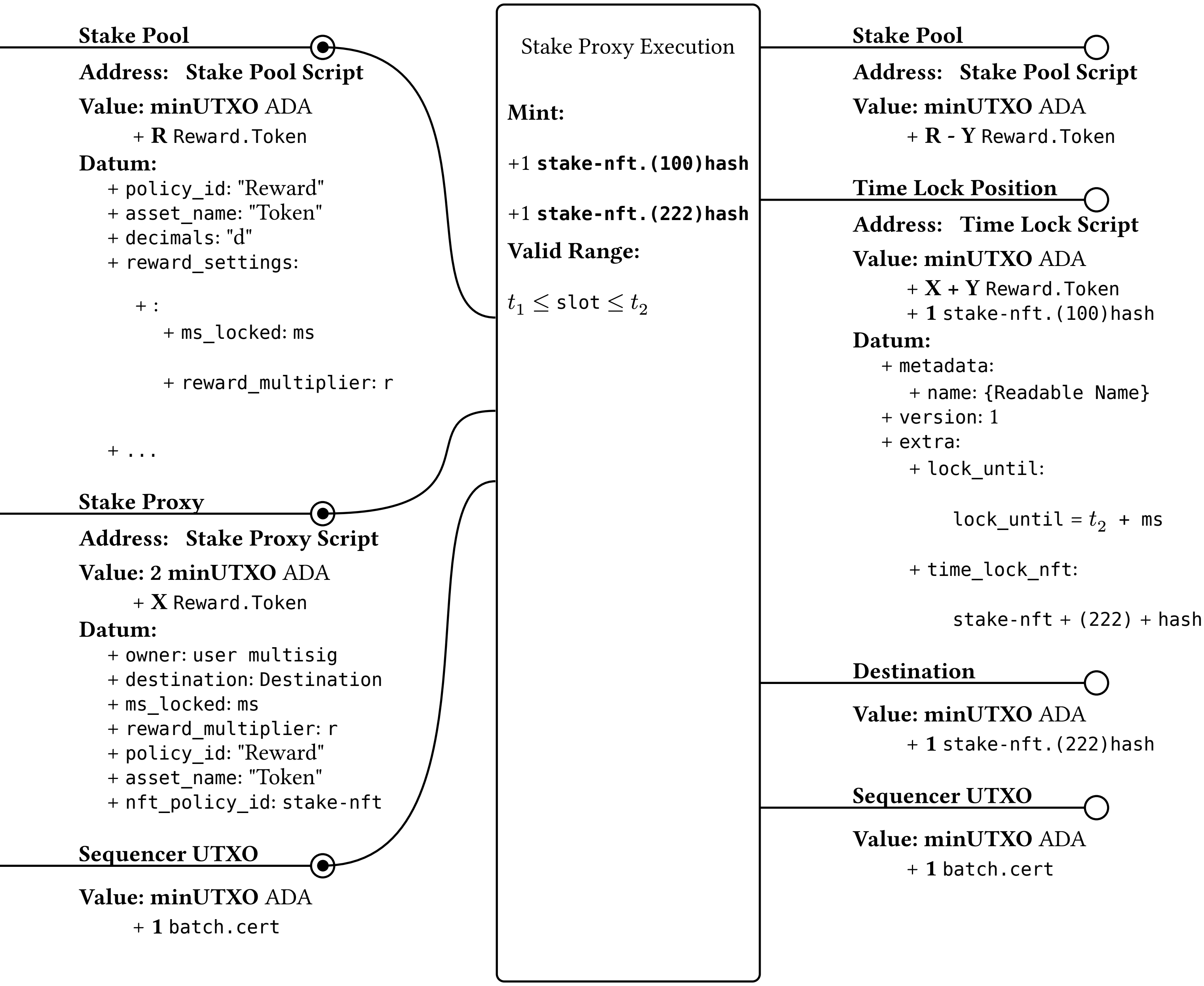
1. The owning **User** spends the **Stake Proxy** to cancel the order before it gets filled
2. The assets can be paid back into the script with new settings, turning the cancellation into an update



Note: The user can also update the order by paying back into the Stake Proxy Script with a new datum.

A <u>Stake Proxy</u> execution

1. A **<u>Sequencer</u>** spends the **<u>Stake Proxy</u>** UTXO, and creates a **<u>Time Lock</u>** UTXO, and mints a **<u>Stake NFT</u>** token paid to the original owning **<u>User</u>**
2. Similar to creating a **<u>Time Lock</u>** position
3. Must also contain a “Batcher Certificate”, an asset with the policy ID from the Stake Proxy validators parameters.



Note:

`Y = X * reward_multiplier`

`hash` refers to the first 28-bytes of the `blake2b-256` hash of the **<u>Stake Pool</u>** input TxRef.

2.d.ii - Validators

In the transactions outlined in the previous section, we made reference to 4 different scripts. Here is a detailed specification of what each of these scripts enforces:

1. **Stake Pool** Spending Script
    1. This script is parameterized by `time_lock_hash`, the script hash of the **Time Lock** script
    2. This script is a spending script, meaning it is run by the node with a datum, a redeemer, and the script context.
    3. The **Stake Pool** UTXO can be spent in one of two cases: to cancel/update the **Stake Pool** , or to create a **Time Lock** position
        - Note: This is done by checking an `or` of each case, rather than using script redeemers
    4. The **Project Owner** can, at any time, cancel the reward program and reclaim all remaining assets, or update any field on the datum if both of the following are satisfied:
        1. The script purpose is `Spend`
        2. The **Stake Pool** UTXO `owner` **Multisig Script** is satisfied
        - Note: Spending the funds to some other address would cancel the reward program
        - Note: Spending the funds back into the script address with an different datum effectively updates the **Stake Pool** settings
    5. The **Stake Pool** UTXO can be spent to create a **Time Lock** position if all of the following are satisfied:
        1. The script purpose is `Spend`
        2. The transaction lower bound is finite, and greater than or equal to the **Stake Pool** `datum.open_time`
        3. There is exactly one input with the **Stake Pool** script payment credential (**CNCT-100**), which is known by finding the transaction input with the same output reference from the script context purpose.
        4. There is exactly one output with the **Stake Pool** script payment credential
        5. The output with the **Stake Pool** script payment credential must have the same address (including staking address) as the **Stake Pool** input.
        6. There is exactly one output with the **Time Lock** script payment credential

7. The datum attached to the singular **Stake Pool** output is an inline datum of type `TimeLockDatum` with the same value as the datum attached to the singular **Stake Pool** input
8. The redeemer contains a `reward_index` `Int` field
9. Let `reward_setting` be the `redeemer.reward_index`th element of the `reward_settings` list in the datum attached to the **Stake Pool** input
10. `reward_setting.reward_multiplier` must be greater than or equal to `0`
11. The transaction validity range has two finite bounds that differ by at most one hour in milliseconds (3600000) (**CNCT-300**)
12. The datum attached to the time lock position has `extra.lock_until` set to a POSIX millisecond timeout that is equal to `transaction.validity_range.upper_bound + reward_setting.ms_locked` (**CNCT-300**)
13. The correct quantity of **Reward Token** is present in the **Time Lock** output. See **CNCT-402** for more details.
    1. Let `withdrawn_amount` be the quantity of `(input_datum.policy_id, input_datum.asset_name)` in the **Stake Pool** input, minus the quantity of the same in the **Stake Pool** output (i.e. the amount removed from the **Stake Pool**)
    2. Let `total_locked_output` be the quantity of `(input_datum.policy_id, input_datum.asset_name)` in the **Time Lock** output
    3. Let `derived_user_locked_output` be `total_locked_output` minus `withdrawn_amount` (i.e. the amount of **Reward Token** provided by the user, rather than withdrawn from the treasury)
    4. Let `expected_total_locked_output` be `dervied_user_locked_output` multiplied by `1 + reward_setting.reward_multiplier` rounded down (i.e. the amount the user provided scaled by the appropriate reward percentage)
    5. Expect `total_locked_output` to equal `expected_total_locked_output`
14. The value paid to the **Stake Pool** is correct, notably the values of other tokens on the UTXO are unchanged.
    1. The value (excluding lovelace) on the single **Stake Pool** input is equal to the value (excluding lovelace) on the single **Stake Pool** output, plus `withdrawn_amount`

    2. The lovelace on the single **Stake Pool** input is less than or equal to the lovelace on the single **Stake Pool** output, to allow adjustments for the minUTXO, or collection of fees.
    - n.b. This might seem unintuitive at first, so pay close attention to which is the input and which is the output.
15. The asset described in the **Time Lock** datum is actually minted. See **CNCT-001** and **CNCT-002** .
16. The asset described in the **Time Lock** datum has the correct policy ID, which is the same policy ID as this spending script hash. See **CNCT-001** and **CNCT-002** .

2. The **Stake NFT** minting script
   1. The **Stake NFT** minting policy shares a script hash with the **Stake Pool** script, so they can be mutually dependent.
   2. Because it is a multi-validator, The **Stake NFT** mint script is also parameterized by `time_lock_hash`, the hash of the **Time Lock** script
   3. The **Stake NFT** minting script can be run under two conditions: either to mint a **Stake NFT** (and it's CIP-68 reference token), or to burn a **Stake NFT** (and it's CIP-68 reference token).
      - Note: this is done via a field on the redeemer.
   4. The redeemer contains:
      1. `stake_pool_index`, the index into the inputs where we should expect to find a UTXO locked by the **Stake Pool** script.
      2. `time_lock_index`, the index into the outputs where we should expect to find an output to the **Time Lock** script.
      3. `mint`, a boolean on whether we are minting or burning the token
   5. The **Stake NFT** can be minted if the following conditions are met:
      1. The redeemer `mint` field is true
      2. The script purpose is `Mint`
      3. The number of entries in the flattened minting value must be exactly 2. That is, exactly two distinct tokens must be minted.
      4. There is exactly one token being minted with a policy ID of **Stake NFT** and an asset name prefixed by the bytes #"000643b0" (a CIP-68 (100) reference token). Let this be the `reference_nft`.

5. There is exactly one token being minted with a policy ID of **Stake NFT** and an asset name prefixed by the bytes #"000de140″ (a CIP-68 (222) NFT). Let this be the `stake_nft`.
6. Let `asset_name` be the asset name of the `reference_nft` with the prefix (the first 4 bytes) dropped.
7. The asset name of the `stake_nft` with the prefix (the first 4 bytes) dropped must equal `asset_name`.
8. The quantity of minted `reference_nft` and `stake_nft` must each be positive `1`.
9. The input at `stake_pool_index` must exist. Let this input be the `stake_pool_input`
10. The `stake_pool_input` must have script payment credential with a script hash of `stake_pool_hash`
11. The output at `time_lock_index` must exist. Let this input be the `time_lock_output`
12. The time_lock_output must have a script payment credential with a script hash of `time_lock_hash`
13. The `stake_pool_input` must have an inline datum of type `StakePoolDatum`. Let this datum be `stake_pool_datum`.
14. The `time_lock_output` must have an inline datum of type `TimeLockDatum`. Let this datum be `time_lock_datum`.
15. Let `raw_amount` be the amount of **Reward Token** (identified by `stake_pool_datum.policy_id` and `stake_pool_datum.asset_name`) in the `time_lock_output`
16. Expect the `time_lock_output` value to have a `quantity_of` the reference NFT (with `own_policy` and the `reference_nft` name) equal to `1`.
17. Let `proper_asset_name` be the first 28 bytes of the `blake2b_256` hash of the `serialised` `stake_pool_input.output_reference`. Since each transaction output can only be spent once, this tx_ref will be unique, and the `blake2b_256` hash will also be unique with extremely high probability. See **CNCT-403** for an analysis.
18. Let `proper_meta_name` be the concatenation of "Stake NFT ", the **Reward Token** asset name, a dash, and a human readable expression of `time_lock_datum.extra.lock_until`
    - Because `proper_meta_name` is for human consumption, the implementation is complex, and the potential attack vector is low, we did not fully audit this code.

- As such, smart contracts looking to consume a **Stake NFT** , tools displaying the value locked by the **Stake NFT** , and users looking to purchase a **Stake NFT** should rely on the amount locked alongside the reference token instead.
- See **CNCT-101** and **CNCT-205** for a discussion of similar issues.

19. `asset_name` must equal `proper_asset_name`
20. The `extra.time_lock_nft` field must equal the concatenation of `own_policy` and the `stake_nft` asset name.
21. The CIP-68 metadata located at `time_lock_output.datum.metadata["name"]` must equal `proper_meta_name`.
22. The CIP-68 metadata located at `time_lock_output.datum.metadata["locked_amount"]` must be equal to the concatenation of:
    1. `[(`
    2. The **Reward Token** policy ID
    3. `,`
    4. The **Reward Token** asset name
    5. `,`
    6. `raw_amount`
    7. `)]`
23.

6. The **Stake NFT** can be burned if the following conditions are met:
   1. The redeemer `mint` field is false
   2. The script purpose is `Mint`
   3. Let `burned` be the flattened transaction mint value
   4. Let `burned_count` be the number of distinct assets in `burned`
   5. Let `reference_nfts` be the list of tuples in `burned` such that the policy ID is `own_policy` and the first four bytes of the asset name equal `000643b0`
   6. Let `stake_nfts` be the list of tuples in `burned` such that the policy ID is `own_policy` and the first four bytes of the asset name equal `000de140`
   7. The correct quantity of tokens must be burned, which means that all of the following conditions are met:

   1. The lengths of `reference_nfts` and `stake_nfts` must be equal.
   2. `burned_count` must be equal to the sum of the lengths of `reference_nfts` and `stake_nfts`; i.e. these must be the only assets we burn.
8. Each of the burns must be performed correctly, which means that for each asset in `reference_nfts`, all of the following must be true:
   1. Let `asset_name` be the `reference_nft` asset name with the first 4 bytes dropped
      - Recall that this corresponds to the hash of the stake pool UTXO spent in the transaction where it was minted, and uniquely identifies the position.
   2. There must be at least one entry in `stake_nfts` such that the asset name without the first 4 bytes is equal to `asset_name`. Let this be the `stake_nft`.
      - Note: This, combined with 2.6.7, and the uniqueness of each of these tokens from the minting policy, implicitly mean that `reference_nfts` and `stake_nfts` are in a 1-1 correspondence, and all must be burned.
   3. The quantity of "minted" `reference_nft` must be equal to `-1`
      - That is, the token must be burnt, rather than minted
   4. The quantity of "minted" `stake_nft` must be equal to `-1`
   5. There must be at least one input locked with the `time_lock_hash` script credential, with the reference NFT in the value. Let this input be the `time_lock_input`
      - This is also implicitly exactly one, because of the uniqueness of the reference NFT
   6. The `time_lock_input` must have an inline datum of type `TimeLockDatum`. Let this datum be `time_lock_datum`
   7. The value of `time_lock_datum.extra.time_lock_nft` must equal the concatenation of `stake_nft.policy_id` and `stake_nft.asset_name`
      - This also implicitly ensures that the `time_lock_nft` is `own_policy_id`
   8. The transaction lower bound (the earliest slot the transaction may appear on chain) must be finite and after or equal to the value of `time_lock_datum.extra.lock_until`.

3. The **Time Lock** script
   1. The **Time Lock** script is not parameterized
   2. The **Time Lock** script is a spending policy, meaning it is run by the node with a datum, a redeemer, and the script context.

3. A **Time Lock** UTXO can be spent only if the following conditions are met:
    1. The script context purpose is `Spend`
    2. The **Stake NFT** specified in the datum is burned.
        1. Specifically, let `policy_id` be the first 28 bytes of `datum.time_lock_nft`
        2. Let `asset_name` be `datum.time_lock_nft` with the first 28 bytes dropped.
        3. Let `minted_quantity` be the quantity of `(policy_id, asset_name)` in the transaction mint field.
        4. `minted_quantity` must be `-1`.
    3. The transaction lower bound (the absolute earliest slot the transaction may appear on chain) must be Finite and after `datum.extra.lock_until`

4. The **Stake Proxy** script
    1. The **Stake Proxy** script is parameterized by `time_lock_hash`, the hash of the **Time Lock** script, and `batcher_certificate`, a ByteArray of the Asset ID that must be present on the transaction
    2. The **Stake Proxy** script is a spending policy, meaning it is run by the node with a datum, a redeemer, and the script context.
    3. A **Stake Proxy** UTXO can be spent only if **either** of the following conditions are met:
        1. It is spent as part of a valid refund transaction, which is valid only if all of the following are satisfied:
            1. The script context purpose must be `Spend`
            2. The **Multisig Script** condition at `datum.owner` must be satisfied
        2. It is spent as part of a lock transaction, which is valid only if all of the following are satisfied:
            1. The script context purpose must be `Spend`. Let the `Spend` output reference be `my_output_reference`
            2. There is at least one output such that the `value` on that output contains an asset with a policy ID equal to the first 28 bytes of `batcher_certificate`, an asset name equal to the remainder of `batcher_certificate`, and a quantity greater than 0
            3. There is exactly one input with the **Stake Proxy** script payment credential
            4. There is exactly one output with the **Time Lock** script payment credential. Let this input be the `time_lock_output`.

5. The datum attached to the `time_lock_output` must be an `InlineDatum` of time `TimeLockDatum`. Let this datum be called `time_lock_datum`.
6. The `time_lock_output` must have a value with exactly 3 assets
   - This is ADA, the **Reward Token** , and the **Stake NFT** CIP-68 reference token, which all must be distinct. See **CNCT-307** .
7. Exactly `datum.lovelace_amount` lovelace must be in the value of the `time_lock_output`
8. The quantity of `(datum.policy_id, datum.asset_name)` in the value of `time_lock_output` must be equal to `datum.asset_amount`
   - If the user lies about this, the **Stake Pool** script will prevent the transaction from being processed.
9. Let `minted_policy_id` be the first 28 bytes of `time_lock_datum.extra.time_lock_nft`
10. Let `minted_asset_name` be `time_lock_datum.extra.time_lock_nft` with the first 28 bytes dropped.
11. `minted_policy_id` must be equal to `datum.nft_policy_id`
    - The intention here is that `datum.nft_policy_id` commits the **Sequencer** to minting the **Stake NFT** The **Stake Pool** script also ensures this is the correct policy.
12. The transaction `mint` field must have a `quantity_of (minted_policy_id, minted_asset_name)` equal to `1`
13. There must be exactly one output such that the address of the output is equal to `datum.destination.address` and the datum of the output is equal to `datum.destination.datum`. Let this output be `destination_output`
14. The lovelace on the **Stake Proxy** input being spent must be exactly 2 ADA (2 million lovelace) greater than the lovelace on `time_lock_output` plus the lovelace on the `destination_output`
    - This corresponds to a 2 ADA fee collected by the batcher, part of which is used to pay the transaction fee. This is not configurable.
15. Let `relevant_outputs` be the the sum of:
    1. the value of the `time_lock_output`
    2. the value of the `destination_output`
    3. the 2 million lovelace fee

16. Let `relevant_inputs` be the sum of:
    1. the value of the transaction mint (which includes the **Stake NFT** and its reference token),
    2. the total value on the **Stake Proxy** input being spent
    3. the **Reward Token** s from the stake pool, calculated as the difference between the tokens locked on the **Time Lock** output and the amount declared in the datum
17. Expect `relevant_inputs` to equal `relevant_outputs`
18. The single **Stake NFT** , defined by `(minted_policy_id, minted_asset_name)`, must be in the `destination_output` value.
19. `datum.reward_multiplier` must not be less than 0.
20. The reward must be calculated correctly. Specifically the quantity of **Reward Token** on the output must be equal to the quantity declared in the datum times 1 plus `datum.reward_multiplier` rounded down.
    - Note that the quantity declared in the datum must be the amount the user supplied implicitly, or **Stake Pool** script would fail, as would 4.3.2.17
21. The time lock output must have a `lock_until` field equal to the transaction upper bound plus `datum.ms_locked` minus 1 hour, and the transaction upper bound and lower bound must be within one hour of eachother.

3 - Findings Summary

ID	Title	Severity	Status
CNCT-000	Multiple Satisfaction on the Stake Proxy Script	Critical	Resolved
CNCT-001	Minting multiple Stake NFTs	Critical	Resolved
CNCT-002	Minting arbitrary Stake NFTs	Critical	Resolved
CNCT-003	Accidental 'always true' minting policy	Critical	Resolved
CNCT-100	Multiple Satisfaction on Stake Pools	Major	Resolved
CNCT-101	Hijacked Stake NFT metadata	Major	Resolved
CNCT-102	Hijacking of staking credential	Major	Resolved
CNCT-200	Lack of Proxy Incentives	Minor	Resolved
CNCT-201	Stake Pool Sniping	Minor	Resolved
CNCT-202	Surplus tokens can be stolen	Minor	Resolved
CNCT-203	Serialization risks in Stake NFT	Minor	Resolved
CNCT-204	Negative Reward Multiplier	Minor	Resolved

CNCT-205	Use of decimals in the stake pool datum can mislead users	Minor	Resolved
CNCT-206	Opening Time Lock Position with no NFT	Minor	Acknowledged
CNCT-207	Extra degree of freedom for sequencer	Minor	Resolved
CNCT-300	Simpler expression of locking window	Info	Acknowledged
CNCT-301	MinUTXO imbalance on proxy script	Info	Resolved
CNCT-302	Lack of partial satisfaction for stake positions	Info	Acknowledged
CNCT-303	Rounding surplus for stake positions	Info	Acknowledged
CNCT-304	Lack of fungibility for **Time Lock** positions	Info	Acknowledged
CNCT-305	Compromises on composability	Info	Resolved
CNCT-306	Changes to MinUTXO might disallow creating new positions	Info	Resolved
CNCT-307	Cannot distribute ADA as a reward	Info	Acknowledged
CNCT-400	Duplicate Reward Settings	Witness	Resolved
CNCT-401	Negative `ms_locked`	Witness	Resolved

CNCT-402	Correctness of reward calculation	Witness	Resolved
CNCT-403	Uniqueness argument of Stake NFTs	Witness	Resolved

CNCT-000 - Multiple Satisfaction on the Stake Proxy Script

Severity	Status	Commit
Critical	Resolved	dfc103cb0e7d91e6ede578063b87a2871ccc02b3

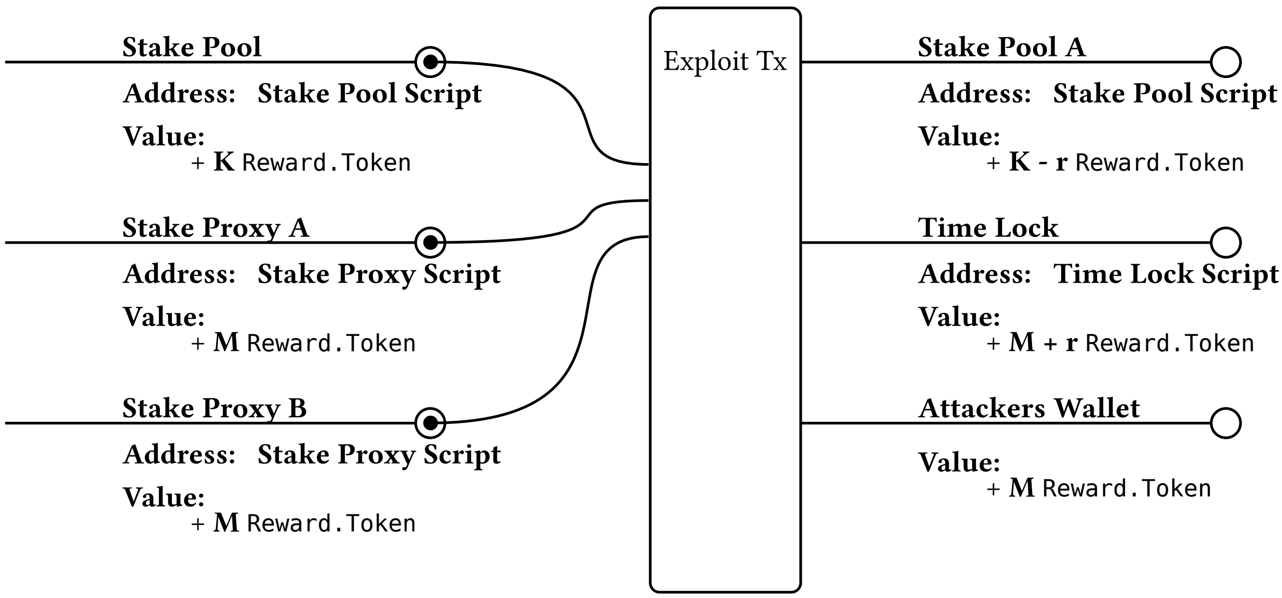
Description

A malicious sequencer can include multiple identical **Stake Proxy** positions in a **Stake Proxy Execution** transaction.

Unlike **CNCT-100**, it is likely to be common that multiple identical stake proxies are created. For example, a large depositor (with $100,000 worth of **Reward Token**) may want to open 100 **Time Lock** positions, each with $1,000 worth of **Reward Token**, so that each token can be sold individually.

Because the **Stake Proxy** script searches for the first **Time Lock** output, and doesn't check for other **Stake Proxy** scripts on the inputs, each one would find a **Time Lock** output that satisfied the conditions in their datum, but the attacker could pocket the difference.

Here is a sketch of what that transaction might look like:



Recommendation

Since Coinecta does not expect high sustained volume, so processing multiple stake pools in a single transaction is not a goal.

In light of that, We recommend the simplest solution of enforcing that only a single input with the **Stake Proxy** payment credential is present in the transaction.

Resolution

This issue was resolved as of commit `dfc103cb0e7d91e6ede578063b87a2871ccc02b3`.

CNCT-001 - Minting multiple Stake NFTs

Severity	Status	Commit
Critical	Resolved	82cc4906616b07d5318342ca643dd38164a7d466

Description

The **Stake NFT** minting script doesn't enforce the quantity of minted **Stake NFT** s. It does enforce that only one **CIP-68 reference token** is minted, and that **only two distinct asset IDs** are minted, but it doesn't check that the actual access token portion only mints one copy of the **Stake NFT** , or that both policy IDs are actually correct.

At least one must be correct, otherwise the **Stake NFT** minting policy wouldn't be running in the first place, but the other token could be any Policy ID.

Imagine, then, that someone mints 2 copies of the **Stake NFT** for a **Time Lock** position worth 1m USD. They could then fairly easily sell one of them on an NFT marketplace for 500k USD (this would be a great deal, normally), and keep the other to unlock the funds themselves.

Additionally, just checking for the quantity doesn't help. Imagine that someone mints the correct `policy_id, asset_name` for the reference token, but an arbitrary `always_succeeds` policy ID for the stake NFT access token itself, using the correct asset name.

The script would allow this:

- There are exactly 2 distinct minted asset IDs
- The asset names are the same
- etc.

But now, someone could later mint a duplicate token and perform a similar attack as above. In this case, the damage would be more limited, as the attacker would be unable to sell the NFT, as the policy ID wouldn't be the expected policy ID.

Recommendation

Enforce that the quantity of both the **Stake NFT** CIP-68 reference tokens and actual access tokens is 1, and that each has a `policy_id` of `own_policy`.

```
when ctx.purpose is {
  Mint(own_policy) -> {
    let minted = from_minted_value(ctx.transaction.mint) |> flatten()
```

```
    // Ensure only 2 distinct tokens are minted
    expect 2 == list.length(minted)
    // Ensure that one of those is the CIP-68 token
    expect [reference_nft] = minted |> list.filter(fn(mt) {
      mt.1st == own_policy && take(mt.2nd, 4) == reference_prefix
    })
    // Ensure that we mint only one reference NFT
    expect 1 == reference_nft.3rd
    // Ensure that one of those is the Stake NFT
    expect [stake_nft] = minted |> list.filter(fn(mt) {
      mt.1st == own_policy && take(mt.2nd, 4) == stake_nft_prefix
    })
    // Ensure that only one Stake NFT is minted
    expect 1 == stake_nft.3rd
    ...
  }
}
```

Resolution

This issue was resolved as of commit `82cc4906616b07d5318342ca643dd38164a7d466`.

CNCT-002 - Minting arbitrary Stake NFTs

Severity	Status	Commit
Critical	Resolved	cde176e014f48a9ef42af815c1058fbd3df1d53c

Description

The **Stake NFT** minting script doesn't enforce that the `time_lock_nft` field in the **Time Lock** datum is set to the correct value. Additionally, the **Stake Pool** script doesn't enforce that the **Stake NFT** is actually minted.

Each of these could be used to open a **Time Lock** position with an arbitrary **Stake NFT** that doesn't enforce the correct conditions. For example, I could trick a user into opening a position without minting the **Stake NFT** . Or, a user could open a **Time Lock** position with an `always_succeed` minting policy, which they could trick someone into buying and then redeem the tokens themselves.

Recommendation

In the **Stake NFT** minting script, enforce that the `time_lock_nft` field in the **Time Lock** datum is set to the correct value, using `own_policy_id` from the script purpose.

In the **Stake Pool** script, check that a single quantity of the asset in the `time_lock_nft` field is actually minted.

Resolution

This issue was resolved as of commit `cde176e014f48a9ef42af815c1058fbd3df1d53c`.

CNCT-003 - Accidental 'always true' minting policy

Severity	Status	Commit
Critical	Resolved	2d43c56dde87f189127991bbe7ff5e0b2ebd37ff

Description

The **Stake NFT** is defined as a validator with three arguments: an ignored `datum`, a redeemer, and the script context.

This accidentally converts the minting policy into an "always succeeds" minting policy. The node applies the actual redeemer to the first argument, the script context to the second argument. This results in a lambda term, which is not an error, and so the transaction succeeds.

See This discussion for more information.

Recommendation

Luckily, the fix in this case is trivial, just remove the extra argument from the minting validator.

Resolution

This issue was resolved as of commit `2d43c56dde87f189127991bbe7ff5e0b2ebd37ff`.

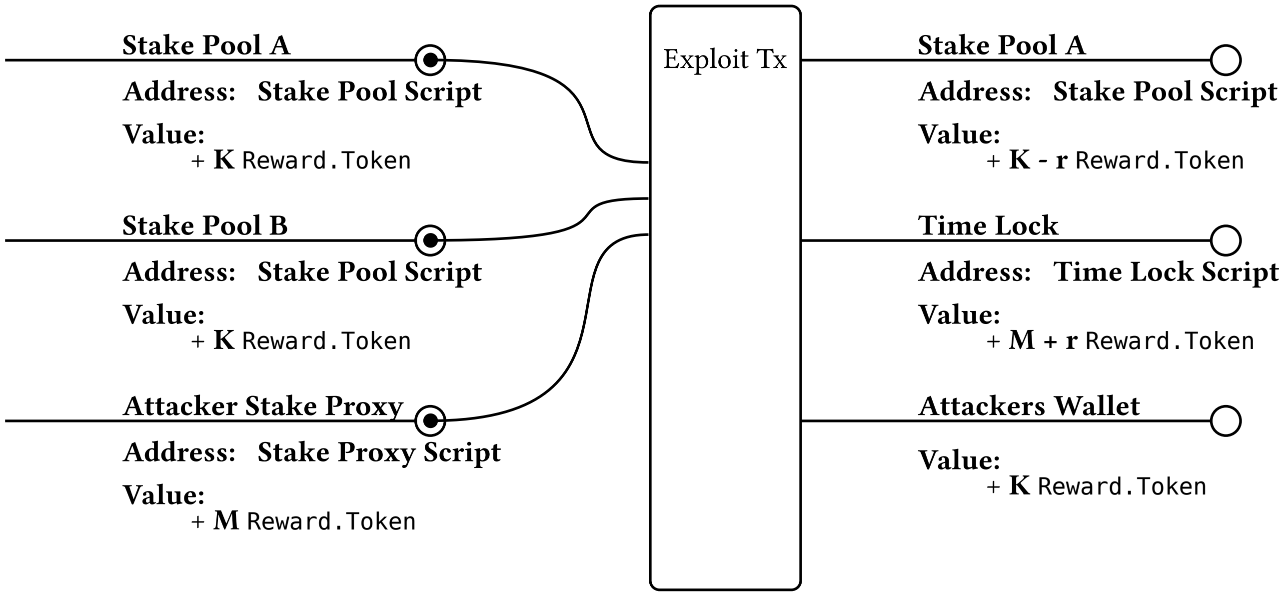
CNCT-100 - Multiple Satisfaction on Stake Pools

Severity	Status	Commit
Major	Resolved	7ba5ec32a8e13e17cd89f11910f7c0bbe0ba2f5b

Description

If there are two **Stake Pool** s with identical datums and quantities, then it is possible to include both of them when opening a **Time Lock** position. Each one will search for the first **Stake Pool** UTXO on the outputs, and compare the values and datums and be satisfied that the **Stake Pool** treasury has been correctly preserved.

This leaves the attacker free to steal the surplus **Reward Token** . Below is a rough sketch of the transaction in such an exploit.



We rated this as Major because, although it is likely rare that there would be two **Stake Pool** s with identical settings (and likely represents a mistake on part of the **Project Owner**), the potential funds at risk are massive.

Recommendation

We recommend enforcing that only a single input with the **Stake Pool** payment credential is present in the transaction.

Resolution

This issue was resolved as of commit `7ba5ec32a8e13e17cd89f11910f7c0bbe0ba2f5b`.

CNCT-101 - Hijacked Stake NFT metadata

Severity	Status	Commit
Major	Resolved	81b4ab4cd40407cacf02817369e7ada5219169ea

Description

The **Stake NFT** minting script doesn't enforce that the CIP-68 reference token is paid to the **Time Lock** script (or even some metadata administrator role).
This would allow the user to pay the reference script to their own wallet with a datum describing a higher value for the **Time Lock** position than is really there. This could fool another user into buying the **Stake NFT** for more than it is worth.

Recommendation

Check that the `quantity_of(time_lock_output.value, own_policy, reference_nft.2nd)` is equal to 1 in the **Stake NFT** minting script.

Resolution

This issue was resolved as of commit `81b4ab4cd40407cacf02817369e7ada5219169ea`.

CNCT-102 - Hijacking of staking credential

Severity	Status	Commit
Major	Resolved	c1aa108f4141afacb10175999e2eae525e527c09

Description

Any user who interacts with the **Stake Pool** to open a **Time Lock** position can change the staking credential attached to the **Stake Pool** UTXO.

This would be largely inconsequential, because the ADA at this UTXO will usually be only the min-UTXO, but given the increasing prevalence of on-chain voting, especially as a proxy for things like Catalyst voting, the impact of this could be dramatic.

Imagine, for example, a popular project that has 10% of their supply in the **Stake Pool** A shrewd attacker could hijack this right before a vote and have an outsized impact on the outcome of the vote.

Recommendation

When spending the stake pool to create a time lock position, enforce that the stake credential is unchanged between the input and the output.

Resolution

This issue was resolved as of commit `c1aa108f4141afacb10175999e2eae525e527c09`.

CNCT-200 - Lack of Proxy Incentives

Severity	Status	Commit
Minor	Resolved	41bad3171036f0047863b85c878b1ba3120dde07

Description

There is no incentive structure around running **Sequencer** s, and the rewards in a **Stake Pool** are distributed on a "first-come, first-served" basis. This could lead to two negative effects:

- The system could stall, with no sequencers running, resulting in the average user being unable to open a **Time Lock** .
- Technical users running sequencers could have an advantage over non-technical users, processing their own **Stake Proxy** requests first.

Recommendation

We recommend that you either:

- Add a small fee that the **Sequencer** can collect from the **Stake Proxy** script when processing it. This will encourage many actors to run sequencers, increasing the probability that users orders are processed sooner.
- Make the operation of a sequencer a permissioned process, and have either Coinecta or the **Stake Pool** owner run the sequencer themselves. The **Stake Pool** owner is the one providing the tokens in the first place, and can cancel it at any time, so it seems appropriate that they can process orders in any order they see fit.

Resolution

This issue was resolved as of commit `41bad3171036f0047863b85c878b1ba3120dde07`, with the comment: We have chosen to make the sequencer permissioned

CNCT-201 - Stake Pool Sniping

Severity	Status	Commit
Minor	Resolved	41bad3171036f0047863b85c878b1ba3120dde07

Description

It is possible to begin processing orders against a **Stake Pool** immediately upon creation. This means that it is susceptible to "snping", whereby an automated bot watches the chain for the opening of the stake pool, and immediately locks enough tokens to claim a majority of the rewards from the pool.

Recommendation

We recommend adding an "open time" to the **Stake Pool** , before which **Stake Proxy** scripts cannot be processed. This will allow projects to set and communicate a clear start time to all users, and allows users to lock their positions in advance of the opening of the pool.

Alternatively, with permissioned sequencing, the **Stake Pool** owner can avoid processing orders before the pool is open. This, however, may be less trusted by users, as there may be a fear that the **Stake Pool** owner will process orders for their inner circle over the community.

Resolution

This issue was resolved as of commit `41bad3171036f0047863b85c878b1ba3120dde07`, with the comment: We have chosen to make the sequencer permissioned

CNCT-202 - Surplus tokens can be stolen

Severity	Status	Commit
Minor	Resolved	0b3a255e1c4bcec06020044dcf90d70f57262fd7

Description

Any surplus tokens included by the **User** on the **Stake Proxy** UTXO can be stolen by the **Sequencer** . This is presumably not a big deal, because the user is not meant to include any extra assets on the **Stake Proxy** UTXO.

This does, however, limit composibility: If the **User** wants to send the **Stake NFT** to a script destination along with some other assets (such as an authenticating for a DAO), then the **Sequencer** can steal those assets, making this kind of composition impossible.

Worse, if someone makes a mistake, not realizing this, it could result in the loss of significant funds.

Recommendation

Enforce that the sum of the values on the **Time Lock** UTXO and the assets sent to the destination are equal to the assets from the **Stake Proxy** input, plus the **Stake NFT** and its reference token.

Resolution

This issue was resolved as of commit `0b3a255e1c4bcec06020044dcf90d70f57262fd7`, with the comment: The implementation here led to an additional finding **CNCT-207**

CNCT-203 - Serialization risks in Stake NFT

Severity	Status	Commit
Minor	Resolved	4dc7f4fafacbfe23aab6c9396264406017056699

Description

The **Stake NFT** generates its name as a concatenation of several fields in an attempt to ensure uniqueness.

However, the serialization of several of these pieces is not fixed-width. For example, the integer timestamp or the stake pool output reference may serialize in CBOR as a different number of bytes. The code, however, assumes that this produces a specific number of bytes, and so hard codes taking the next 17 bytes of the hash of the script UTXO.

This means that if the unix timestamp increases and serializes to take up an additional byte, the creation of all future **Stake NFT** tokens will be broken, as the asset name length would be too large for a cardano asset name.

Recommendation

The attempt to ensure uniqueness is unneccesary, as taking the first 224 bits of a blake-2b 256 hash is already sufficient to ensure uniqueness. We recommend setting the **Stake NFT** asset name to the CIP-68 prefix, followed by the first 28 bytes of the blake-2b 256 hash of the **Stake Pool** input reference. We provide an argument for why this is safe in **CNCT-403** .

Resolution

This issue was resolved as of commit `4dc7f4fafacbfe23aab6c9396264406017056699`.

CNCT-204 - Negative Reward Multiplier

Severity	Status	Commit
Minor	Resolved	1dabb53b6a88344a1d225f68a54e728bdea2d406

Description

A malicious **Project Owner** could set the `reward_multiplier` field to a negative value. In this case, the user would be forced to **pay** to lock up tokens. A user should review their transactions carefully, and be able to avoid this scenario, but may absent-mindedly approve based on trust in the Coinecta staking contracts.

Note that there may be use cases for a `reward_multiplier` of 0, such as if there is some other off-chain incentive offered for holders of **Stake NFT** s.

Recommendation

Enforce that the `reward_multiplier` selected is not less than zero.

```
pub fn add_reward(amount: Int, reward: Rational) -> Int {
  expect Less != rational.compare(reward, rational.zero())
  rational.floor(
    rational.mul(
      rational.from_int(amount),
      rational.add(rational.from_int(1), reward),
    ),
  )
}
```

Resolution

This issue was resolved as of commit `1dabb53b6a88344a1d225f68a54e728bdea2d406`.

CNCT-205 - Use of decimals in the stake pool datum can mislead users

Severity	Status	Commit
Minor	Resolved	ac10b86acdc1a981406c4814859d793c40d73020

Description

A malicious **Project Owner** could set the `decimals` property incorrectly and mislead users. For example, they could set `decimals` to 0 for a reward token with 6 decimals; Then, locking up 1 **Reward Token** would create a **Stake NFT** with CIP-68 metadata that claims to have 1,000,000 **Reward Token**, which could then be sold on an NFT marketplace to an unsuspecting user.

Recommendation

Remove `decimals`, and encode the full `(policy_id, asset_name, amount)` in the CIP-68 metadata. Allow wallets or dApps that plan to display the data do so using the regularly registered metadata they resolve for the `policy_id` and `asset_name`.

Resolution

This issue was resolved as of commit `ac10b86acdc1a981406c4814859d793c40d73020`.

CNCT-206 - Opening Time Lock Position with no NFT

Severity	Status	Commit
Minor	Acknowledged	

Description

The **Stake Pool** script does not enforce that an **Stake NFT** is minted for the **Time Lock** position.
A user could thus open a time-lock position with the incorrect **Stake NFT** specified in the datum.
Depending on what the user specifies, this could have one of two consequences:

- If the user specifies no **Stake NFT** , or a token that cannot be burned, then the position will be locked forever.
- If the user specifies a non-standard **Stake NFT** , then the **Stake NFT** will not have the same policy ID as other **Stake NFT** s in the ecosystem, and may be difficult / impossible to sell.

Recommendation

Neither are particularly detrimental to the Coinecta protocol, and constitute user mistake, but it might be worth enforcing this regardless.
It is slightly awkward to enforce this, as it creates a circular dependency between the scripts.
If you do want to fix this, the two common ways to resolve this are:

- Move the Stake NFT logic into the stake pool script, as a "multi-validator"; in this way, they each have access to eachothers script hashes, because they share the same script hash!
- Store the relevant script hashes in a global datum, locked by an NFT, and include that datum as a reference input.

We recommend the former, as it also allows you to remove the Stake pool parameter from the NFT mint validator.

Resolution

This issue was acknowledged by the project team with the comment: In the end nothing can prevent someone from paying into the time lock address and lock assets forever.We dont think it is worth enforcing in the stake pool validator.

CNCT-207 - Extra degree of freedom for sequencer

Severity	Status	Commit
Minor	Resolved	47b5516c52b62985d69613ebe76c83f990c12c20

Description

Currently, after resolving **CNCT-202** , when executing the **Stake Proxy** script, we enforce the following conditions:

- Exactly three distinct tokens are locked at the time lock output
- At least 2 ADA is locked at the time lock output
- At most 2 ADA is deducted from the stake proxy input to pay the sequencer
- The sum of the time lock output and the destination, minus the sequencer fee, is equal to the stake proxy input

This is mostly safe because:

- There must always be ADA on an output, the stake pool script forces the time lock output to contain **Reward Token** s, and the **Stake NFT** minting script forces the time lock output to contain a CIP-68 reference token for the **Stake NFT** . This means that the tokens included in the time lock output is not a choice the sequencer can make.
- The **Stake NFT** minting script enforces that the reference token has a quantity of 1, and the **Stake Proxy** script enforces the quantity of the **Reward Token** s on the time lock output.

However, this leaves the sequencer free to choose how much surplus ADA is directed to the time lock output vs the destination.

Ultimately this is mostly harmless:

- The sequencer is a permissioned entity
- The ADA still belongs to the user, it's just now locked for their time lock period
- It's unlikely that there is a protocol that requires both the stake NFT and a significant amount of ADA on the other side of the trade, so users are unlikely to include extra ADA regardless

However, if such a protocol arose, and a user locked 10,000 ADA alongside their stake proxy, intending to send it along with the stake NFT to the destination (which may be another script that utilizes that ADA), then the sequencer could lock most of that ADA in the time lock position instead.

Recommendation

If you wish to resolve this, we recommend some variation of:

- Specify the exact value from the input that can be "used" for the time lock output
- Enforce in the stake proxy script that the time lock output is this value, minus the sequencer fee, plus the reward percentage, plus the reference NFT
- Enforce that any remainder is sent to the destination.

Resolution

This issue was resolved as of commit `47b5516c52b62985d69613ebe76c83f990c12c20`.

CNCT-300 - Simpler expression of locking window

Severity	Status	Commit
Info	Acknowledged	

Description

The core logic of the staking contracts is about enforcing that the **Time Lock** position remains locked for the correct duration. This means ensuring that the assets are locked for **at least** the minimum duration offered by the **Stake Pool** , and ensuring that a users assets aren't locked egregiously longer than expected.

The way this is implemented currently is correct, though confusing to follow:

- The transaction upper bound plus the minimum lock duration minus one hour must be less than the timestamp the **Time Lock** is locked until
- The transaction lower bound plus the minimum lock duration plus one hour must be greater than the timestamp the **Time Lock** is locked until

It requires carefully reasoning, or case analysis with pen and paper, to convince yourself that this achieves the objective.

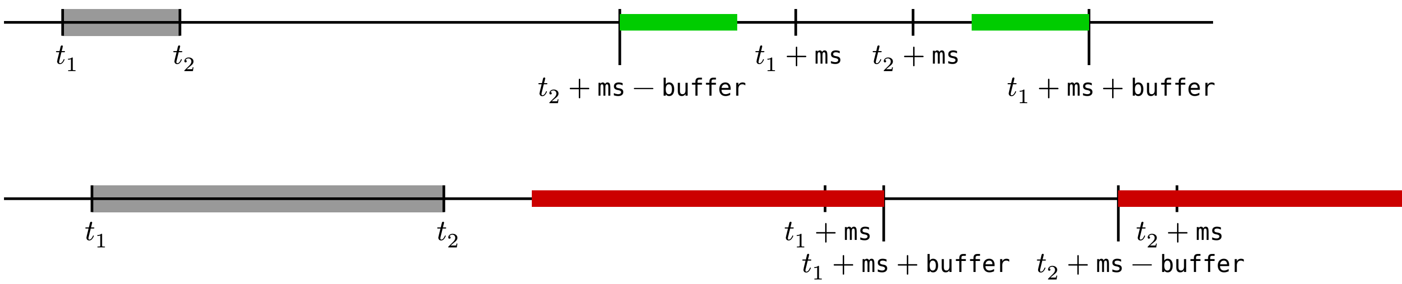


Figure 1.

Two different examples of how `lock_until` is evaluated.

Top: The transaction window is small, and any `lock_until` within the green shaded regions is valid.

Bottom: The transaction window is too large, and there is no `lock_until` that satisfies both conditions.

Recommendation

A simpler way to achieve this goal would be to express the condition as:

- The transaction upper bound and the transaction lower bound must be within one hour of eachother (ensuring at least 1-hour accuracy of the validity range)

- Ensuring that the **Time Lock** is locked until exactly the transaction upper bound plus the minimum lock duration.

This achieves the same goal.

- If the transaction upper bound is set in the past, in an attempt to unlock the position early, the transaction will not be valid in the first place.
- If the transaction upper bound is set further in the future to attempt to unfairly lock a users tokens for longer than need be, either the interval will be longer than an hour, or the transaction lower bound will be in the future, again making the transaction invalid.

This logic is also easier to understand, explain, and ensure correctness.

Resolution

This issue was acknowledged by the project team with the comment: We intend to implement this change.

CNCT-301 - MinUTXO imbalance on proxy script

Severity	Status	Commit
Info	Resolved	0b3a255e1c4bcec06020044dcf90d70f57262fd7

Description

When executing a **Stake Proxy** to produce a **Time Lock** position on behalf of the user, there is an imbalance in the required ADA on the inputs and the outputs. On the inputs, we have the **Stake Pool** UTXO, the **Stake Proxy** UTXO, and a UTXO containing the **Sequencer** ‘s **Batcher Certificate** . Each one of these requires a minimum quantity of ADA (usually around 1 ADA, though it may be more depending on the datum and assets). However, on the outputs, we have a minimum of 4 UTXOs: The **Stake Pool** output, the **Time Lock** position, the **Sequencer** s **Batcher Certificate** , and the **Stake NFT** paid to the destination. This means that a **Stake Proxy** UTXO must contain enough ADA for both of the users outputs. Otherwise, the order may be unprocessable unless the **Sequencer** pays for that minimum UTXO requirement. Calculating what this minimum requirement will be off-chain is subtle and error-prone.

Recommendation

We recommend that you either:

- Ensure that you carefully calculate the minimum required ADA, and ensure (off-chain) that it is included when the **Stake Proxy** order is created
- Pick some practical constant, such as 5 ADA, and enforce that 10 ADA is included in the **Stake Proxy** UTXO, 5 of which is paid to the new **Time Lock** Position, and 5 of which is included with the **Stake NFT** sent to the destination.

Resolution

This issue was resolved as of commit `0b3a255e1c4bcec06020044dcf90d70f57262fd7`.

CNCT-302 - Lack of partial satisfaction for stake positions

Severity	Status	Commit
Info	Acknowledged	

Description

The **Stake Pool** script allows disbursement of a portion of the held **Reward Token** s. If someone locks a large number of **Reward Token** in the **Stake Proxy** script, it may be unable to be satisfied because the amount of **Reward Token** it is entitled to is smaller than the remaining balance in the pool. This may inconvenience some users, as they might expect their order to be partially filled, locking the minimum amount of **Reward Token** that it needs to reclaim the rest of the **Stake Pool** balance, and returning the rest.

Recommendation

If this quality is important to you, we recommend something like the following:

```
let maximum_reward = add_reward(offered_amount, reward_fraction)
let entitled_reward = max(maximum_reward, stake_pool_balance)
let required_lock = calculate_lock_from_reward(entitled_reward)
let surplus = offered_amount - required_lock
```

And then ensure that `required_lock` is locked in the **Time Lock** position, rather than `maximum_reward`. This would also address **CNCT-303**

Resolution

This issue was acknowledged by the project team with the comment: This is not currently a priority

CNCT-303 - Rounding surplus for stake positions

Severity	Status	Commit
Info	Acknowledged	

Description

Currently the protocol rounds in favor of the **Project Owner** that is, a very small amount more **Reward Token** s get locked than would be strictly neccesary to earn the amount of **Reward Token** that get withdrawn from the treasury.

For most assets, which use 6 decimal places and have large supplies, this has no material impact to the user, as it's a few millionths of a token.

However, if you imagine some very low liquidity token, with no allocated decimal places (such as the `XDIAMOND` token), this could be several hundred ADA worth of under-paid or over-locked rewards.

For example, consider `XDIAMOND`. If the **Project Owner** were to offer 10% return for a 3 month lock, and the user locked 10 XDIAMOND, they would receive 1 XDIAMOND bonus. A different user, however, who locked 19 XDIAMOND, would also receive 1 XDIAMOND, despite locking 9 extra tokens.

Recommendation

Similar to **CNCT-302** , we recommend first computing forward to calculate the entitled reward, and then reversing that calculation to compute the required lock amount.

In pseudocode:

```
let entitled_rewards = floor(mul(offered_amount, 1 + reward))
let required_lock = div(entitled_rewards, 1+reward)
let surplus = offered_amount - required_lock
```

And then ensure that `required_lock` is paid to the **Time Lock** position, rather than `entitled_rewards`, and `surplus` is paid to the destination along side the **Stake NFT** .

Resolution

This issue was acknowledged by the project team with the comment: Given that most tokens we plan to launch will have some number of decimal places, and users can calculate the exact amount needed to lock, this is not currently a priority.

CNCT-304 - Lack of fungibility for Time Lock positions

Severity	Status	Commit
Info	Acknowledged	

Description

One of the stated goals of the protocol is to allow the lockup of the underlying **Reward Token** , but allowing these assets to maintain their liquidity and trade on an open market for speculators.

However, there are two related barriers to this goal.

First, users with large positions would need to find a seller for the **entire** position; If this is worth $100,000 USD, it may be difficult to find a buyer for such a large position at once.

Second, speculators are likely to regularly "sweep the floor" when the market price for these positions is below the expected value of the **Reward Token** locked. This means they will likely end up with hundreds of **Stake NFT** positions, making unlocking these positions expensive and difficult.

Recommendation

We recommend adding two new operations to the **Time Lock** script available to the user with the **Stake NFT** : Split and Merge.

Split would allow the user to take one **Time Lock** position and split it into multiple positions, with the **Reward Token** divided arbitrarily between them, and each with the same `lock_until`. This would allow a user to shave off smaller portions of a large position and sell them, increasing their access to this liquidity.

Merge would allow the reverse: given multiple **Time Lock** positions and their **Stake NFT** s, merge both into one **Time Lock** position. For the `lock_until`, you could use the maximum `lock_until` among the inputs, maintaining the original goal of the **Project Owner** of ensuring the tokens remain locked for the minimum duration. The user gives up the time difference, but may be willing to do so for the convenience (especially if all the positions have already unlocked, or are within a day or two of eachother).

Resolution

This issue was acknowledged by the project team with the comment: This is not currently a priority

CNCT-305 - Compromises on composability

Severity	Status	Commit
Info	Resolved	dc1bc70ab0844d78ba05a57a8c11b8646800bc82

Description

Right now, the **Stake Proxy** script enforces that the **Stake NFT** is paid to a wallet address with the payment credential equal to the `owner`.
This limits composibility with othe protocols. For example, perhaps a DAO would like to lock up some **Reward Token** s in return for protocol-owned yield; In this case, the resulting **Stake NFT** is a property of the on-chain DAO smart contract, but the current design would require the DAO elect an intermediary who receives the NFT and trust that they will send it to the DAO smart contract correctly. Similarly, the `owner` of a **Stake Proxy** is always just a single public key hash, which prevents large multisig wallets or scripts from safely creating a **Stake Proxy** order.

Recommendation

We recommend two changes to improve composibility with other scripts:

- Add a `destination` field, which includes an address and a datum to pay the **Stake NFT** to; With this, users could create a **Time Lock** position and immediately list the **Stake NFT** on JPG.store. It also supports the DAO use case we described above.
- Change the `owner` field to a more complex object that allows multisig and script participation. We maintain an open source library called Aicone that has utilities that enable this. It supports all the same constructs of Cardano native scripts, as well as a `Script` condition, which enforces that a particular script is in the stake withdrawals. See the stake validator pattern documented by Anastasia Labs for more details.

Resolution

This issue was resolved as of commit `dc1bc70ab0844d78ba05a57a8c11b8646800bc82`.

CNCT-306 - Changes to MinUTXO might disallow creating new positions

Severity	Status	Commit
Info	Resolved	f9a942f1bf7957fe5339e64309a2247197475ca2

Description

Currently, when opening a **Time Lock** position, the **Stake Pool** script enforces that the input value is exactly equal to the output value, minus the **Reward Token** . This is to prevent loading up the **Stake Pool** script with junk tokens, but also leaves the **Stake Pool** slightly vulnerable to a change in protocol parameters.

If the Cardano protocol parameters were to change, adjusting increasing the minUTXO requirements, then no new **Time Lock** positions could be opened until the **Project Owner** updated the **Stake Pool** UTXO to add more ADA.

Recommendation

It is unlikely that such a protocol parameter change will be made that increases the minUTXO requirements; Even so, any such change is likely to lower it, rather than raise it.

However, if you'd like to guard against this possibility, we recommend relaxing the check, only ensuring that the **Stake Pool** UTXO has exactly the same tokens as the input, but that the quantity of each (except the **Reward Token**) is greater than that on the input.

```
expect self_output.value |> value.flatten |> list.all(fn (policy, name, quantity) {
  let input_quantity = value.quantity_of(self.output.value, policy, name)
  if input_quantity == 0 {
    expect quantity == 0
  } else if policy == datum.policy_id && asset_name == datum.asset_name {
    expect quantity == input_quantity - reward_amount
  } else {
    expect quantity >= input_quantity
  }
})
```

Resolution

This issue was resolved as of commit `f9a942f1bf7957fe5339e64309a2247197475ca2`, with the comment: Allows only ADA to increase, which is perhaps better than the recommended approach.

CNCT-307 - Cannot distribute ADA as a reward

Severity	Status	Commit
Info	Acknowledged	

Description

The scripts make several assumptions that the **Reward Token** is distinct from lovelace. For example:

```
//Time lock should have lovelace, staked asset + reference nft
expect list.length(value.flatten(time_lock_output.value)) == 3
```

This prevents these scripts from being used to distribute ADA.

Recommendation

If this is something that is desired, review and correct all such conditions.

We'd be happy to provide a more detailed list on request.

Resolution

This issue was acknowledged by the project team with the comment: ADA rewards for staking are not in scope so it is correct that it is not possible.

CNCT-400 - Duplicate Reward Settings

Severity	Status	Commit
Witness	Resolved	

Description

We considered the case where a **Stake Pool** might have multiple reward settings with the same `ms_locked` value, but different percentages.

We deemed this safe, because:

- If someone is opening their own **Time Lock** position themselves, they specify the `reward_index` and thus can choose the `reward_setting` with the correct `ms_locked` value; They may even choose the lesser one, for tax purposes, for example.
- If someone opens a **Time Lock** position via the **Stake Proxy** script, they specify the reward percentage they expect to receive. The **Sequencer** must then provide the `reward_index` pointing to the correct `reward_setting`

CNCT-401 - Negative `ms_locked`

Severity	Status	Commit
Witness	Resolved	

Description

We considered the case where a **Stake Pool** might have a zero or negative `ms_locked` value. We deemed this safe, because ultimately it just results in a **Time Lock** position that can be immediately unlocked.

CNCT-402 - Correctness of reward calculation

Severity	Status	Commit
Witness	Resolved	

Description

Given the nuance of the reward calculation, we thoroughly analyzed and make an argument here for its correctness.

The intention of the contracts is for the **Stake Pool** to disburse a portion of the **Reward Token** s to the user only in proportion to the amount of **Reward Token** that is locked by the user for a set period of time.

That is, one way to express the reward calculation is:

- Let `reward_setting` be the chosen reward setting
- Let `locked_until` be the timestamp that the singular **Time Lock** output unlocks
- `locked_until` must be greater than `transaction.upper_bound` plus `reward_setting.locked_ms`
- Let `user_input_tokens` be any **Reward Token** the sum of all **Reward Token** on inputs excluding the **Stake Pool** input.
  - This assumes there is only one stake pool input
  - This assumes that all other inputs are in control of (or authorized by, such as through the stake proxy script) a single user
- Let `minted_tokens` be any **Reward Token** minted in the transaction, which may be negative.
  - This assumes that if a user is able to mint the **Reward Token** , they should also be entitled to include it in their **Time Lock** position
- Let `stake_pool_input` be the quantity of **Reward Token** on the singular **Stake Pool** input
- Let `stake_pool_output` be the quantity of **Reward Token** on the singular **Stake Pool** output
- Let `total_locked_output` be the quantity of **Reward Token** on the singular **Time Lock** output
- Let `total_unlocked_output` be the quantity of **Reward Token** s on outputs excluding the **Stake Pool** and **Time Lock** outputs, such as change returned to the user.
- Let `user_locked_tokens` equal `user_input_tokens + minted_tokens - total_unlocked_output`

- Let `withdrawn_amount` be the amount removed from the stake pool, as defined by `stake_pool_input - stake_pool_output`
- Let `reward_amount` equal `user_locked_tokens * reward_setting.reward_percentage`
- `withdrawn_amount` must equal `reward_amount` (Condition A)
- `total_locked_output` must equal `user_locked_tokens + reward_amount` (Condition B)

In the specification (and the code), however, it is defined as follows:

- The definitions above are reused.
- Let `derived_user_locked_output` be the quantity of **Reward Token** locked by the user, defined as `total_locked_output - withdrawn_amount`
- Let `expected_total_locked_output` be `derived_user_locked_output * (1 + reward_setting.reward_percentage)`
- `total_locked_output` must equal `expected_total_lock_output` (Condition C)

We make an argument here that these definitions are equivalent, i.e. that Condition A and B hold if and only if Condition C holds.

First, let us introduce a lemma:

- If we consider minted tokens one of the "inputs" to the transaction, then `user_input_tokens + minted_tokens + stake_pool_input` constitutes the sum total inputs to the transaction; in particular, `user_input_tokens` is defined such that this is true.
- Similarly, `stake_pool_output + total_locked_output + total_unlocked_output` constitutes the sum total outputs to the transaction; in particular, `total_unlocked_output` is defined such that this is true.
- Because the cardano ledger enforces that transactions are balanced, these two quantities must be equal.

Now we show both directions if the if and only if:

- Assume (A) and (B) hold
  - expanding (A) gives `stake_pool_input - stake_pool_output == user_input_tokens + minted_tokens - total_unlocked_output`
  - expanding (B) gives `total_locked_output == user_locked_tokens + user_locked_tokens * reward_setting.reward_percentage`
  - factoring this gives `total_locked_output == user_locked_tokens * (1 + reward_setting.reward_percentage)`

- comparing with the condition we're trying to show (C), we just need to show that `user_locked_tokens == derived_user_locked_output`
- That is, `user_input_tokens + minted_tokens - total_unlocked_output == total_locked_output - withdrawn_amount`
- That is, `user_input_tokens + minted_tokens - total_unlocked_output == total_locked_output - (stake_pool_input - stake_pool_output)`
- That is, `user_input_tokens + minted_tokens - total_unlocked_output == total_locked_output - stake_pool_input + stake_pool_output`
- Moving inputs and outputs to the left and right sides respectively, `user_input_tokens + minted_tokens + stake_pool_input == total_locked_output + stake_pool_output + total_unlocked_output`
- However, this is just the lemma we gave above. Thus, (C) holds.

- Assume (C) holds
  - expanding (C) gives `total_locked_output == derived_user_locked_output * (1 + reward_setting.reward_percentage)`
  - expanding again, this gives `total_locked_output == (total_locked_output - withdrawn_amount) * (1 + reward_setting.reward_percentage)`
  - expanding again, this gives `total_locked_output == (total_locked_output - (stake_pool_input - stake_pool_output)) * (1 + reward_setting.reward_percentage)`
  - Using the balancing of the transaction, we observe that `user_input_tokens + minted_tokens + stake_pool_input == total_locked_output + stake_pool_output + total_unlocked_output` can be rearranged to `user_input_tokens + minted_tokens - total_unlocked_output == stake_pool_output + total_locked_output - stake_pool_input`, or `user_input_tokens + minted_tokens - total_unlocked_output == total_locked_output - (stake_pool_input - stake_pool_output)`
  - The left hand side is the definition of `user_locked_tokens`, so we can rewrite the statement from the fourth bullet as `total_locked_output == user_locked_tokens * (1 + reward_setting.reward_percentage)`
  - This multiplies out to `total_locked_output == user_locked_tokens + user_locked_tokens * reward_setting.reward_percentage`

- `user_locked_tokens * reward_setting.reward_percentage` is the definition of `reward_amount`
- Therefore, (B) holds, and we just need to show that (A) holds.
- However, using the same fact regarding the balancing of the transaction, where we had that `user_locked_tokens = total_locked_output - (stake_pool_input - stake_pool_output)`, we can see that `total_locked_output - user_locked_tokens == stake_pool_input - stake_pool_output`
- But the left is just the definition for `reward_amount` and the right is just the definition for `withdrawn_amount`
- Therefore, this can be used to rewrite this `reward_amount = withdrawn_amount`, and we've shown that (A) holds.

CNCT-403 - Uniqueness argument of Stake NFTs

Severity	Status	Commit
Witness	Resolved	

Description

When minting a **Stake NFT**, it is assigned a unique asset name. This is done to ensure that only one **Stake NFT** is associated with each **Time Lock** position, and underpins the security of the protocol. It's worth, then, making some argument for why this name can be relied on as unique.

This name is generated by taking the first 28 bytes (224 bits) of the `blake2b_256` hash of the spent **Stake Pool** input when minting the transaction.

Since each transaction output can be spent only once, the **Stake Pool** output reference will be unique.

Then, `blake2b_256` is a cryptographic hash function that is strongly believed to be uniformly distributed and collision resistant.

But is it safe to take only the first 28 bytes?

Since `blake2b_256` is uniformly distributed, the first 28 bytes will also be uniformly distributed. This means that there are 2^{224} possible values for this hash. Even with a birthday paradox argument, someone would need to generate 2^{112} different hashes before there was a 50% chance of a collision.

For comparison, the entire Bitcoin network hashing power currently calculates $552 * 10^{18}$ SHA256 hashes per second. It would take the bitcoin network roughly 298,000 years to have a 50% chance of a collision.

Additionally, to grind out different potential hashes, you would need to generate a chain of two transactions: one that produced a new **Stake Pool** UTXO, and one that minted the **Stake NFT**. This extra processing would slow down this attack even further.

Therefore, we judge the uniqueness of the **Stake NFT** asset name to be secure.

4 - Appendix

4.a - Disclaimer

This Smart Contract Security Audit Report ("Report") is provided on an "as is" basis, for informational purposes only, and should not be construed as investment advice or any other kind of advice on legal, financial, or other matters. The entities and individuals involved in preparing this Report ("Auditors") do not guarantee the accuracy, completeness, or usefulness of the information provided herein and shall not be held liable for any contents, errors, omissions, or inaccuracies in this Report or for any actions taken in reliance thereon.

The Auditors make no claims, promises, or guarantees about the absolute security of the smart contracts audited and the underlying code. The findings, interpretations, and conclusions presented in this Report are based on the best efforts of the Auditors and reflect their professional judgment at the time of the audit. The blockchain and cryptocurrency landscape is rapidly evolving, and new vulnerabilities may emerge that were not identified or considered at the time of the audit. As such, this Report should not be considered as a comprehensive guarantee of the audited smart contracts' security.

The Auditors disclaim, to the fullest extent permitted by law, any and all warranties, whether express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement. The Auditors shall not be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this Report, even if advised of the possibility of such damage.

This Report is not exhaustive and is subject to change without notice. The Auditors reserve the right to update, modify, or revise this Report based on new information, subsequent developments, or further analysis. The Auditors encourage all interested parties to conduct their own independent research and due diligence when evaluating the security of smart contracts.

By using or relying on this Report, you agree to indemnify and hold harmless the Auditors from any claim, demand, action, damage, loss, cost, or expense, including attorney fees, arising out of or relating to your use of or reliance on this Report.

If you have any questions or require further clarification regarding this Report, please contact the contact@sundaeswap.finance.

4.b - Issue Guide

4.b.i - Severity

Severity	Description
Critical	Critical issues highlight exploits, bugs, loss of funds, or other vulnerabilities that prevent the dApp from working as intended. These issues have no workaround.
Major	Major issues highlight exploits, bugs, or other vulnerabilities that cause unexpected transaction failures or may be used to trick general users of the dApp. dApps with Major issues may still be functional.
Minor	Minor issues highlight edge cases where a user can purposefully use the dApp in a non-incentivized way and often lead to a disadvantage for the user.
Info	Info are not issues. These are just pieces of information that are beneficial to the dApp creator, or should be kept in mind for the off-chain code or end user. These are not necessarily acted on or have a resolution, they are logged for the completeness of the audit.
Witness	Witness findings are affirmative findings, which covers bizarre corner cases we considered and found to be safe. Not all such cases are covered, but when something is considered interesting, or might be a common question, we try to include it.

4.b.ii - Status

Status	Description
Resolved	Issues that have been **fixed** by the **project** team.
Mitigated	Issues that have a **partial mitigation**, and are now vulnerable in only **extreme** corner cases.

Acknowledged	Issues that have been **acknowledged** or **partially fixed** by the **project** team. Projects can decide to not **fix** issues for whatever reason.
Identified	Issues that have been **identified** by the **audit** team. These are waiting for a response from the **project** team.

4.c - Revisions

This report was created using a git based workflow. All changes are tracked in a github repo and the report is produced using typst. The report source is available here. All versions with downloadable PDFs can be found on the releases page.

4.d - About Us

Sundae Labs stands at the forefront of innovation within the Cardano ecosystem, distinguished by its pioneering development of the first Automated Market Maker (AMM) Decentralized Exchange (DEX) on Cardano. As a trusted leader in blockchain technology, we offer a comprehensive suite of products and services designed to enhance the Cardano network's functionality and security. Our offerings include Sundae Rewards, Sundae Governance, Sundae Exchange, and Sundae Taste Test—an automated price discovery platform—all available on a Software as a Service (SaaS) basis. These solutions empower other high-profile projects within the ecosystem by providing them with turnkey rewards and governance capabilities, thereby fostering a more robust and scalable blockchain infrastructure.

Beyond our product offerings, Sundae Labs is deeply committed to the advancement of the Cardano community and its underlying technology. We contribute significantly to research and development efforts aimed at improving Cardano's security and scalability. Our engagement with Input Output Global (IOG) initiatives, such as Voltaire, and participation in core technological discussions underscore our dedication to the Cardano ecosystem's growth. Additionally, our expertise extends to software development consulting services, including product design and development, and conducting security audits. Sundae Labs is not just a contributor but a vital partner in Cardano's journey towards achieving its full potential.

4.d.i - Links

Mehen Stablecoin - https://mehen.io

Sundae Labs - https://sundae.fi

Sundae Public Audits - https://github.com/SundaeSwap-finance/sundae-audits-public