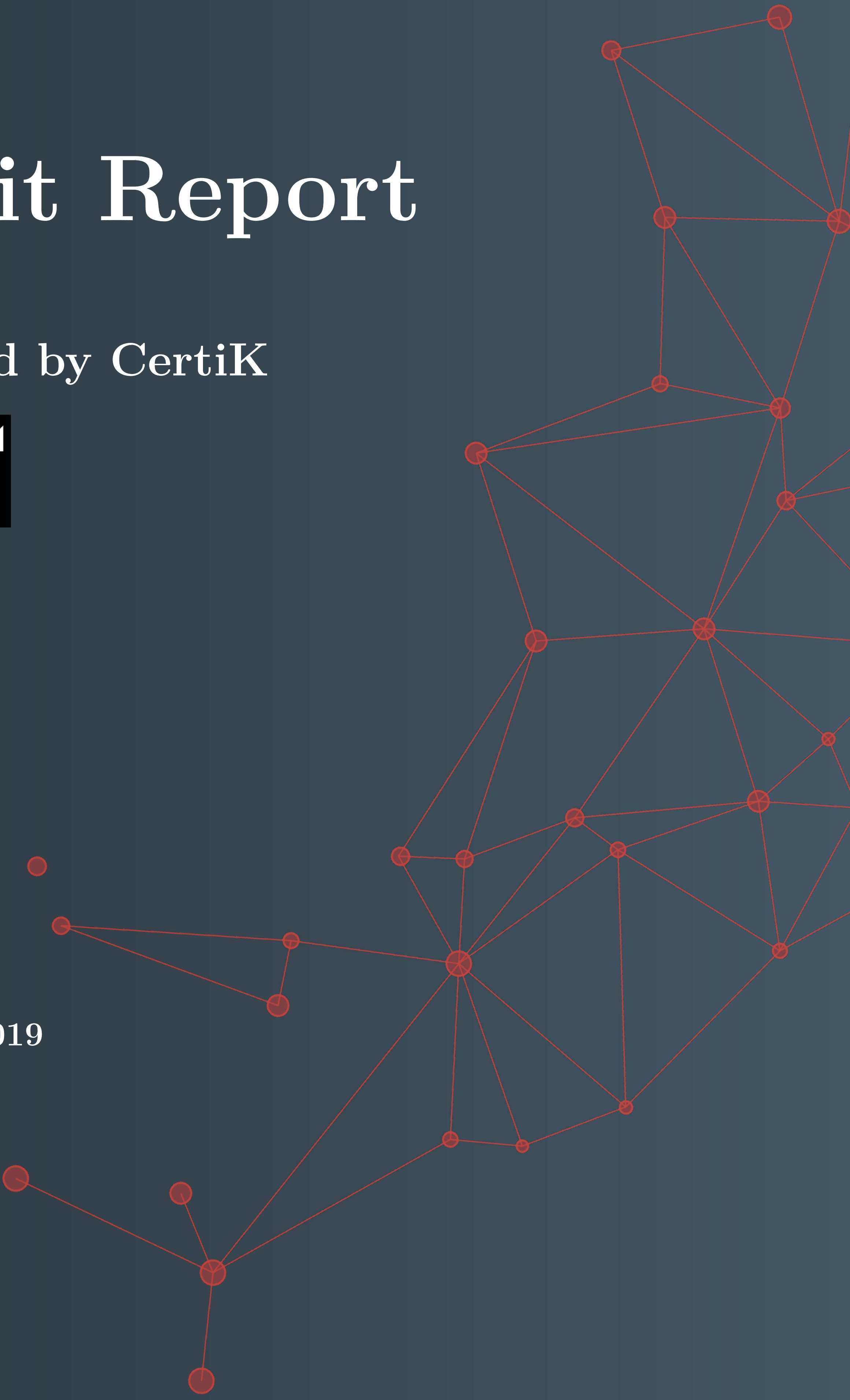
CERTIK

Audit Report

Produced by CertiK

for FETCH



Sep 13th, 2019

CertiK Audit Report
For Fetch.AI

Request Date: 2019-08-13
Revision Date: 2019-09-13
Platform Name: Ethereum

Contents

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 Verification Services Agreement between CertiK and Fetch.AI(the “Company”), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, 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 without CertiK’s prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: `https://certik.org/`

Executive Summary

This report has been prepared for Fetch.AI to discover issues and vulnerabilities in the source code of their dutchStaking and simpleStakePool smart contracts. A comprehensive examination has been performed, utilizing CertiK's Formal Verification Platform, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practice and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line by line manual review of the entire codebase by industry experts.

Vulnerability Classification

CertiK categorizes issues into 3 buckets based on overall risk levels:

Critical

The code implementation does not match the specification, or it could result in the loss of funds for contract owner or users.

Medium

The code implementation does not match the specification under certain conditions, or it could affect the security standard by lost of access control.

Low

The code implementation does not follow best practices, or use suboptimal design patterns, which may lead to security vulnerabilies further down the line.

Testing Summary

PASS

CERTIK *believes this smart contract passes security qualifications to be listed on digital asset exchanges.*

Sep 13, 2019

Type of Issues

CertiK smart label engine applied 100% formal verification coverage on the source code. Our team of engineers ao scanned the source code using our proprietary static analysis tools and code-review methodologies. The following technical issues were found:

Title	Description	Issues	SWC ID
Integer Overflow and Underflow	An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type.	0	SWC-101
Function incorrectness	Function implementation does not meet the specification, leading to intentional or unintentional vulnerabilities.	0	
Buffer Overflow	An attacker is able to write to arbitrary storage locations of a contract if array of out bound happens	0	SWC-124
Reentrancy	A malicious contract can call back into the calling contract before the first invocation of the function is finished.	0	SWC-107
Transaction Order Dependence	A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	0	SWC-114
Timestamp Dependence	Timestamp can be influenced by minors to some degree.	0	SWC-116
Insecure Compiler Version	Using an fixed outdated compiler version or floating pragma can be problematic, if there are publicly disclosed bugs and issues that affect the current compiler version used.	0	SWC-102 SWC-103
Insecure Randomness	Block attributes are insecure to generate random numbers, as they can be influenced by minors to some degree.	0	SWC-120

"tx.origin" for authorization	tx.origin should not be used for authorization. Use msg.sender instead.	0	SWC-115
Delegatecall to Untrusted Callee	Calling into untrusted contracts is very dangerous, the target and arguments provided must be sanitized.	0	SWC-112
State Variable Default Visibility	Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.	0	SWC-108
Function Default Visibility	Functions are public by default. A malicious user is able to make unauthorized or unintended state changes if a developer forgot to set the visibility.	0	SWC-100
Uninitialized variables	Uninitialized local storage variables can point to other unexpected storage variables in the contract.	0	SWC-109
Assertion Failure	The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.	0	SWC-110
Deprecated Solidity Features	Several functions and operators in Solidity are deprecated and should not be used as best practice.	0	SWC-111
Unused variables	Unused variables reduce code quality	0	

Vulnerability Details

Critical

No issue found.

Medium

No issue found.

Low

No issue found.

Review Notes

Source Code SHA-256 Checksum[1]

- **dutchStaking.vy**
  `7377dd040f1d398747c0933d6ef0d81e7cbf15514a2b6570a24191321472763e`

- **simpleStakePool.vy**
  `a25a58d9a7bb86e47a1c5f5f4f5795151e928296b7f071f27e12566d04a8e604`

Summary

CertiK was chosen by Fetch.AI to audit the design and implementation of its `dutchStaking` and `simpleStakePool` smart contracts. To ensure comprehensive protection,the source code has been analyzed by the proprietary CertiK formal verification engine and manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with the best practices in the space.

Overall we found the smart contracts to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the contract is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, keep improving the codebase, and more test coverage and sandbox deployments before the mainnet release.

Documentation

CertiK used the following source of truth to enhance the understanding of Fetch.AI's systems:

1. Fetch.AI Whitepaper[2]
2. Fetch.AI Developer Documentation[3]
3. Fetch.AI Medium Press[4]
4. Project README[5]
5. Project Test Cases[5]

All listed sources act as specification. For any inconsistency discovered between the actual code behavior and the specification, CertiK would consult with the Fetch.AI team for further discussion and confirmation.

[1] Commit: 2cfbd1d0c2edc86cb8f74881d311444cac60b33c
[2] Whitepaper: `https://fetch.ai/uploads/technical-introduction.pdf`
[3] Documentation: `https://docs.fetch.ai/`
[4] Medium: `https://medium.com/fetch-ai`
[5] GitHub: `https://github.com/fetchai/research-staking-contract`

Components

The following simplifed sequence graphs are used to give a brief demonstration of the function logics. The dashed arrows ⇠, ⇢ are redefined for the `token` contract, while the solid arrows ←, → are used for the staking contract.

Owner Priviledged

The following methods in figure 1, 2, figure 3, and figure 4 are to be called by the `dutchStaking` contract owner only.

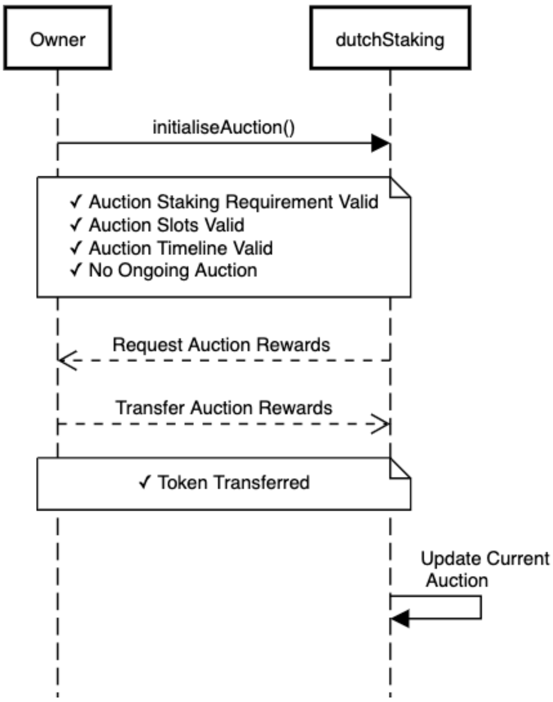


Figure 1: initialiseAuction

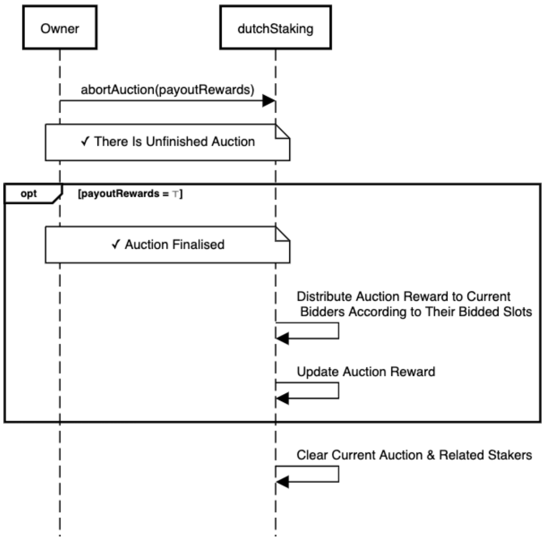

Owner
dutchStaking
abortAuction(payoutRewards)
✓ There Is Unfinished Auction
opt
[payoutRewards = ⊤]
✓ Auction Finalised
Distribute Auction Reward to Current Bidders According to Their Bidded Slots
Update Auction Reward
Clear Current Auction & Related Stakers

Figure 2: abortAuction

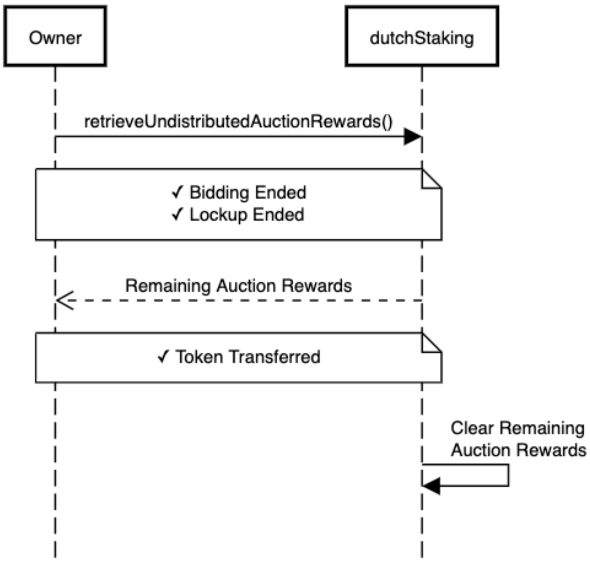


Figure 3: retrieveUndistributedAuctionRewards

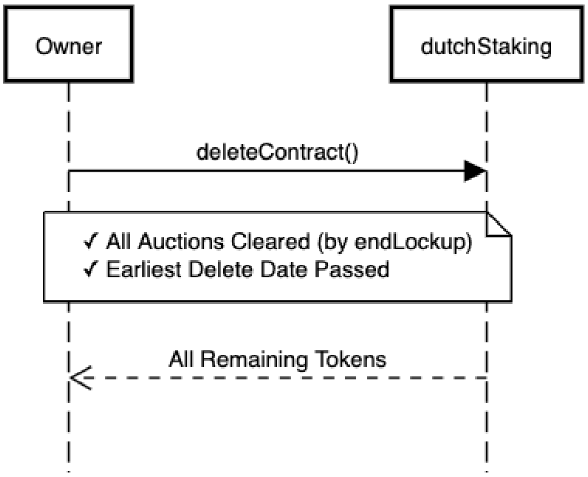

Owner
dutchStaking
deleteContract()
✓ All Auctions Cleared (by endLockup)
✓ Earliest Delete Date Passed
All Remaining Tokens

Figure 4: deleteContract

The determination of whether the current auction is at the bidding phase is determined as follows (see figure 5):

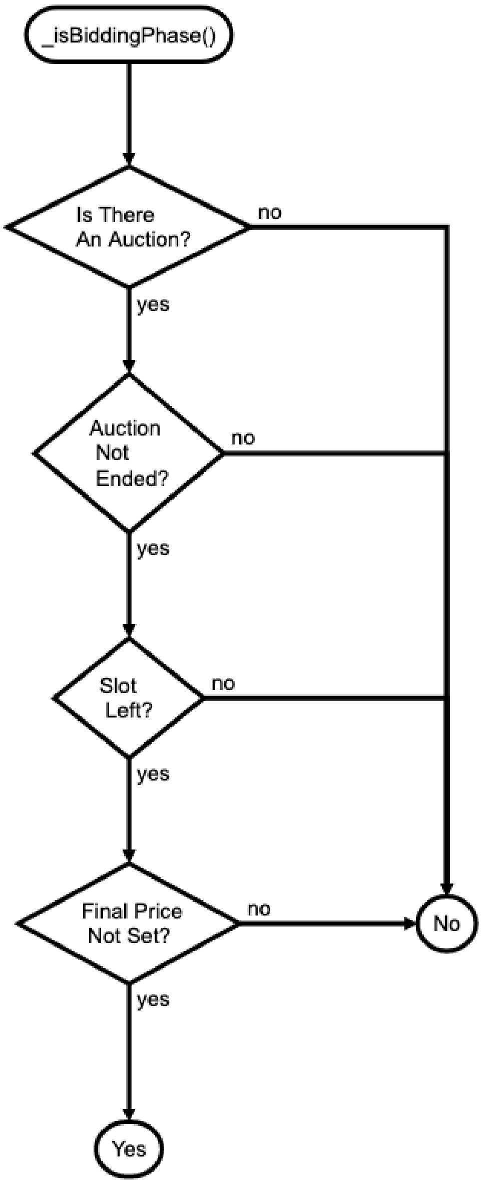


Figure 5: isBiddingPhase

Pool Register

Any individual or contract may register as a pool address for an auction. Users may pledge money to a registered pool. The following methods in figure 6 are to be called by account that desires to be registered as pool and registered pool, respectively.

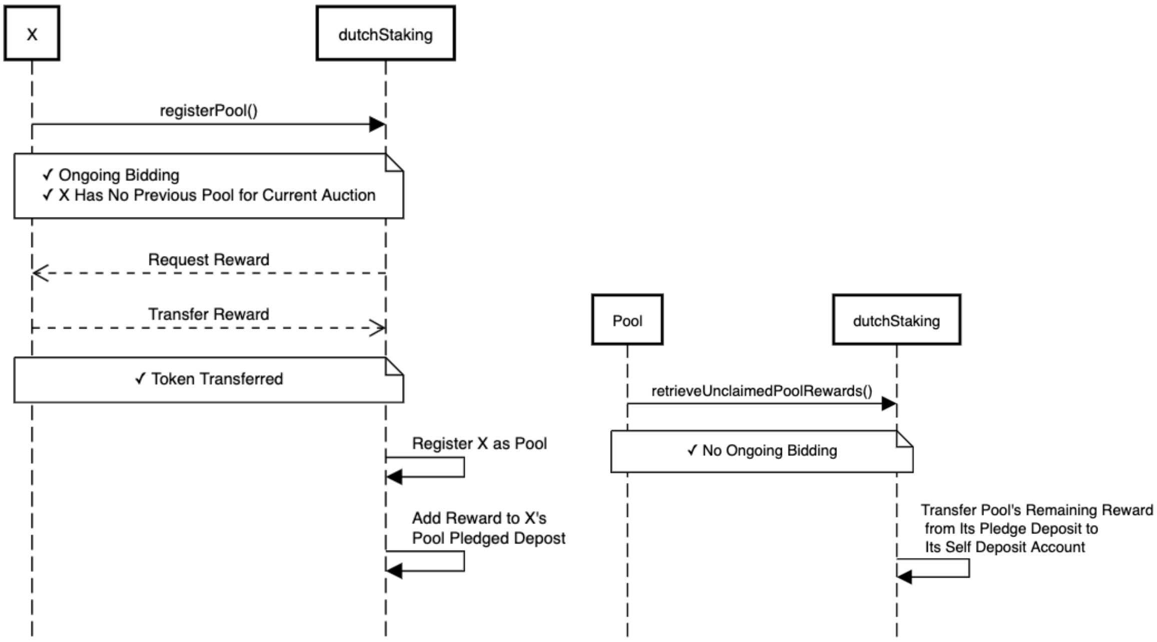


Figure 6: registerPool and retrieveUnclaimedPoolRewards

Any Participant

The following methods in figure 7, 8, 9, figure 12 and figure 13 can be called by any auction participant.

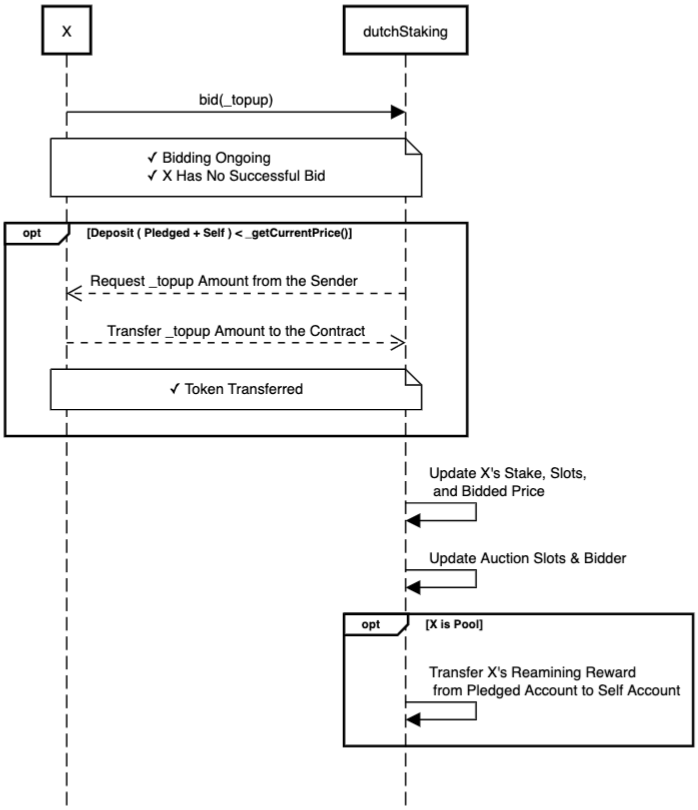

X
dutchStaking
bid(_topup)
✓ Bidding Ongoing
✓ X Has No Successful Bid
opt
[Deposit (Pledged + Self) < _getCurrentPrice()]
Request _topup Amount from the Sender
Transfer _topup Amount to the Contract
✓ Token Transferred
Update X's Stake, Slots,
and Bidded Price
Update Auction Slots & Bidder
opt
[X is Pool]
Transfer X's Reamining Reward
from Pledged Account to Self Account

Figure 7: bid

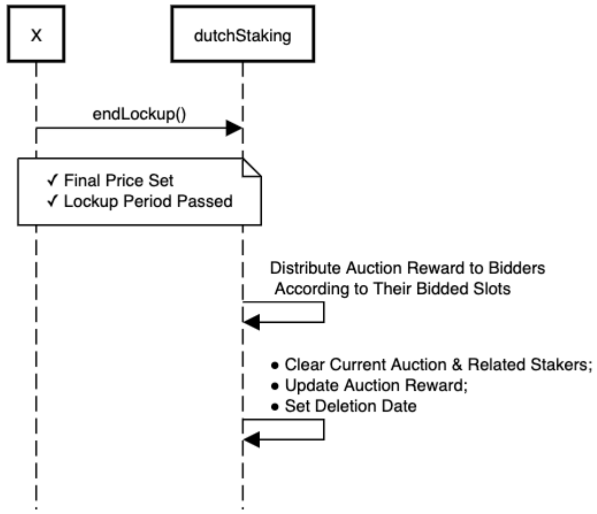

X
dutchStaking
endLockup()
✓ Final Price Set
✓ Lockup Period Passed
Distribute Auction Reward to Bidders
According to Their Bidded Slots
● Clear Current Auction & Related Stakers;
● Update Auction Reward;
● Set Deletion Date

Figure 8: endLockup

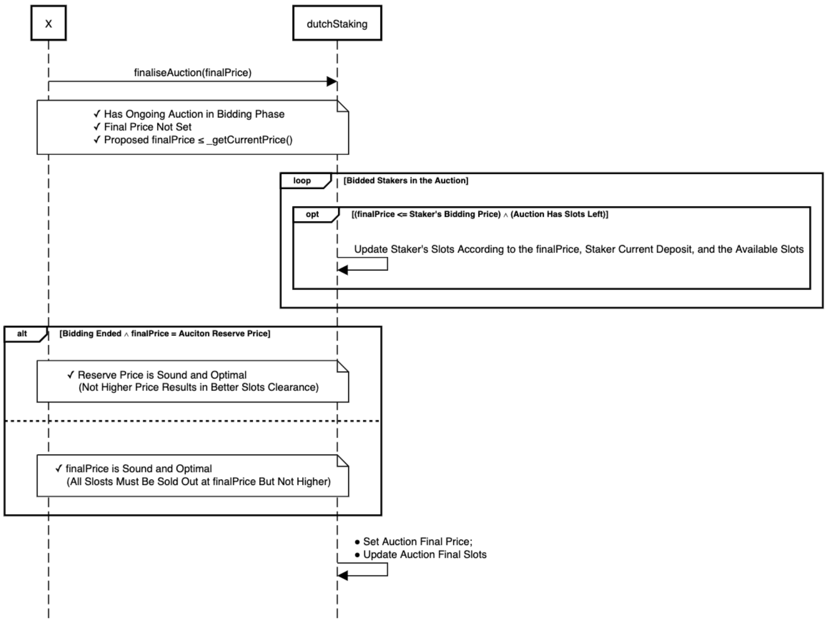

X
dutchStaking
finaliseAuction(finalPrice)
✓ Has Ongoing Auction in Bidding Phase
✓ Final Price Not Set
✓ Proposed finalPrice ≤ _getCurrentPrice()
loop
[Bidded Stakers in the Auction]
opt
[(finalPrice <= Staker's Bidding Price) ∧ (Auction Has Slots Left)]
Update Staker's Slots According to the finalPrice, Staker Current Deposit, and the Available Slots
alt
[Bidding Ended ∧ finalPrice = Auciton Reserve Price]
✓ Reserve Price is Sound and Optimal
(Not Higher Price Results in Better Slots Clearance)
✓ finalPrice is Sound and Optimal
(All Slosts Must Be Sold Out at finalPrice But Not Higher)
● Set Auction Final Price;
● Update Auction Final Slots

Figure 9: finaliseAuction

The flowchart in figure 10 demonstrates the calculation of the price of the slot at the moment in the current auction. It uses `_getScheduledPrice` and `_isFinalised` as shown in figure 10 and 11.

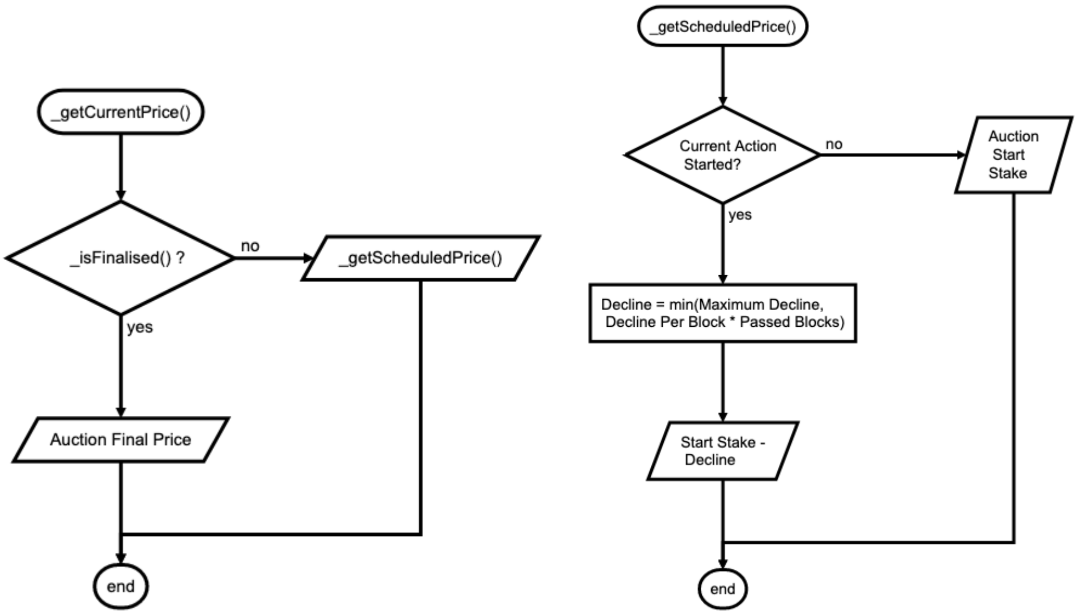


Figure 10: getCurrentPrice & getScheduledPrice

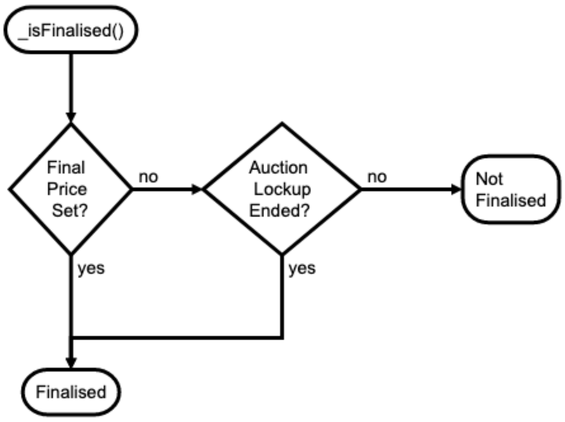


Figure 11: isFinalised

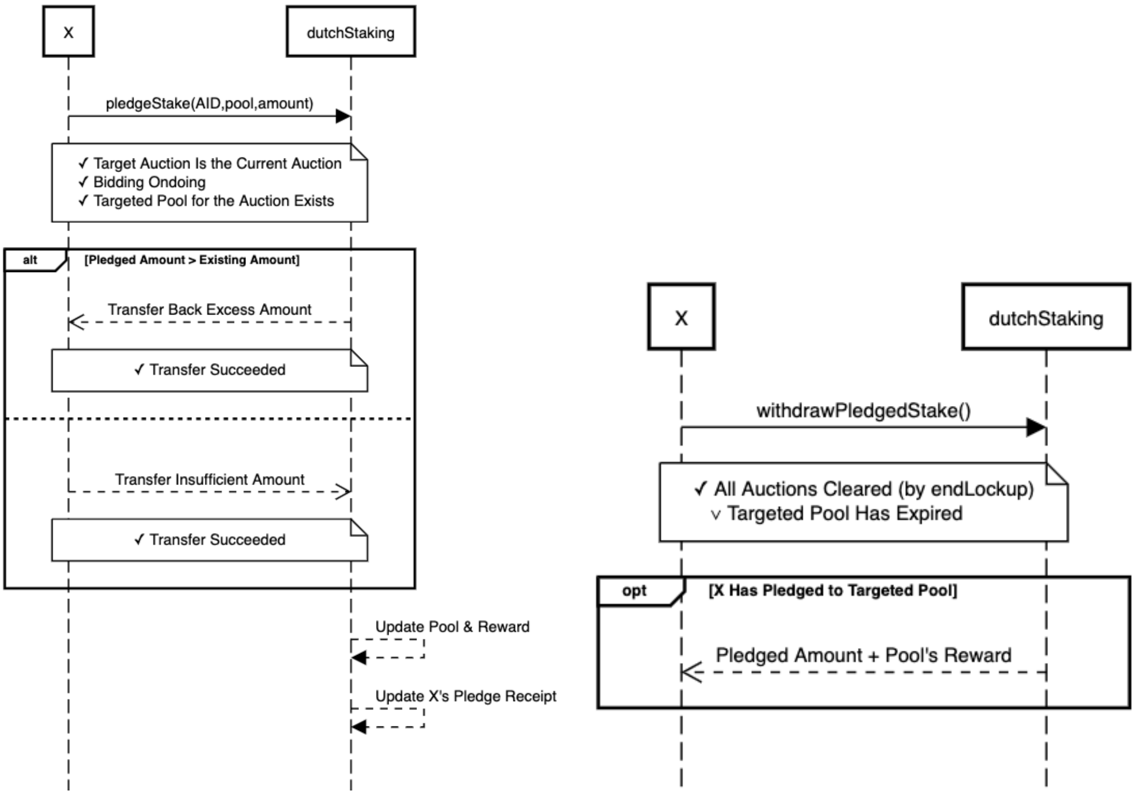

X
dutchStaking
pledgeStake(AID,pool,amount)
✓ Target Auction Is the Current Auction
✓ Bidding Ondoing
✓ Targeted Pool for the Auction Exists
alt
[Pledged Amount > Existing Amount]
Transfer Back Excess Amount
✓ Transfer Succeeded
Transfer Insufficient Amount
✓ Transfer Succeeded
Update Pool & Reward
Update X's Pledge Receipt
X
dutchStaking
withdrawPledgedStake()
✓ All Auctions Cleared (by endLockup)
∨ Targeted Pool Has Expired
opt
[X Has Pledged to Targeted Pool]
Pledged Amount + Pool's Reward

Figure 12: pledgeStake and withdrawPledgedStake

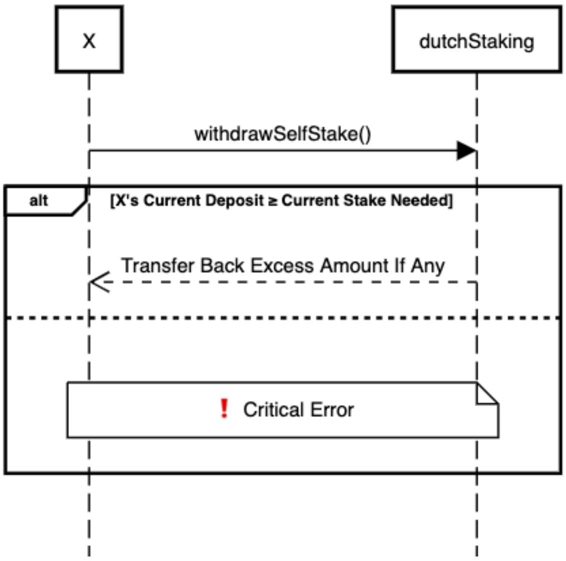

X
dutchStaking
withdrawSelfStake()
alt
[X's Current Deposit ≥ Current Stake Needed]
Transfer Back Excess Amount If Any
! Critical Error

Figure 13: withdrawSelfStake

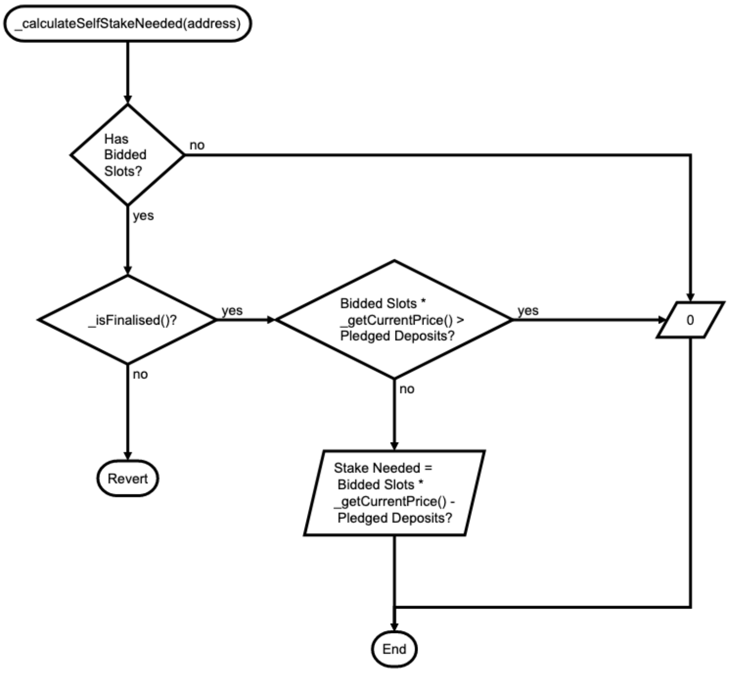

_calculateSelfStakeNeeded(address)
Has Bidded Slots?
no
yes
_isFinalised()?
yes
Bidded Slots * _getCurrentPrice() > Pledged Deposits?
yes
0
no
no
Revert
Stake Needed = Bidded Slots * _getCurrentPrice() - Pledged Deposits?
End

Figure 14: calculateSelfStakeNeeded

The calculation of the minimal required stake at the moment is shown in figure 14.

Details

Items in this section are low impact to the overall aspects of the smart contracts, thus will let client to decide whether to have those reflected in the final deployed version of source codes. They are labeled CRITICAL, MAJOR, MINOR, INFO, and DISCUSSION (in decreasing significance level).

dutchStaking.vy commit *8d1179ccff03690343fdb345909923ffbfb347a5*, previous

- MAJOR `getCurrentPrice`: Taking the ceiling for `declinePerBlock` may result in a price lower than `reserveStake`. If this is not the desired behavior, recommend adding condional clause in `getCurrentPrice()`.
  - (FetchAI - Confirmed) Fixed in commit *7e030eead901aa92c041cfddd8d5dec6fc18fd4a*.
- INFO Recommend supplementing informative error messages to all `assert` statements.
  - (FetchAI - Confirmed) Added in commit *7e030eead901aa92c041cfddd8d5dec6fc18fd4a*.
- DISCUSSION `bid()`: `AID` can be added to function parameter to clarity. It can also be added to the event log.
  - (FetchAI - Confirmed) Refactored in commit *7e030eead901aa92c041cfddd8d5dec6fc18fd4a*.

- DISCUSSION `calculateSelfStakeNeeded()`: The calculation of `selfStakeNeeded` can be refactored for clarity:

  ```
  if self.isStaker[_address]:
        selfStakeNeeded = self.rewardPerSlot
        if self.getCurrentPrice() > self.pledgedDeposits[_address]:
           selfStakeNeeded += (self.getCurrentPrice() - self.pledgedDeposits[
               _address])
  ```

  - (FetchAI - Confirmed) Refactored in commit *7e030eead901aa92c041cfddd8d5dec6fc18fd4a*·

- DISCUSSION `Auction`: The use of `int128`for `slotsSold`, `slotsOnSale`, and `MAX_SLOTS` may be switched to `uint256` for consistency with other fields.

  - (FetchAI - Confirmed) Switched in commit *7e030eead901aa92c041cfddd8d5dec6fc18fd4a*·

- DISCUSSION The `block.number` plays an important role in the contract. Recommend revisiting the difference between `block.number` and `block.timestamp` to ensure that the business need is met.

  `block.timestamp`: Manipulatable by the miner;

  `block.number`: The Ethereum block confirmation currently takes approximately `14 seconds`, and the average block time is between 13 $\sim$ 15 seconds. However the `block.number` will be a dangourous and inaccurate choice of time control during `difficulty bomb` stage or hard/soft fork upgrade of the network.

  - (FetchAI - Confirmed) Resolved by the newly added `abortAuction()` method in commit *2cfbd1d0c2edc86cb8f74881d311444cac60b33c*·

- DISUCSSION `isStaker, stakers`: An owner priviledged function capable of removing malicious staker may be considered added to help prevent griefing attack.

  - (FetchAI - Confirmed) Resolved by the newly added `abortAuction()` method in commit *2cfbd1d0c2edc86cb8f74881d311444cac60b33c*·

Best practice

Smart contract development requires a particular engineering mindset. A failure in the initial construction can be catastrophic, and changing the project after the fact can be exceedingly difficult.

To ensure success and to avoid the challenges above smart contracts should here to best practices at their conception. Below, we summarized a checklist of key points that help to indicate a high overall quality of the current project. (✓ indicates satisfaction; × indicates unsatisfaction; – indicates inapplicablility)

General

- ✓ Corrent environment settings, e.g. compiler version, test framework
- ✓ No compiler warnings
- ✓ Provide error message along with `assert`
- ✓ Use events to monitor contract activities
- ✓ Import and use libraries properly
- – Correct upgradibility mechanism
- ✓ Correct time dependency

Vyper Specific

- ✓ Correct usage of `as_unitless_number()`
- ✓ No redundant default function
- ✓ Correct visibility for functions
- ✓ Correct visibility for state variables
- – Correct handling of `@payable` function
- ✓ No manipulatable obstruction for `selfdestruct`

Privilege Control

- ✓ Provide pause functionality for control and emergency handling
- ✓ Provide time buffer between certain operations
- ✓ Provide proper access control for functions
- ✓ Establish rate limit for certain operations
- ✓ Restrict access to sensitive functions
- ✓ Restrict permission to contract destruction

Documentation

- ✓ Provide project README and execution guidance
- ✓ Provide inline comment for function intention
- ✓ Provide instruction to initialize and execute the test files

Testing

- ✓ Provide migration scripts
- ✓ Provide test scripts and coverage for potential scenarios

Source Code

dutchStaking.vy

```
#------------------------------------------------------------------------------
#
#   Copyright 2019 Fetch.AI Limited
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#------------------------------------------------------------------------------
from vyper.interfaces import ERC20

units: {
    tok: "smallest ERC20 token unit",
}

# maximum possible number of stakers a new auction can specify
MAX_SLOTS: constant(uint256) = 300
# number of blocks during which the auction remains open at reserve price
RESERVE_PRICE_DURATION: constant(uint256) = 25  # number of blocks
# number of seconds before deletion of the contract becomes possible after last lockupEnd() call
DELETE_PERIOD: constant(timedelta) = 60 * (3600 * 24)
# defining the decimals supported in pool rewards per token
REWARD_PER_TOK_DENOMINATOR: constant(uint256(tok)) = 100000

# Structs
struct Auction:
    finalPrice: uint256(tok)
    lockupEnd: uint256
    slotsSold: uint256
    start: uint256
    end: uint256
    startStake: uint256(tok)
    reserveStake: uint256(tok)
    declinePerBlock: uint256(tok)
    slotsOnSale: uint256
    uniqueStakers: uint256

struct Pledge:
    amount: uint256(tok)
    AID: uint256

struct Pool:
    remainingReward: uint256(tok)
    rewardPerTok: uint256(tok)
    AID: uint256

# Events
Bid: event({AID: uint256, _from: indexed(address), currentPrice:
 ↪  uint256(tok), amount: uint256(tok)})
NewAuction: event({AID: uint256, start: uint256, end: uint256,
    lockupEnd: uint256, startStake: uint256(tok), reserveStake:
     ↪  uint256(tok),
    declinePerBlock: uint256(tok), slotsOnSale: uint256,
    rewardPerSlot: uint256(tok)})
PoolRegistration: event({AID: uint256, _address: address,
    maxStake: uint256(tok), rewardPerTok: uint256(tok)})
NewPledge: event({AID: uint256, _from: indexed(address), operator:
 ↪  address, amount: uint256(tok)})
AuctionFinalised: event({AID: uint256, finalPrice: uint256(tok),
 ↪  slotsSold: uint256(tok)})
LockupEnded: event({AID: uint256})
AuctionAborted: event({AID: uint256, rewardsPaid: bool})

# Contract state
token: ERC20
owner: public(address)
earliestDelete: public(timestamp)
# address -> uint256 Slots a staker has won in the current auction
 ↪  (cleared at endLockup())
stakerSlots: map(address, uint256)
# auction winners
stakers: address[MAX_SLOTS]

# pledged stake + committed pool reward, excl. selfStakerDeposit; pool
 ↪  -> deposits
pledgedDeposits: public(map(address, uint256(tok)))
# staker (through pool) -> Pledge{pool, amount}
poolStakerDeposits: public(map(address, Pledge))
# staker (directly) -> amount
selfStakerDeposits: public(map(address, uint256(tok)))
# staker (directly) -> price at which the bid was made
```

```
bidAtPrice: public(map(address, uint256(tok)))
# pool address -> Pool
registeredPools: public(map(address, Pool))

# Auction details
currentAID: public(uint256)
auction: public(Auction)
totalAuctionRewards: public(uint256(tok))
rewardPerSlot: public(uint256(tok))

##################################################################################
# Constant functions
##################################################################################
# @notice True from auction initialisation until either we hit the lower bound on being clear or
#   the auction finalised through finaliseAuction()
@private
@constant
def _isBiddingPhase() -> bool:
    return ((self.auction.lockupEnd > 0)
            and (block.number < self.auction.end)
            and (self.auction.slotsSold < self.auction.slotsOnSale)
            and (self.auction.finalPrice == 0))

# @notice Returns true if either the auction has been finalised or the lockup has ended
# @dev self.auction will be cleared in endLockup() call
# @dev reserveStake > 0 condition in initialiseAuction() guarantees that finalPrice = 0 can never be
#   a valid final price
@private
@constant
def _isFinalised() -> bool:
    return (self.auction.finalPrice > 0) or (self.auction.lockupEnd == 0)

# @notice Calculate the scheduled, linearly declining price of the dutch auction
@private
@constant
def _getScheduledPrice() -> uint256(tok):
    startStake_: uint256(tok) = self.auction.startStake
    start: uint256 = self.auction.start
    if (block.number <= start):
        return startStake_
    else:
        # do not calculate max(startStake - decline, reserveStake) as that could throw on negative startStake - decline
        decline: uint256(tok) = min(self.auction.declinePerBlock * (block.number - start),
                                    startStake_ - self.auction.reserveStake)
        return startStake_ - decline

# @notice Returns the scheduled price of the auction until the auction is finalised. Then returns
#   the final price.
# @dev Auction price declines linearly from auction.start over _duration, then
# stays at _reserveStake for RESERVE_PRICE_DURATION
# @dev Returns zero If no auction is in bidding or lock-up phase
@private
@constant
def _getCurrentPrice() -> (uint256(tok)):
    if self._isFinalised():
        return self.auction.finalPrice
    else:
        scheduledPrice: uint256(tok) = self._getScheduledPrice()
        return scheduledPrice

# @notice Returns the lockup needed by an address that stakes directly
# @dev Will throw if _address is a bidder in current auction & auciton not yet finalised, as the
#   slot number & price are not final yet
# @dev Calling endLockup() will clear all stakerSlots flags and thereby set the required
#   lockups to 0 for all participants
@private
@constant
def _calculateSelfStakeNeeded(_address: address) -> uint256(tok):
    selfStakeNeeded: uint256(tok)
    # these slots can be outdated if auction is not yet finalised / lockup hasn't ended yet
    slotsWon: uint256 = self.stakerSlots[_address]

    if slotsWon > 0:
        assert self._isFinalised(), "Is bidder and auction not finalised yet"
        pledgedDeposit: uint256(tok) = self.pledgedDeposits[_address]
        currentPrice: uint256(tok) = self._getCurrentPrice()

        if (slotsWon * currentPrice) > pledgedDeposit:
            selfStakeNeeded += (slotsWon * currentPrice) - pledgedDeposit
    return selfStakeNeeded

###################################################################################
```

```
# Main functions
######################################################################################
@public
def __init__(_ERC20Address: address):
    self.owner = msg.sender
    self.token = ERC20(_ERC20Address)

# @notice Owner can initialise new auctions
# @dev First auction starts with AID 1
# @dev Requires the transfer of _reward to the contract to be approved with the
#   underlying ERC20 token
# @param _start: start of the price decay
# @param _startStake: initial auction price
# @param _reserveStake: lowest possible auction price >= 1
# @param _duration: duration over which the auction price declines. Total bidding
#   duration is _duration + RESERVE_PRICE_DURATION
# @param _lockup_duration: number of blocks the lockup phase will last
# @param _slotsOnSale: size of the assembly in this cycle
# @param _reward: added to any remaining reward of past auctions
@public
def initialiseAuction(_start: uint256,
                      _startStake: uint256(tok),
                      _reserveStake: uint256(tok),
                      _duration: uint256,
                      _lockup_duration: uint256,
                      _slotsOnSale: uint256,
                      _reward: uint256(tok)):
    assert msg.sender == self.owner, "Owner only"
    assert _startStake > _reserveStake, "Invalid startStake"
    assert (_slotsOnSale > 0) and (_slotsOnSale <= MAX_SLOTS), "Invald slot number"
    assert _start >= block.number, "Start before current block"
    # NOTE: _isFinalised() relies on this requirement
    assert _reserveStake > 0, "Reserve stake has to be at least 1"
    assert self.auction.lockupEnd == 0, "End current auction"

    self.currentAID += 1

    # Use integer-ceil() of the fraction with (+ _duration - 1)
    declinePerBlock: uint256(tok) = (_startStake - _reserveStake + _duration - 1) / _duration
    end: uint256 = _start + _duration + RESERVE_PRICE_DURATION
    self.auction.start = _start
    self.auction.end = end
    self.auction.lockupEnd = end + _lockup_duration
    self.auction.startStake = _startStake
    self.auction.reserveStake = _reserveStake
    self.auction.declinePerBlock = declinePerBlock
    self.auction.slotsOnSale = _slotsOnSale
    # Also acts as the last checked price in _updatePrice()
    self.auction.finalPrice = 0

    # add auction rewards
    self.totalAuctionRewards += _reward
    self.rewardPerSlot = self.totalAuctionRewards /
     ↪  self.auction.slotsOnSale
    success: bool = self.token.transferFrom(msg.sender, self,
     ↪  as_unitless_number(_reward))
    assert success, "Transfer failed"

    log.NewAuction(self.currentAID, _start, end, end + _lockup_duration,
     ↪  _startStake,
                   _reserveStake, declinePerBlock, _slotsOnSale,
                    ↪  self.rewardPerSlot)

# @notice Move unclaimed auction rewards back to the contract owner
# @dev Requires that no auction is in bidding or lockup phase
@public
def retrieveUndistributedAuctionRewards():
    assert msg.sender == self.owner, "Owner only"
    assert self._isBiddingPhase() == False, "In bidding phase"
    assert self.auction.lockupEnd == 0, "Lockup ongoing"
    undistributed: uint256(tok) = self.totalAuctionRewards
    clear(self.totalAuctionRewards)

    success: bool = self.token.transfer(self.owner,
     ↪  as_unitless_number(undistributed))
    assert success, "Transfer failed"

# @notice The owner can clear the auction and all recorded slots in the
 ↪  case of an emergency and
# thereby immediately lift any lockups and allow the immediate
 ↪  withdrawal of any made deposits.
# @param payoutRewards: whether rewards get distributed to bidders
@public
def abortAuction(payoutRewards: bool):
    assert msg.sender == self.owner, "Owner only"
    assert self.auction.lockupEnd > 0, "Nothing to abort"

    staker: address
    rewardPerSlot_: uint256(tok)
    slotsSold: uint256 = self.auction.slotsSold

    if payoutRewards:
        assert self._isFinalised(), "Not finalised"
        rewardPerSlot_ = self.rewardPerSlot
        self.totalAuctionRewards -= slotsSold * rewardPerSlot_

    for i in range(MAX_SLOTS):
        staker = self.stakers[i]
        if staker == ZERO_ADDRESS:
            break

        if payoutRewards:
            self.selfStakerDeposits[staker] += self.stakerSlots[staker] * rewardPerSlot_
        clear(self.stakerSlots[staker])

    clear(self.stakers)
    clear(self.auction)
    clear(self.rewardPerSlot)

    log.AuctionAborted(self.currentAID, payoutRewards)


# @notice Enter a bid into the auction. Requires the sender's deposits + _topup >= currentPrice or
#   specify _topup = 0 to automatically calculate and transfer the topup needed to make a bid at the
#   current price. Beforehand the sender must have approved the ERC20 contract to allow the transfer
#   of at least the topup to the auction contract via ERC20.approve(auctionContract.address, amount)
# @param _topup: Set to 0 to bid current price (automatically calculating and transfering required topup),
#   o/w it will be interpreted as a topup to the existing deposits
# @dev Only one bid per address and auction allowed, as time of bidding also specifies the priority
#   in slot allocation
# @dev No bids below current auction price allowed
@public
def bid(_topup: uint256(tok)):
    assert self._isBiddingPhase(), "Not in bidding phase"
    assert self.stakerSlots[msg.sender] == 0, "Sender already bid"

    _currentAID: uint256 = self.currentAID
    currentPrice: uint256(tok) = self._getCurrentPrice()
    totDeposit: uint256(tok) = self.pledgedDeposits[msg.sender] + self.selfStakerDeposits[msg.sender]

    # cannot modify input argument
    topup: uint256(tok) = _topup
    if (currentPrice > totDeposit) and(_topup == 0):
        topup = currentPrice - totDeposit
    else:
        assert totDeposit + topup >= currentPrice, "Bid below current
         ↪  price"

    # Update deposits & stakers
    self.bidAtPrice[msg.sender] = currentPrice
    self.selfStakerDeposits[msg.sender] += topup
    slots: uint256 = min((totDeposit + topup) / currentPrice,
     ↪  self.auction.slotsOnSale - self.auction.slotsSold)
    self.stakerSlots[msg.sender] = slots
    self.auction.slotsSold += slots
    self.stakers[self.auction.uniqueStakers] = msg.sender
    self.auction.uniqueStakers += 1

    # If pool: move unclaimed rewards and clear
    if self.registeredPools[msg.sender].AID == _currentAID:
        unclaimed: uint256(tok) =
         ↪  self.registeredPools[msg.sender].remainingReward
        clear(self.registeredPools[msg.sender])
        self.pledgedDeposits[msg.sender] -= unclaimed
        self.selfStakerDeposits[msg.sender] += unclaimed

    # Transfer topup if necessary
    if topup > 0:
        success: bool = self.token.transferFrom(msg.sender, self,
         ↪  as_unitless_number(topup))
        assert success, "Transfer failed"
    log.Bid(_currentAID, msg.sender, currentPrice, totDeposit + topup)

# @Notice Anyone can supply the correct final price to finalise the
 ↪  auction and calculate the number of slots each
#   staker has won. Required before lock-up can be ended or withdrawals
 ↪  can be made
# @param finalPrice: proposed solution for the final price. Throws if
 ↪  not the correct solution
# @dev Allows to move the calculation of the price that clear the
 ↪  auction off-chain
@public
def finaliseAuction(finalPrice: uint256(tok)):
    currentPrice: uint256(tok) = self._getCurrentPrice()
    assert finalPrice >= currentPrice, "Suggested solution below current
     ↪  price"
    assert self.auction.finalPrice == 0, "Auction already finalised"
    assert self.auction.lockupEnd >= 0, "Lockup has already ended"

    slotsOnSale: uint256 = self.auction.slotsOnSale
    slotsRemaining: uint256 = slotsOnSale
    slotsRemainingP1: uint256 = slotsOnSale
    finalPriceP1: uint256(tok) = finalPrice + 1

    uniqueStakers_int128: int128 = convert(self.auction.uniqueStakers,
     ↪  int128)
    staker: address
    totDeposit: uint256(tok)
    slots: uint256
    currentSlots: uint256
    _bidAtPrice: uint256(tok)

    for i in range(MAX_SLOTS):
        if i >= uniqueStakers_int128:
            break

        staker = self.stakers[i]
        _bidAtPrice = self.bidAtPrice[staker]
        slots = 0

        if finalPrice <= _bidAtPrice:
            totDeposit = self.selfStakerDeposits[staker] +
             ↪  self.pledgedDeposits[staker]

            if slotsRemaining > 0:
                # finalPrice will always be > 0 as reserveStake required
                 ↪  to be > 0
                slots = min(totDeposit / finalPrice, slotsRemaining)
                currentSlots = self.stakerSlots[staker]
                if slots != currentSlots:
                    self.stakerSlots[staker] = slots
                slotsRemaining -= slots

            if finalPriceP1 <= _bidAtPrice:
                slotsRemainingP1 -= min(totDeposit / finalPriceP1,
                 ↪  slotsRemainingP1)

        # later bidders dropping out of slot-allocation as earlier
         ↪  bidders already claim all slots at the final price
        if slots == 0:
            clear(self.stakerSlots[staker])
            clear(self.stakers[i])

    if (finalPrice == self.auction.reserveStake) and
     ↪  (self._isBiddingPhase() == False):
        # a) reserveStake clears the auction and reserveStake + 1 does
         ↪  not
        doesClear: bool = (slotsRemaining == 0) and (slotsRemainingP1 > 0)
        # b) reserveStake does not clear the auction, accordingly neither will any other higher price
        assert (doesClear or (slotsRemaining > 0)), "reserveStake is not the best solution"
    else:
        assert slotsRemaining == 0, "finalPrice does not clear auction"
        assert slotsRemainingP1 > 0, "Not largest price clearing the auction"

    self.auction.finalPrice = finalPrice
    self.auction.slotsSold = slotsOnSale - slotsRemaining
    log.AuctionFinalised(self.currentAID, finalPrice, slotsOnSale - slotsRemaining)

# @notice Anyone can end the lock-up of an auction, thereby allowing everyone to
#   withdraw their stakes and rewards. Auction must first be finalised through finaliseAuction().
@public
def endLockup():
    # Prevents repeated calls of this function as self.auction will get reset here
    assert self.auction.finalPrice > 0, "Auction not finalised yet or no auction to end"
    assert block.number >= self.auction.lockupEnd, "Lockup not over"

    slotsSold: uint256 = self.auction.slotsSold
    rewardPerSlot_: uint256(tok) = self.rewardPerSlot
    self.totalAuctionRewards -= slotsSold * rewardPerSlot_
    self.earliestDelete = block.timestamp + DELETE_PERIOD

    # distribute rewards & cleanup
    staker: address

    for i in range(MAX_SLOTS):
        staker = self.stakers[i]
        if staker == ZERO_ADDRESS:
            break

        self.selfStakerDeposits[staker] += self.stakerSlots[staker] * rewardPerSlot_
        clear(self.stakerSlots[staker])

    clear(self.stakers)
    clear(self.auction)
    clear(self.rewardPerSlot)

    log.LockupEnded(self.currentAID)

# @param AID: auction ID, has to match self.currentAID
# @param _totalReward: total reward committed to stakers, has to be
 ↪  paid upon
#   calling this and be approved with the ERC20 token
# @param _rewardPerTok: _rewardPerTok / REWARD_PER_TOK_DENOMINATOR will
 ↪  be paid
#   for each stake pledged to the pool. Meaning _rewardPerTok should
 ↪  equal
#   reward per token * REWARD_PER_TOK_DENOMINATOR (see
 ↪  getDenominator())
@public
def registerPool(AID: uint256,
                 _totalReward: uint256(tok),
                 _rewardPerTok: uint256(tok)):
    assert AID == self.currentAID, "Not current auction"
    assert self._isBiddingPhase(), "Not in bidding phase"
    assert self.registeredPools[msg.sender].AID < AID, "Pool already
     ↪  exists"
    assert self.registeredPools[msg.sender].remainingReward == 0,
     ↪  "Unclaimed rewards"

    self.registeredPools[msg.sender] = Pool({remainingReward:
     ↪  _totalReward,
                                             rewardPerTok: _rewardPerTok,
                                             AID: AID})
    # overwrite any pledgedDeposits that existed for the last auction
    self.pledgedDeposits[msg.sender] = _totalReward

    success: bool = self.token.transferFrom(msg.sender, self,
     ↪  as_unitless_number(_totalReward))
    assert success, "Transfer failed"

    maxStake: uint256(tok) = (_totalReward * REWARD_PER_TOK_DENOMINATOR)
     ↪  / _rewardPerTok
    log.PoolRegistration(AID, msg.sender, maxStake, _rewardPerTok)

# @notice Move pool rewards that were not claimed by anyone into
#   selfStakerDeposits. Automatically done if pool enters a bid.
# @dev Requires that the auction has passed the bidding phase
@public
def retrieveUnclaimedPoolRewards():
    assert ((self._isBiddingPhase() == False)
             or (self.registeredPools[msg.sender].AID <
              ↪  self.currentAID)), "Bidding phase of AID not over"

    unclaimed: uint256(tok) =
     ↪  self.registeredPools[msg.sender].remainingReward
    clear(self.registeredPools[msg.sender])

    self.pledgedDeposits[msg.sender] -= unclaimed
    self.selfStakerDeposits[msg.sender] += unclaimed

# @notice Pledge stake to a staking pool. Possible from auction
 ↪  intialisation
#   until the end of the bidding phase or until the pool has made a
 ↪  bid.
#   Stake from the last auction can be taken over to the next auction.
 ↪  If amount
#   exceeds the previous stake, this contract must be approved with the
 ↪  ERC20 token
#   to transfer the difference to this contract.
# @dev Only one pledge per address and auction allowed
# @dev If decreasing the pledge, the difference is immediately paid out
# @dev If the pool operator has already bid, this will throw with
 ↪  "Rewards depleted"
# @param AID: The auction ID
# @pool: The address of the pool
# @param amount: The new total amount, not the difference to existing
 ↪  pledges
@public
def pledgeStake(AID: uint256, pool: address, amount: uint256(tok)):
    assert AID == self.currentAID, "Not current AID"
    assert self._isBiddingPhase(), "Not in bidding phase"
    assert self.registeredPools[pool].AID == AID, "Not a registered pool"

    existingPledgeAmount: uint256(tok) =
     ↪  self.poolStakerDeposits[msg.sender].amount
    assert self.poolStakerDeposits[msg.sender].AID < AID, "Already
     ↪  pledged"

    reward: uint256(tok) = ((self.registeredPools[pool].rewardPerTok *
     ↪  amount)
                            / REWARD_PER_TOK_DENOMINATOR)
    assert self.registeredPools[pool].remainingReward >= reward, "Rewards
     ↪  depleted"
    self.registeredPools[pool].remainingReward -= reward

    # pool reward is already included in pledgedDeposits
    self.pledgedDeposits[pool] += amount
    self.poolStakerDeposits[msg.sender] = Pledge({amount: amount +
     ↪  reward,
                                                  AID: AID})

    if amount > existingPledgeAmount:
        success: bool = self.token.transferFrom(msg.sender, self,
         ↪  as_unitless_number(amount - existingPledgeAmount))
        assert success, "Transfer failed"
    elif amount < existingPledgeAmount:
        success: bool = self.token.transfer(msg.sender,
         ↪  as_unitless_number(existingPledgeAmount - amount))
        assert success, "Transfer failed"

    log.NewPledge(AID, msg.sender, pool, amount)

# @notice Withdraw any self-stake exceeding the required lockup. In
 ↪  case sender is a bidder in the
#   current auction, this requires the auction to be finalised through
 ↪  finaliseAuction(),
#   o/w _calculateSelfStakeNeeded() will throw
@public
def withdrawSelfStake() -> uint256(tok):
    selfStake: uint256(tok) = self.selfStakerDeposits[msg.sender]
    selfStakeNeeded: uint256(tok) =
     ↪  self._calculateSelfStakeNeeded(msg.sender)
    # not guaranteed to be initialised to 0 without setting it
     ↪  explicitly
    withdrawal: uint256(tok) = 0

    if selfStake > selfStakeNeeded:
        withdrawal = selfStake - selfStakeNeeded
        self.selfStakerDeposits[msg.sender] -= withdrawal
    elif selfStake < selfStakeNeeded:
        assert False, "Critical failure"

    success: bool = self.token.transfer(msg.sender,
     ↪  as_unitless_number(withdrawal))
    assert success, "Transfer failed"

    return withdrawal

# @notice Withdraw pledged stake after the lock-up has ended
@public
def withdrawPledgedStake() -> uint256(tok):
    withdrawal: uint256(tok)
    if ((self.poolStakerDeposits[msg.sender].AID < self.currentAID)
        or (self.auction.lockupEnd == 0)):
        withdrawal += self.poolStakerDeposits[msg.sender].amount
        clear(self.poolStakerDeposits[msg.sender])

    success: bool = self.token.transfer(msg.sender,
     ↪  as_unitless_number(withdrawal))
    assert success, "Transfer failed"

    return withdrawal

# @notice Allow the owner to remove the contract, given that no auction is
#   active and at least DELETE_PERIOD blocks have past since the last lock-up end.
@public
def deleteContract():
    assert msg.sender == self.owner, "Owner only"
    assert self.auction.lockupEnd == 0, "In lockup phase"
    assert block.timestamp >= self.earliestDelete, "earliestDelete not reached"

    contractBalance: uint256 = self.token.balanceOf(self)
    success: bool = self.token.transfer(self.owner, contractBalance)
    assert success, "Transfer failed"

    selfdestruct(self.owner)

#################################################################################
# Getters
#################################################################################
@public
@constant
def getERC20Address() -> address:
    return self.token

@public
@constant
def getDenominator() -> uint256(tok):
    return REWARD_PER_TOK_DENOMINATOR

@public
@constant
def getFinalStakerSlots(staker: address) -> uint256:
    assert self._isFinalised(), "Slots not yet final"
    return self.stakerSlots[staker]

# @dev Always returns an array of MAX_SLOTS with elements > unique bidders = zero
@public
@constant
def getFinalStakers() -> address[MAX_SLOTS]:
    assert self._isFinalised(), "Stakers not yet final"
    return self.stakers

```

```
@public
@constant
def getFinalSlotsSold() -> uint256:
    assert self._isFinalised(), "Slots not yet final"
    return self.auction.slotsSold

@public
@constant
def isBiddingPhase() -> bool:
    return self._isBiddingPhase()

@public
@constant
def isFinalised() -> bool:
    return self._isFinalised()

@public
@constant
def getCurrentPrice() -> uint256(tok):
    return self._getCurrentPrice()

@public
@constant
def calculateSelfStakeNeeded(_address: address) -> uint256(tok):
    return self._calculateSelfStakeNeeded(_address)
```

simpleStakePool.vy

```
#-------------------------------------------------------------------------------
#
#   Copyright 2019 Fetch.AI Limited
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#-------------------------------------------------------------------------------
from vyper.interfaces import ERC20
import interfaces.dutchStakingInterface as Auction
```

```

units: {
    tok: "smallest ERC20 token unit",
}

# Only for the pool owner to keep track. This info could also be
 ↪  inferred from the auction contract,
# allowing to safe storage costs
struct Pool:
    maxStake: uint256(tok)
    totalReward: uint256(tok)
    rewardPerTok: uint256(tok)

token: ERC20
auctionContract: Auction
owner: public(address)

# AID -> pool
registeredPools: public(map(uint256, Pool))
rewardPerTokDenominator: uint256(tok)


@public
def __init__(_ERC20Address: address, _auctionContract: address):
    self.owner = msg.sender
    self.token = ERC20(_ERC20Address)
    self.auctionContract = Auction(_auctionContract)
    self.rewardPerTokDenominator = self.auctionContract.getDenominator()

# @dev Requires that this contract has an ERC20 balance of _totalReward
# @dev Cleans up storage for any registered pool for the previous
 ↪  auction
@public
def registerPool(AID: uint256,
                  _maxStake: uint256(tok),
                  _totalReward: uint256(tok),
                  _rewardPerTok: uint256(tok)):
    assert msg.sender == self.owner, "Owner only"
    assert (_totalReward * self.rewardPerTokDenominator) / _maxStake ==
     ↪  _rewardPerTok, "_totalReward, _rewardPerTok mismatch"

    self.registeredPools[AID] = Pool({maxStake: _maxStake,
                                      totalReward: _totalReward,
                                      rewardPerTok: _rewardPerTok})

    self.token.approve(self.auctionContract,
     ↪  as_unitless_number(_totalReward))
    self.auctionContract.registerPool(AID, _totalReward, _rewardPerTok)

    clear(self.registeredPools[AID - 1])

# @dev Enter a bid at the current price, given that pledgedDeposits >=
 ↪  price
@public
def bidPledgedStake():
    assert msg.sender == self.owner, "Owner only"
    amount: uint256(tok)
    self.auctionContract.bid(amount)

# @notice Make a bid at the current price, adding any amount exceeding
#   pledgedDeposits as selfStake. Requires that this contract has an
 ↪  ERC20
#   balance of that amount
@public
def bidPledgedAndSelfStake(amount: uint256(tok)):
    assert msg.sender == self.owner, "Owner only"

    currentPrice: uint256(tok) = self.auctionContract.getCurrentPrice()
    existingPoolStake: uint256(tok) =
     ↪  self.auctionContract.pledgedDeposits(self) +
     ↪  self.auctionContract.selfStakerDeposits(self)
    toApprove: uint256(tok)

    if (amount == 0) and (currentPrice > existingPoolStake):
        toApprove = currentPrice - existingPoolStake
    else:
        assert amount >= currentPrice - existingPoolStake, "Amount below
         ↪  price"
        toApprove = amount - existingPoolStake

    self.token.approve(self.auctionContract,
     ↪  as_unitless_number(toApprove))
    self.auctionContract.bid(toApprove)

# @notice Withdraw self stake and accumulated rewards, transfer them to
 ↪  this contract
@public
def withdrawSelfStake() -> uint256(tok):
    assert msg.sender == self.owner, "Owner only"
    return self.auctionContract.withdrawSelfStake()

# @notice Withdraw this contracts balance
# @param amount: amount to transfer to the owner. Set to 0 to transfer
 ↪  full balance
@public
def retrievePoolBalance(amount: uint256):
    assert msg.sender == self.owner, "Owner only"
    if amount == 0:
        self.token.transfer(self.owner, self.token.balanceOf(self))
    else:
        self.token.transfer(self.owner, amount)

# @notice Retrieve unclaimed pool rewards.
# @dev Automatically done if a bid is entered
@public
def retrieveUnclaimedPoolRewards():
    assert msg.sender == self.owner, "Owner only"
    self.auctionContract.retrieveUnclaimedPoolRewards()
```