Quantstamp Security Assessment Certificate

August 3rd 2020 — Quantstamp Verified

XDai Easy Staking

This security assessment was prepared by Quantstamp, the leader in blockchain security

Executive Summary

Type	Smart contract
Auditors	Sung-Shine Lee, Research Engineer Sebastian Banescu, Senior Research Engineer Jake Goh Si Yuan, Security Auditor
Timeline	2020-07-16 through 2020-08-03
EVM	Muir Glacier
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	README.md
Documentation Quality	High
Test Quality	High

Source Code

Repository	Commit
easy-staking-contracts	d1f41c
None	ebecd
None	724a1

Goals

- Do functions have proper access control logic?
- Are there centralized components which users should be aware of?
- Do the contracts adhere to best practices?
- Are the calculations and funds distribution correct?

Total Issues	**9** (8 Resolved)
High Risk Issues	**1** (1 Resolved)
Medium Risk Issues	**1** (1 Resolved)
Low Risk Issues	**4** (4 Resolved)
Informational Risk Issues	**3** (2 Resolved)
Undetermined Risk Issues	0 (0 Resolved)

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.
Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Resolved	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

In general, the code is well written, well documented, and well tested. We have, nevertheless, identified one high and one medium severity issue. The high severity issue points out the inadequate implementation of reentrancy guard which still allows reentrancy. The medium refers to the unchecked external calls.

ID	Description	Severity	Status
QSP-1	Reentrancy Guard not implemented properly	High	Fixed
QSP-2	Unchecked external calls	Medium	Fixed
QSP-3	Withdrawal Unlock Duration can be set very small	Low	Fixed
QSP-4	Inconsistent use of re-entrancy guard	Low	Fixed
QSP-5	Privileged Roles and Ownership	Low	Fixed
QSP-6	Loss of Precision in Arithmetic Calculations	Low	Fixed
QSP-7	Underspecific `claimTokens` leads to winner-takes-all	Informational	Fixed
QSP-8	Trapped Tokens and Temporary Denial of Service due to overflow of `lastDepositIds[address]`	Informational	Acknowledged
QSP-9	Block Timestamp Manipulation	Informational	Fixed

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.6.12

Steps taken to run the tools:

1. Installed the Slither tool: `pip install slither-analyzer`
2. Run Slither from the project directory: `slither .s`

Findings

QSP-1 Reentrancy Guard not implemented properly

Severity: *High Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: The `_setLocked` function description indicates that it should prevent reentrancy. However, it does not check whether the lock is being held before and thus diverts from typical mutex implementations. This causes different kinds of problems:

- In the `deposit()` where `_setLocked()` is called, there is no check whether `locked` is set to `true`. Therefore, this method does not stop reentrancy.
- In `onTokenTransfer()`, it checks the lock and only deposits when it is unlocked. Note that it doesn't fail the transaction when the lock is `true`. It is possible to create a mismatch between contracts as the contract that calls this function might think the deposit succeeded.

Recommendation: Use the reentrancy guard from OpenZeppelin.
Update: The team informed us that `setLocked()` is not to guard reentrancy and its main functionality is to avoid the `_deposit()` to be called repeatedly during the ERC677 token transfer. The team updated the comment in `ebecd` and we consider the implementation reasonable.

QSP-2 Unchecked external calls

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: In calls `token.transfer`, `token.mint` and `token.transferFrom`, the return result from these external calls to `token` are not checked. In case of possible flawed implementation or unthrown failure there can be inconsistent state between `token` and `EasyStaking`. For example, if the `token.transfer()` failed in L448, the balances are still updated in the contract at L437 and it would emit an `Withdrawn` event that has actually failed.
Slither findings:

- EasyStaking.deposit(uint256,uint256) (EasyStaking.sol#207-213) ignores return value by token.transferFrom(msg.sender,address(this),_amount) (EasyStaking.sol#211)
- EasyStaking.claimTokens(address,address) (EasyStaking.sol#282-298) ignores return value by token.transfer(_to,amount) (EasyStaking.sol#292)
- EasyStaking._withdraw(address,uint256,uint256,bool) (EasyStaking.sol#432-450) ignores return value by token.transfer(liquidityProvidersRewardAddress,feeValue) (EasyStaking.sol#446)
- EasyStaking._withdraw(address,uint256,uint256,bool) (EasyStaking.sol#432-450) ignores return value by token.transfer(_sender,amount) (EasyStaking.sol#448)
- EasyStaking._mint(address,uint256,uint256) (EasyStaking.sol#458-469) ignores return value by token.mint(address(this),total) (EasyStaking.sol#463)
- EasyStaking._mint(address,uint256,uint256) (EasyStaking.sol#458-469) ignores return value by token.transfer(liquidityProvidersRewardAddress,total.sub(userShare)) (EasyStaking.sol#466)

Recommendation: Always check the return values of external calls and act accordingly.
Update: The issue is fixed in `ebecd` according to the recommendation.

QSP-3 Withdrawal Unlock Duration can be set very small

Severity: *Low Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: In `setWithdrawalUnlockDuration()`, if the unlock duration is small, e.g. 1 block, while users can still technically withdraw their funds, in practice, it might be very hard for them to do so.

Recommendation: Consider requiring the duration to be sufficient for end-users to be able to easily withdraw their funds.
Update: The issue is fixed in `ebecd` via adding a requirement so that the unlock duration has to be greater than 1 hour.

QSP-4 Inconsistent use of re-entrancy guard

Severity: *Low Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: The contract occasionally makes external calls to `IERC20Mintable token` address. It is assumed that this is the `STAKE` token contract and is generally trusted. This `token` state variable is also immutable.
However, in function `deposit(uint256, uint256)` L207 we see that the mutex is used to protect function call `token.transferFrom`. Thus the trust model here can be viewed as there may possibly be some reentrancy possibilities from the external calls to `token`. Under this trust model, all external calls to `token` should consider the possibility of reentrancy.
There are further non-view calls to `token` through functions such as `token.transfer`, `token.mint`. We suggest using the re-entrancy guard with those calls to be consistent and secure.

Recommendation: Use reentrancy guard in a consistent way.
Update: On `ebecd`, the original description is related to `QSP-1` and thus is considered to be fixed. However, that means that contract as its current form is not protected explicitly by re-entrancy guard. We suggest to implement a proper re-entrancy guard on `_withdraw()` and `_deposit()`.
Update: On `724a1`, the issue is resolved by implementing re-entrancy guard on `withdraw()` and `_deposit()`.

QSP-5 Privileged Roles and Ownership

Severity: *Low Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: Smart contracts will often have `owner` variables to designate the person with special privileges to make modifications to the smart contract. However, this centralization of power needs to be made clear to the users, especially when the `owner` is given higher level of privileges.
The owner of the `EasyStaking` contract is able to change several important parameters of the contract repeatedly at any moment in time. These parameters influence:

1. the accrued emissions: `setTotalSupplyFactor` and `setSigmoidParameters`
2. the amount of fees that are charged for instant withdrawals: `setFee`
3. the time when withdrawals with no fees can be performed: `setWithdrawalLockDuration` and `setWithdrawalUnlockDuration`
4. the address where liquidity provider rewards (fees) are transferred to: `setLiquidityProvidersRewardAddress`.

Additionally, the owner can claim unsupported tokens accidentally sent to the contract: `claimTokens`.
The owner could even front-run end-users by calling functions such as `setFee()` when an end-user makes a forced withdrawal via `makeForcedWithdrawal()`. Similar front-running scenarios can happen with the other functions mentioned above as well.

Recommendation: This should be made clear to the end-users via the documentation. Currently, the functions that the owner can call are listed in the README.md, however, the consequences of these functions may not be clear to end-users.
Update: The issue is partially mitigated in `ebecd` by adding a 7-day period between the request to set a new value and the final setting of this value. Still, we recommend to communicate what the owner can do in the README to completely mitigate the issue.
Update: On `724a1`, the team updated the `README` and resolved the issue.

QSP-6 Loss of Precision in Arithmetic Calculations

Severity: *Low Risk*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: Solidity integer division might truncate. As a result, performing a multiply before a division might lead to loss of precision. There are 2 occurrences in `getAccruedEmission` on L397 and L398.

```
total = _amount.mul(MAX_EMISSION_RATE).div(1 ether).mul(timePassed).div(YEAR);
userShare = _amount.mul(userEmissionRate).div(1 ether).mul(timePassed).div(YEAR);
```

Recommendation: Move the division after the multiplication.
Update: Fixed in `ebecd` according to recommendation.

QSP-7 Underspecific `claimTokens` leads to winner-takes-all

Severity: *Informational*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: The owner-only function `claimTokens` is intended to help retrieve tokens and native token sent to the contract address, to be forwarded to a `_to` payable address.
Currently, the function calculates the amount of tokens/native token to be forwarded by taking the entire balance received unknowingly by the contract address. If this was intentional as the forwarding address is meant to be an intermediary that allowed for further deliberate distribution, then the issue is

no more.
However, if not, despite `onlyOwner` access control to this function, this may lead to some unintentional and intentional flaws.

Exploit Scenario:

1. Alice has sent 10 `XDai` to the contract. She tries to claim the `XDai`s back to her, and the `owner` steps in to help.
2. During the time before the `owner` is able to send a transaction calling `claimTokens(address(0), address(Alice))`, Bob also sent some `XDai` to the contract.
3. The `owner`, not able to distinguish Bob's transaction before sending out his own, sends out Alice and Bob's total `XDai`s to Alice.

Recommendation: Set another parameter `uint256 _amount` to `claimTokens`.
Update: Fixed in `ebecd` according to recommendation.

QSP-8 Trapped Tokens and Temporary Denial of Service due to overflow of `lastDepositIds[address]`

Severity: *Informational*

Status: Acknowledged

Description: `lastDepositIds[address]` is used as a way to track the different unique deposits IDs for a given address. In `deposit()` and `withdraw()`, the deposit ID `0` is treated as a special case as a validation for wrong ID. Thus funds would be locked if they are deposited with ID `0`.
At the same time, given that actions of `withdraw` and `deposit` require the validation of `depositId <= lastDepositIds[address]`, it can be considered a temporary denial of service if `lastDepositIds[address]` is set to `0` through overflow. It is only temporary as it can be circumvented by bringing deposits into the account again and increasing `lastDepositIds[address]`.

Exploit Scenario:

1. Increase `lastDepositIds[address]` to `MAXINT(uint256)` through `deposit()` or external token transfer `onTokenTransfer()`.
2. Perform external token transfer `onTokenTransfer()`, triggering overflow and bringing `lastDepositIds[_sender]` to `0`.

Recommendation: Before allowing `++lastDepositIds[_sender]`, or pass responsibility to `_deposit` to validate the for `_id` the same way as `_withdraw`, we recommend to perform validation on `onTokenTransfer` either with `SafeMath` which would reject further deposits or simply fix the max ID and prevent the ID from overflowing.
Be mindful that this is setting the expectation that `MAXINT(uint256)` is the `id` that all deposits beyond that number will default into, if solution is to prevent overflow from happening. (or reject further deposits).
Update: The team informed us that "We exclude the possibility of creating such a large number of deposits". We consider this a reasonable assumption.

QSP-9 Block Timestamp Manipulation

Severity: *Informational*

Status: Fixed

File(s) affected: `EasyStaking.sol`

Description: Projects may rely on block timestamps for various purposes. However, it's important to realize that miners individually set the timestamp of a block, and attackers may be able to manipulate timestamps for their own purposes. If a smart contract relies on a timestamp, it must take this into account. Here, the user emission rate is computed based on the `block.timestamp`, which could be affected by malicious miners.

Recommendation: Add an explicit warning in the end-user documentation indicating that expiration timestamps can have a 900 second error.
Update: This is fixed in `ebecd` according to the recommendation.

Adherence to Specification

The implementation adheres to the documentation provided.

Code Documentation

The Ethereum code generally adheres to the inline comments and provided documentation. Code comments were included throughout.

Adherence to Best Practices

1. In `EasyStaking.sol`, `_withdraw()`, when the `_amount` is `0`, the function withdraws everything for the user. This is not intuitive and may become a source of error if other projects try to integrate with this project. We recommend using `MAX_UINT256` as a special value as it is clearer and would not appear in normal calculations.
2. In `EasyStaking.sol` the following parameters of the event are not indexed:

- `Deposited`, `L33`: `amount`, `balance`, `accruedEmission` and `prevDepositDuration`.
- `Withdrawn`, `L59`: `amount`, `fee`, `balance`, `accruedEmission` and `lastDepositDuration`.
- `FeeSet`, `L74`: `value` and `sender`
- `WithdrawalLockDurationSet`, `L81`: `value` and `sender`
- `WithdrawalUnlockDurationSet`, `L88`: `value` and `sender`
- `TotalSupplyFactorSet`, `L95`: value`and`sender`
- `SigmoidParametersSet`, `L104`: `a`, `b`, `c` and `sender`
- `LiquidityProvidersRewardAddressSet`, `L111`: `value` and `sender`

3. `EasyStaking.sol`, `L396`: The `require` statement here should be replaced with an `assert` statement, because it is never expected for that invariant to be false. (Fixed in `ebecd`)
4. `EasyStaking.sol`, `L429`: Should add the comment about how the special case where the function performs differently when `_amount` is `0`. (Fixed in `ebecd`)

Test Results

Test Suite Results

```
Contract: EasyStaking
  initialize
     ✓ should be set up correctly (64ms)
     ✓ fails if any of parameters is incorrect (1504ms)
  deposit
     ✓ should deposit (232ms)
     ✓ should accrue emission (462ms)
     ✓ should deposit using an old id (1846ms)
     ✓ fails if deposit value is zero (65ms)
     ✓ fails if wrong deposit id (62ms)
  onTokenTransfer
     ✓ should deposit (143ms)
     ✓ should accrue emission (598ms)
     ✓ should deposit using an old id (1442ms)
     ✓ fails if deposit value is zero (104ms)
     ✓ fails if not a token address (74ms)
  makeForcedWithdrawal
     ✓ should withdraw (712ms)
     ✓ should withdraw with accrued emission (335ms)
     ✓ should withdraw part and accrue emission (315ms)
     ✓ should accrue emission for different users from 1 address (1554ms)
     ✓ fails if trying to withdraw more than deposited (392ms)
     ✓ fails if wrong deposit id (238ms)
     ✓ fails if zero balance (215ms)
     ✓ should withdraw entire deposit by several parts (1693ms)
     ✓ should withdraw the same amount (726ms)
  requestWithdrawal
     ✓ should request (207ms)
     ✓ fails if wrong deposit id (58ms)
  makeRequestedWithdrawal
     ✓ should withdraw (550ms)
     ✓ should fail if not requested (140ms)
     ✓ should fail if too early (226ms)
     ✓ should fail if too late (199ms)
  totalStaked
     ✓ should be calculated correctly (2235ms)
  setFee
     ✓ should set (97ms)
     ✓ fails if not an owner (53ms)
     ✓ fails if greater than 1 ether (89ms)
  setWithdrawalLockDuration
     ✓ should set (208ms)
     ✓ fails if not an owner (154ms)
     ✓ fails if equal to zero (67ms)
  setWithdrawalUnlockDuration
     ✓ should set (112ms)
     ✓ fails if not an owner (80ms)
     ✓ fails if equal to zero (50ms)
  setTotalSupplyFactor
     ✓ should set (107ms)
     ✓ fails if not an owner (76ms)
     ✓ fails if greater than 1 ether (55ms)
  setSigmoidParameters
     ✓ should set (104ms)
     ✓ fails if not an owner (50ms)
     ✓ fails if wrong values (111ms)
  setLiquidityProvidersRewardAddress
     ✓ should set (235ms)
     ✓ fails if not an owner (161ms)
     ✓ fails if equal to zero (52ms)
     ✓ fails if equal to the address of EasyStaking contract (50ms)
  claimTokens
     ✓ should claim tokens (579ms)
     ✓ should claim STAKE tokens (602ms)
     ✓ should claim ether (239ms)
```

```
      ✓ should claim and send ether even if receiver reverts it (388ms)
      ✓ fails if not an owner (56ms)
      ✓ fails if invalid recipient (102ms)
    getSupplyBasedEmissionRate
      ✓ should be calculated correctly (343ms)
      ✓ can't be more than 7.5% (302ms)
    getAccruedEmission
      ✓ should be calculated correctly (306ms)
    ExtendedMath
      ✓ sqrt should be within the gas limit and calculated correctly (4775ms)
      ✓ sqrt of 0-3 (260ms)
      ✓ pow2 of 0 (58ms)


  59 passing (1m)
```

Code Coverage

The code is well covered by the tests with all the important branches being covered and extensive assertions.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	99.3	91.11	100	99.26	
EasyStaking.sol	99.3	91.11	100	99.25	382
IERC20Mintable.sol	100	100	100	100	
Sacrifice.sol	100	100	100	100	
contracts/lib/	96.67	87.5	100	100	
ExtendedMath.sol	100	87.5	100	100	
Sigmoid.sol	94.12	87.5	100	100	
contracts/mocks/	100	75	100	100	
ERC677Mock.sol	100	75	100	100	
EasyStakingMock.sol	100	100	100	100	
ExtendedMathMock.sol	100	100	100	100	
ReceiverMock.sol	100	100	100	100	
All files	**98.96**	**89.47**	**100**	**99.45**	

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

```
78d0f022f079afd6e8071e67758a15e0aa17b189dd2ca28fef7efdde051766aa ./contracts/EasyStaking.sol

01c571355f45ab0db20ded6eb4da8c90fefd19f48688fd1a7678b9fa33a091e6 ./contracts/Sacrifice.sol

4e4d16fea083d2fe783492c5f957dcda22ed4ec924f4e7a5157a2ed9019b632f ./contracts/IERC20Mintable.sol

fabca94e5245834ffd919a39a8b8e2919bda5bd5e9bbc02fadf9a9eba39e534b ./contracts/lib/ExtendedMath.sol

b98a37cf6e47d730fcc4f265f5920ce472f819c6218e76029a35c40fe6c5df59 ./contracts/lib/Sigmoid.sol

8441319cd35b7be0ac5d0a93bc5b840152358ccb664be6bd8b8c70eb0a366511 ./contracts/mocks/ERC677Mock.sol

3aeb4b12470f381631b29bd25293bb88a05d8838d9564a4b1a32a090b6b5a69c ./contracts/mocks/ExtendedMathMock.sol

f410676dc0a00f0125014fd138e47b76c61da3220314543392ee2060beb2a652 ./contracts/mocks/EasyStakingMock.sol

bffa38500954e66d804f3be47f277bd0f6b9b740af86993ee5f240728bbb3c73 ./contracts/mocks/ReceiverMock.sol
```

Tests

```
49c60ac3946139d1b96ab7b0fe4e40339ed62d24522fc5abaaeea7fad918a10c ./test/EasyStaking.test.js
```

Changelog

- 2020-07-27 - Initial report
- 2020-08-03 - Final report

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure blockchain platforms at scale using computer-aided reasoning tools, with a mission to help boost the adoption of this exponentially growing technology.

With over 1000 Google scholar citations and numerous published papers, Quantstamp's team has decades of combined experience in formal verification, static analysis, and software verification. Quantstamp has also developed a protocol to help smart contract developers and projects worldwide to perform cost-effective smart contract security scans.

To date, Quantstamp has protected $5B in digital asset risk from hackers and assisted dozens of blockchain projects globally through its white glove security assessment services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Quantstamp's collaborations with leading academic institutions such as the National University of Singapore and MIT (Massachusetts Institute of Technology) reflect our commitment to research, development, and enabling world-class blockchain security.