Quantstamp Smart Contract Audit

PoolTogether Contracts

This smart contract audit was prepared by Quantstamp, the protocol for securing smart contracts.

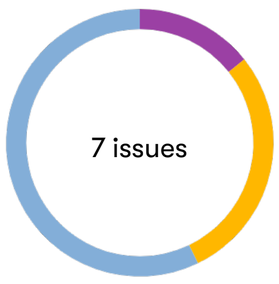
Quantstamp helps to secure blockchain applications such as smart contracts. We are developing a new protocol for smart contract verification, performing professional audits and consultations, and developing security tools. Quantstamp also has expertise in application security and secure software development.

Executive Summary

Type	Protocol Contracts
Auditors	Alex Murashkin, Senior Software Engineer Kacper Bąk, Senior Research Engineer Sung-Shine Lee, Research Engineer
Timeline	2019-06-07 through 2019-06-21
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	PoolTogether Pre-Audit Checklist Grant Request Checklist Code Comments and README
Source Code	

Repository	Commit
pooltogether-contracts	b302461
kleros (SortitionSumTreeFactory.sol)	605bee4
fixidity (FixidityLib.sol)	3547a29

Total Issues	**7** (5 Resolved)
High Risk Issues	0
Medium Risk Issues	**1** (1 Resolved)
Low Risk Issues	**2** (2 Resolved)
Informational Risk Issues	**4** (2 Resolved)
Undetermined Risk Issues	0



Overall Assessment

The code, overall, is well-written and documented, follows best practices and makes a good use of managed dependencies. However, we found one medium-severity issue as well as provided several findings of "low" and "informational" levels. In addition, we suggested a few improvements to dependency management and code documentation.

Most issues have been addressed by the PoolTogether team and marked as "fixed" in the report. Two of the findings were not regarded as issues by the PoolTogether team.

Severity Categories	
High	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	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	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.

Goals

• Evaluating `PoolTogether` contracts for security-related issues

• Evaluating `FixidityLib` and `SortitionSumTreeFactory` for security-related issues

Changelog

• 2019-06-14 - Initial Report

• 2019-06-21 - Audited the diff of `pooltogether-contracts` between the commits `b302461` and `530085b`

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

Setup

Tool Setup:

• Truffle v5.0.1

• Ganache v6.4.1

• Mythril v0.2.7

• Securify

Steps taken to run the tools:

1. Installed Truffle: `npm install -g truffle`
2. Installed Ganache: `npm install -g ganache-cli`
3. Installed the solidity-coverage tool (within the project's root directory): `npm install --save-dev solidity-coverage`
4. Ran the coverage tool from the project's root directory: `./node_modules/.bin/solidity-coverage`
5. Flattened the source code using `truffle-flattener` to accommodate the auditing tools.
6. Installed the Mythril tool from Pypi: `pip3 install mythril`
7. Ran the Mythril tool on each contract: `myth -x path/to/contract`
8. Ran the Securify tool: `java -Xmx6048m -jar securify-0.1.jar -fs contract.sol`

Assessment

Findings

Potential Bug in Functionality

Severity: *Medium*

Status: Fixed

Contract(s) affected: `Pool.sol`

Description: A lottery participant may decide to buy additional tickets by calling the `buyTickets(...)` method more than once. However, the line `Pool.sol, L121` sets the value to `totalDepositNonFixed` which does not depend on ticket purchases made by the user previously.

Exploit Scenario: 1. User buys ten tickets by calling `buyTickets(10)`

1. User would like to buy two additional tickets by calling `buyTickets(2)`
2. The final number of tickets recorded in the `SortitionSumTree` instance is 2, not 12
3. At the draw time, the user has less chances to win, which contradicts the specification.

Likewise, if user buys the same amount of tickets as the first time, they have no change in their chances to win the lottery, but get charged twice for the same number of tickets.

Recommendation: We recommend to either:

1. Fixing the smart contract logic to correctly account for previous purchases by passing the non-fixed value of `entries[msg.sender].amount` to `sortitionSumTrees.set(...)` as `_value` in `Pool.sol, L121`.
2. Limiting the number of calls to `buyTickets(...)` per user so that each participant can only call the method once.

Unlocked Dependency Versions

Severity: *Low*

Status: Fixed

Contract(s) affected: `Pool.sol, PoolManager.sol`

Description: Unlocked ("fuzzy") dependency versions in project's `package.json` may result in compiling the project using versions of dependencies that are different from the ones that were tested or audited.

Exploit Scenario: The reference `CementDAO/Fixidity#master` in `package.json` may result in the project using a version of `fixidity` that is not tested to be compatible or safe. Example: a `CementDAO/Fixidity` contributor updates the code on the `master` branch and introduces a different behaviour or a security vulnerability. Since the version is unlocked, upon a next `yarn install`, it could be unknowingly fetched by a `PoolTogether` developer and used for compiling the `PoolTogether` project.

Likewise, `openzeppelin-eth": "^2.1.3"` may result in installing a newer version of `openzeppelin-eth` (e.g., `2.1.14`), which may not necessarily be as compatible or safe as the original one.

Recommendation: In project's `package.json`, lock dependency versions, e.g.:

`"openzeppelin-eth": "2.1.3"`

For GitHub references, use a specific commit rather than a branch name, e.g.:

`"fixidity": "CementDAO/Fixidity#3547a290825d7e255ca2a4226b3ace5301f0f262",`

In order not to rely on an external repository, consider forking the upstream project and referencing the fork instead.

Centralization of Power

Severity: *Low*

Status: Fixed

Contract(s) affected: `Pool.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 depending on the level of privilege the contract allows to the owner.

Exploit Scenario: The pool owner has the privilege to lock and unlock the smart contract. If the pool owner loses the key, users have no way of getting back their contributions, therefore, the lottery can no longer be considered "lossless".

Recommendation: We recommend:

1. Potentially, introducing a mechanism to allow participants to withdraw once `lockEndBlock` has passed but without having to wait for the pool owner to unlock the pool.
2. Instead of having the sole `owner` account, decentralizing control by supporting a multi-signature M-of-N wallet.

Update: the PoolTogether team has taken the approach (1). This mitigates the issue since users can (under certain conditions) withdraw their contributions without relying on the pool owner to complete the pool. However, users need to be made aware of the following scenario. In case `allowLockAnytime` is set to `true` and `lockEndBlock` is set to a high value (e.g., several weeks or months), the pool owner can unlock the pool at any time while regular users have no ability to withdraw their funds unless they patiently wait for `lockEndBlock`, which can be unrealistic. The recommendation is to make users aware of this possibility and require pool owners to set `lockEndBlock` responsibly.

Unlocked Pragma

Severity: *Informational*

Status: Fixed

Contract(s) affected: `Pool.sol`, `PoolManager.sol`, `UniformRandomNumber.sol`

Description: Most Solidity files specify in the header a version number of the format `pragma solidity (^)0.5.0`. The caret (^) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked." For consistency and to prevent unexpected behavior in the future, it is recommended to remove the caret to lock the file onto a specific Solidity version.

Recommendation: Locking pragma versions, e.g., `pragma solidity 0.5.0`.

Gas Usage / `for` Loop Concerns

Severity: *Informational*

Contract(s) affected: `SortitionSumTreeFactory.sol`

Description: Gas usage is a main concern for smart contract developers and users, since high gas costs may prevent users from wanting to use the smart contract. Even worse, some gas usage issues may prevent the contract from providing services entirely. For example, if a `for` loop requires too much gas to exit, then it may prevent the contract from functioning correctly entirely.

Exploit Scenario: Typical scenarios include denial-of-service attacks when multiple users make too many small contributions, causing the sizes of data structures and loops to be too big, leading to exceeding block gas limit restrictions. While many of the computations in the `SortitionSumTree` data structure have logarithmic complexity which mitigates the issue, unbounded loops are generally not recommended in smart contracts.

Recommendation: Capping the number of contributors or having the ticket price large enough to de-incentivize having too many participants.

Update: The PoolTogether team is aware of the finding, however, does not consider this to be an issue because the number of users is expected to be small (up to ~`1400`), the logarithm of which is even smaller.

Denial-of-Service (DoS)

Severity: *Informational*

Contract(s) affected: `Pool.sol`

Description: A Denial-of-Service (DoS) attack is a situation which an attacker renders a smart contract unusable.

Exploit Scenario: The token provided in the constructor `Pool.sol, L74`, in spite of adherence to the `IERC20` interface, may contain malicious behavior, e.g., `transferFrom` may intentionally always return `false`. If that is the case, the method `buyTickets(...)` may become unusable.

Recommendation: The pool owner needs to verify the token contract to make sure it strictly adheres to the ERC20 specification and does not render the pool unusable.

Update: The PoolTogether team is aware of the finding, however, does not consider this to be an issue because the token address is specified once at the `PoolManager` contract deployment time and then passed into pools created using the given `PoolManager` instance.

Misusing `require()`/`assert()`/`revert()`

Severity: *Informational*

Status: Fixed

Contract(s) affected: `FixidityLib.sol`

Description: `require()`, `revert()`, and `assert()` all have their own specific uses and should not be switched around.

- `require()` checks that certain preconditions are true before a function is run.
- `revert()`, when hit, will undo all computation within the function.
- `assert()` is meant for checking that certain invariants are true. An `assert()` failure implies something that should never happen, such as integer overflow, has occurred.

Recommendation: Using `require(...)` for testing input parameters and `assert(...)` for validating invariants.

Test Results

Test Suite Results

Tests were run successfully. However, the command for running tests was not documented in the README.

```
  Contract: PoolManager
    createPool()
       ✓ should create a new pool (2478911 gas)
       ✓ should allow multiple pool creation (5220320 gas)
    setLockDuration()
       ✓ should update the lock duration (28434 gas)

  Contract: Pool
    supplyRateMantissa()
       ✓ should work (3216121 gas)
    currentInterestFractionFixedPoint24()
       ✓ should return the right value (3216121 gas)
    maxPoolSize()
       ✓ should set an appropriate limit based on max integers (3214265 gas)
    pool that is still open and must respect block start and end
      lock()
         ✓ cannot be locked before the open duration is over (97095 gas)
    pool that is still during the lock period
      complete(secret)
         ✓ cannot be unlocked before the lock duration ends (163541 gas)
    pool with zero open and lock durations
      buyTicket()
         ✓ should fail if not enough tokens approved (149848 gas)
         ✓ should deposit some tokens into the pool (402952 gas)
         ✓ should allow multiple deposits (557509 gas)
      getEntry()
         ✓ should return zero when there are no entries (72110 gas)
      lock()
         ✓ should transfer tokens to the money market (516839 gas)
      unlock()
         ✓ should allow anyone to unlock the pool (87645 gas)
         ✓ should allow the owner to unlock the pool (92439 gas)
        withdraw() after unlock
           ✓ should allow users to withdraw after the pool is unlocked (183336 gas)
      complete(secret)
         ✓ should transfer tokens from money market back (652022 gas)
         ✓ should succeed even without a balance (216933 gas)
      withdraw()
         ✓ should work for one participant (719840 gas)
         ✓ should work for two participants (1066584 gas)
      winnings()
         ✓ should return the entrants total to withdraw (403016 gas)
    when pool cannot be locked yet
      lock()
         ✓ should not work for regular users
         ✓ should support early locking by the owner (113823 gas)
    when pool cannot be unlocked yet
      complete(secret)
         ✓ should still work for the owner (135183 gas)
         ✓ should not work for anyone else
    when fee fraction is greater than zero
       ✓ should reward the owner the fee (3915769 gas)

·-------------------------------------------------------------------|-----------------------------·
|                                 Gas                               ·  Block limit: 10000038 gas  |
······································|·····································|·····························
|  Methods                           ·           10 gwei/gas           ·       285.46 usd/eth        |
················|·····················|···········|···········|···········|···············|·············
|  Contract      ·  Method            ·  Min      ·  Max      ·  Avg      ·  # calls      ·  usd (avg)  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  buyTickets        ·   108702  ·   286892  ·   262039  ·           10  ·       0.75  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  complete          ·    67430  ·   168084  ·   110933  ·            9  ·       0.32  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  initialize        ·        -  ·        -  ·    72110  ·           16  ·       0.21  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  lock              ·    65982  ·   113823  ·    96124  ·           10  ·       0.27  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  unlock            ·    87645  ·    92439  ·    90042  ·            2  ·       0.26  |
················|·····················|···········|···········|···········|···············|·············
|  Pool          ·  withdraw          ·    52818  ·    82818  ·    65488  ·            6  ·       0.19  |
················|·····················|···········|···········|···········|···············|·············
|  PoolManager   ·  createPool        ·  2451763  ·  2478911  ·  2469862  ·            3  ·       7.05  |
················|·····················|···········|···········|···········|···············|·············
|  PoolManager   ·  setLockDuration   ·        -  ·        -  ·    28434  ·            1  ·       0.08  |
················|·····················|···········|···········|···········|···············|·············
|  Token         ·  approve           ·    45727  ·    45855  ·    45803  ·           11  ·       0.13  |
················|·····················|···········|···········|···········|···············|·············
|  Deployments                       ·                                   ·  % of limit   ·             |
·····································|···········|···········|···········|···············|·············
|  CErc20Mock                        ·        -  ·        -  ·   432092  ·        4.3 %  ·       1.23  |
·····································|···········|···········|···········|···············|·············
|  Pool                              ·  3128691  ·  3159459  ·  3143131  ·       31.4 %  ·       8.97  |
·····································|···········|···········|···········|···············|·············
|  PoolManager                       ·        -  ·        -  ·  4423627  ·       44.2 %  ·      12.63  |
·····································|···········|···········|···········|···············|·············
|  Token                             ·        -  ·        -  ·   900778  ·          9 %  ·       2.57  |
·-----------------------------------|-----------|-----------|-----------|---------------|-------------·

  26 passing (1m)
```

Code Coverage

Overall, code coverage is good, however, certain lines of code are lacking coverage as per the table below.

One test was unable to run when measuring code coverage (`maxPoolSize() -> should set an appropriate limit based on max integers`), however, it does not impact coverage data.

```
----------------------------|----------|----------|----------|----------|----------------|
File                        |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------------|----------|----------|----------|----------|----------------|
 contracts/                 |    94.67 |    62.24 |     87.5 |     94.3 |                |
    Pool.sol                |    97.17 |    65.79 |    96.15 |     96.4 |251,472,473,474 |
    PoolManager.sol         |    89.47 |       50 |    69.23 |       90 |196,230,248,266 |
    Token.sol               |      100 |      100 |      100 |      100 |                |
    UniformRandomNumber.sol |    83.33 |       50 |      100 |    85.71 |             16 |
 contracts/compound/        |    91.67 |       50 |    83.33 |    91.67 |                |
    CErc20Mock.sol          |    91.67 |       50 |    83.33 |    91.67 |             25 |
    ICErc20.sol             |      100 |      100 |      100 |      100 |                |
----------------------------|----------|----------|----------|----------|----------------|
All files                   |    94.44 |    61.32 |    86.96 |    94.12 |                |
----------------------------|----------|----------|----------|----------|----------------|
```

Automated Analyses

Mythril

Mythril has reported potential multiplication and addition result overflows, as well as integer underflows in the function `queryLeafs(...)` of `SortitionSumTreeFactory`. These are deemed to be low risk in practice, since the number of elements will not be big enough, and the method does not change the contract state.

The expression `tree.nodes[0]` of `SortitionSumTreeFactory`'s `draw` method was flagged, however, in the context of the project, the tree should not be empty at the time of a draw and should not contain zero-value elements.

The `abs()` method of `FixidityLib` was flagged by Mythril as well, however, it is deemed to be a false-positive.

For `FixidityLib`, several lines were flagged as "reachable exceptions" and integer overflows or underflows. These were deemed as false-positives, however the code should favor `require(...)` over `assert(...)` for checking input data.

Securify

Securify has produced an output with several instances of the `LockedEther` vulnerability, however, upon manual inspections, these were deemed as false-positives.

Adherence to Specification

Status: Fixed

1. "... The more they purchase, the greater the chance they have of winning" seems to be violated in the potential bug scenario described above. See "Potential Bug in Functionality" for the details.

2. To match the comment "the total amount cannot exceed the max pool size" strictly, `Pool.sol, L126`, the sign needs to be "less than or equal".

3. There is a scenario when a pool can remain in the locked state indefinitely. In such a case, the lottery is not quite "lossless" since pool participants cannot withdraw their contributions. See the "Centralization of Power" note above for the details.

Code Documentation

Status: Fixed

1. `PoolManager.sol, L16`: some events and their fields are not documented, e.g., it is not clear what `page` should be like or units of `duration`.

2. Some methods do not have comments, such as `netWinningsFixedPoint24` and `grossWinningsFixedPoint24`.

3. `FixidityLib.sol, L71`: "Maximum" -> "Minimum".

Adherence to Best Practices

1. `Pool.sol, L104`, naming inconsistency: `countFixed` and `totalDeposit` are both using `FixidityLib`, however, `totalDeposit` doesn't have the `Fixed` suffix. At the same time, `totalDepositNonFixed` has the `NonFixed` suffix, which is inconsistent.

2. `Pool.sol`: unused declaration `OwnerWithdrawn(address indexed sender, int256 amount);`.

3. `PoolManager.sol, L27`: it is recommended to include units as a variable name component, e.g., `durationBlocks`. Most variables have that, but not all.

4. `Pool.sol, L31`: it is recommended to initialize important state variables in the constructor, e.g., the state `OPEN` is default because it is the first item in the `enum` list, however, if the order was changed, this would no longer be the case.

Update: the comments above have been addressed by the PoolTogether team. Additional findings (line numbers are as of the commit `530085b`):

1. `Pool.sol, L232`: the message should be `entrant does not exist` for consistency.
2. `Pool.sol, L350`: since entropy depends on the value of `secret`, the return value of `winnerAddress(...)` may vary depending on whether the pool is in the `UNLOCKED` or `COMPLETE` state. This may cause confusion in case users query the smart contract directly on Etherscan. Our recommendation is having the `winnerAddress(...)` method return the zero address unless the pool is in the `COMPLETE` state. Alternatively, computing and storing the winner address at the pool completion time and having `winnerAddress(...)` just return the stored value.

Appendix

File Signatures

The following are the SHA-256 hashes of the audited contracts and test files. A smart contract or file with a different SHA-256 hash has been modified, intentionally or otherwise, after the audit. 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 audit.

Contracts

```
6408860f78c177bc68870f6371ce7293c275fe5354005e0dd8e1e53fa9d49c9d
./contracts/FixidityLib.sol

c49446b8b5ebfd8a66d8a118fa8f50c37a41cc2325798366e6cebdf96743a015
./contracts/Migrations.sol

686e39cbe5f7fd29f44fa26308b3bc0570b8def194a4a78f3ac3979bb6f4a32e
./contracts/Pool.sol

839ccb38ce7c0954617239c3823fa3d62c4ea567fc2c13d19551569dc3d1b8b5
./contracts/PoolManager.sol

15a10e7920a2fa68216da2b63a354b829e8c82d156a1da74fdb15eaa1d5f38cc
./contracts/SortitionSumTreeFactory.sol

83d83600dd90553a4901088f695022eb69dfbc8b6a791ee3ef501fc51d3afb83
./contracts/Token.sol

7dfbf43361ab9dff7371a56a32ec6ef6440da27a2ed3b684cd3686d1f29f3788
./contracts/UniformRandomNumber.sol

6a7e07a5323a30bed3c3647de2d3a11264ac662481cbd219c9fca87ceccf5358
./contracts/compound/CErc20Mock.sol

cbba014401448dfc44f1538a46d314105776283fa9e74961d0248c150fa9dbfb
./contracts/compound/ICErc20.sol
```

Tests

```
4924b44d42f3167c7f2537fadf055295562b321eb12667fca4973800a3ab40be
./test/Pool.test.js

8c89da734b77f39d864da06058f4c34fdc569597091145adcfd2190686264061
./test/PoolManager.test.js
```

About Quantstamp

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

Quantstamp's team boasts decades of combined experience in formal verification, static analysis, and software verification. Collectively, our individuals have over 500 Google scholar citations and numerous published papers. In its mission to proliferate development and adoption of blockchain applications, Quantstamp is also developing a new protocol for smart contract verification to help smart contract developers and projects worldwide to perform cost-effective smart contract security audits.

To date, Quantstamp has helped to secure hundreds of millions of dollars of transaction value in smart contracts and has assisted dozens of blockchain projects globally with its white glove security auditing 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.

Finally, Quantstamp's dedication to research and development in the form of collaborations with leading academic institutions such as National University of Singapore and MIT (Massachusetts Institute of Technology) reflects Quantstamp's commitment to enable world-class smart contract innovation.

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp, Inc. (Quantstamp). Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on the website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The Solidity language itself and other smart contract languages remain under development and are subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity or the smart contract programming language, or other programming aspects that could present security risks. You may risk loss of tokens, Ether, and/or other loss. A report is not an endorsement (or other opinion) of any particular project or team, and the report does not guarantee the security of any particular project. A report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. To the fullest extent permitted by law, we disclaim all warranties, express or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked website, or any website or mobile application featured in any banner or other advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. You may risk loss of QSP tokens or other loss. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.