Quantstamp Security Assessment Certificate

November 18th 2020 — Quantstamp Verified

KeeperDAO Protocol

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

Executive Summary

Type	Defi
Auditors	Leonardo Passos, Senior Research Engineer Fayçal Lalidji, Security Auditor
Timeline	2020-10-29 through 2020-11-17
EVM	Muir Glacier
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	KeeperDAO's Wiki README.md in repository
Documentation Quality	Low
Test Quality	Medium

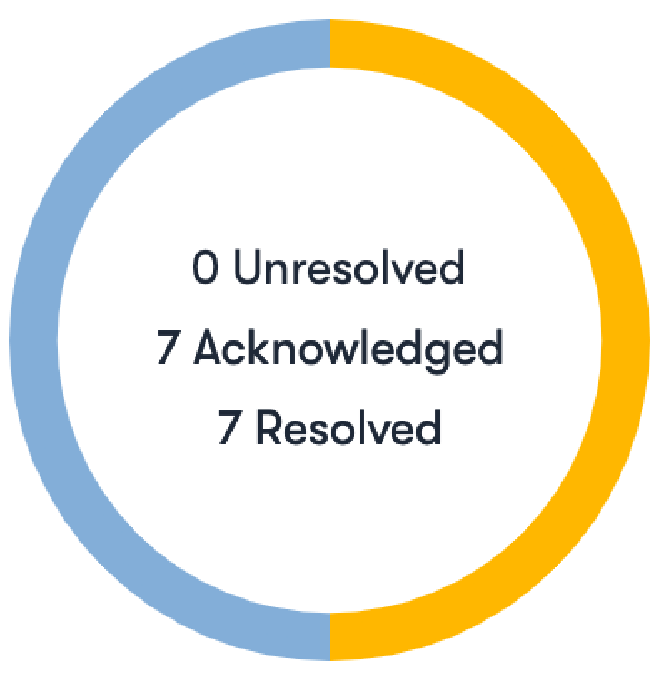
Source Code

Repository	Commit
Initial report	1622a24
Re-audit (1)	eb4e40d
Re-audit (2)	94dea5d

Goals

- Find issues that could lead to fund losses
- Find issues that could allow attacking liquidity pools
- Find inconsistencies between specification and code

Total Issues	14	(7 Resolved)
High Risk Issues	3	(1 Resolved)
Medium Risk Issues	1	(1 Resolved)
Low Risk Issues	4	(3 Resolved)
Informational Risk Issues	5	(1 Resolved)
Undetermined Risk Issues	1	(1 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

Overall, Quantstamp initially identified 14 issues, including high (3), medium (1), low (4), informational (5), and one undetermined severity issue. Among others, our auditing found that there is a risk that liquidity providers lose their funds upon deposit and currently, this is still unresolved. KeeperDAO acknowledged the issue, stating it will be fixed in a later release. Also, documentation still needs to be in synch with the code. Other less severe issues pending a fix include one low severity issue and 4 informational ones.

On the testing side of things, the branch coverage of the liquidity pool (V2) can and should be improved; currently, it is at 83%, but should be as close to 100% as possible.

Altogether, **KeeperDAO was advised not moving the present code to production prior to addressing the concerns herein presented.**

ID	Description	Severity	Status
QSP-1	Liquidity Providers Can Lose Their Deposit Amount	High	Acknowledged
QSP-2	Public Specification Is Outdated	High	Acknowledged
QSP-3	Borrowers Can Borrow Money And Not Pay Any Fees	High	Fixed
QSP-4	Unsafe ERC20 Function Call	Medium	Fixed
QSP-5	`LiquidityPoolV2` Initialization Lacks Input Sanitization	Low	Fixed
QSP-6	`updateFeePool` Lacks Input Sanitization	Low	Fixed
QSP-7	Operator Can Front Run User Transactions To Collect Higher Fees	Low	Acknowledged
QSP-8	Fees Can Be Set to Zero	Low	Fixed
QSP-9	Wrapped ETH Total Supply May Not Reflect The True Balance	Informational	Acknowledged
QSP-10	Allowance Double-Spend Exploit	Informational	Mitigated
QSP-11	Documentation Is Not In Natspec Format	Informational	Acknowledged
QSP-12	Clone-and-Own	Informational	Acknowledged
QSP-13	Privileged Roles and Ownership	Informational	Acknowledged
QSP-14	Token Renaming Is A Non-Standard ERC20 Operation	? Undetermined	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 Liquidity Providers Can Lose Their Deposit Amount

Severity: High Risk

Status: Acknowledged

File(s) affected: `protocol/LiquidityPoolV2.sol`

Description: Currently, the later in time a deposit is made, the lower the proportion of KTokens received is in relation to past deposits. However, due to the deposit fee and depending when a liquidity provider invokes the `deposit` function, the resulting return of KTokens may be zero. In such a case, liquidity providers may effectively lose their deposit amount.

Recommendation: The issue at hand occurs due to integer division in Solidity and truncating. As a simple solution, in `deposit`, check if `mintAmount` is zero, and if so, revert.

QSP-2 Public Specification Is Outdated

Severity: High Risk

Status: Acknowledged

File(s) affected: `protocol/LiquidityPoolV2.sol`

Description: KeeperDAO's public documentation is outdated. For instance, in the public Wiki the mint formula of KTokens has no reference to the deposits fee, nor that the proportion of minted KTokens decreases with time. Hence, based on the public specification alone users may be led to expect a specific behavior from the platform that does not occur in practice.

Recommendation: Thoroughly review the public specification (and whatever other public facing documentation available to users), asserting it fully adheres to the implemented code.

> Update: PR 15 addresses the issues stated in this audit. However, it did not provide any changes to the public docs, namely the `README.md` file and Wiki. Hence, this is issue is still not resolved.

QSP-3 Borrowers Can Borrow Money And Not Pay Any Fees

Severity: High Risk

Status: Fixed

File(s) affected: `protocol/LiquidityV2.sol`

Description: Currently, the borrow fee (fee for short, see L155) is any extra value paid back by borrowers in addition to what they had borrowed. From that value, the pool gets a percentage (the pool fee). However, if the borrow fee is zero, so is the pool fee.

Recommendation: Ensure that the pool fee is set to be a minimum percentage of the borrowed amount and that it is different from zero.

> Update: The issue herein reported is claimed by KeeperDAO to be intentional. The system is intentionally designed to allow borrows to return anything (including zero), because there is an external system that will create incentives to do so. Therefore, based on the clarification provided, we are marking this issue as fixed.

QSP-4 Unsafe ERC20 Function Call

Severity: Medium Risk

Status: Fixed

File(s) affected: `protocol/LiquidityV2.sol`

Description: The `deposit` function uses the ERC20 `transferFrom` in an unsafe way, as it does not check for its return value. Some old ERC20 tokens return `false` instead of reverting the transaction. Hence, if such a token is listed as an underlying asset, an attacker could receive KTokens, while depositing no underlying tokens. With those in hand, the attacker could then withdraw part or all the underlying tokens.

Exploit Scenario:

- `BadToken` (BT) is a fictitious ERC20 token that does not revert upon error; rather it returns `false`.
- `BT` gets registered as a token in `LiquidityV2`.
- Bob (attacker) does not have any BT, but invokes `deposit`, passing 100 as the underlying amount. The `deposit` function invokes the `transferFrom` function on BT, which returns `false`. The latter, however, is ignored by `deposit`, which proceeds to give Bob 90 kBT (assuming a 10% deposit fee).
- Joe has 100 BT and decides to put them in KeeperDAO. As such, he invokes `deposit`.
- Joe receives 81 kBT in return.
- Bob withdraws his 90 kBT. Now the liquidity pool does not have enough BT tokens to pay Joe.

Recommendation: Replace the `transferFrom` call on L104 with `safeTransferFrom`.

QSP-5 `LiquidityPoolV2` Initialization Lacks Input Sanitization

Severity: Low Risk

Status: Fixed

File(s) affected: `protocol/LiquidityV2.sol`

Description: The `initialize` function has no sanity check on the `_borrower` address. The latter could be `0x0` or an EOA address.

Recommendation: Check that `_borrower` is not `0x0`; also check that it is indeed a contract.

> Update: PR 15 does add a check to verify that `_borrower` is different from `0x0`, but lacks a check to verify that it is indeed a contract. Hence, this issue remains unresolved.

QSP-6 `updateFeePool` Lacks Input Sanitization

Severity: Low Risk

Status: Fixed

File(s) affected: `protocol/LiquidityV2.sol`

Description: The `updateFeePool` function has no sanity check on the `_newFeePool` address. If the latter is `0x0`, then collected fees will be effectively burnt.

Recommendation: Check that `_newFeePool` is not `0x0`.

QSP-7 Operator Can Front Run User Transactions To Collect Higher Fees

Severity: Low Risk

Status: Acknowledged

File(s) affected: `protocol/LiquidityPoolV2.sol`

Description: When users submit a transaction (e.g., `deposit` and `borrow`), the operator could front-run the latter and increase platform fees (e.g., `depositFeeInBips` and `poolFeeInBips`). Users will end-up paying more, as the fee they had assumed no longer applies when their transaction gets mined.

Recommendation: Update platform fees whenever there is little incoming traffic (transactions). Alternatively, add an invariant that update fee functions can only execute when the contract is paused.

QSP-8 Fees Can Be Set to Zero

Severity: Low Risk

Status: Fixed

File(s) affected: `protocol/LiquidityV2.sol`

Description: Both `updateDepositFee` and `updatePoolFee` require that the given input fee is greater than or equal to zero. It is unclear why the platform would allow zero fees, unless during specific promotional campaigns to attract users.

Recommendation: If zero fees are indeed allowed as a means to have promotional campaigns, we recommend documenting the code properly so code readers can have better context. If not, we suggest requiring any fee value greater than zero.

> Update: The issue herein reported is claimed by KeeperDAO to be intentional. Therefore, we are marking this issue as fixed.

QSP-9 Wrapped ETH Total Supply May Not Reflect The True Balance

Severity: Informational

Status: Acknowledged

File(s) affected: `protocol/WETH9.sol`

Description: The value returned by the `totalSupply` function in the `WETH9` contract can mismatch the real sum of all users' balances. This is possible due to: (i) ether can be forcibly sent to `WETH9` using a third-party contract that invokes `selfdestruct`, passing the address of `WETH9` as the target recipient. Upon invoking `selfdestruct`, the funds of the contract being destroyed will be transferred to `WETH9`, while bypassing the fallback function of the latter; and (ii) by the fact that `totalSupply` returns the address balance value using `address(this).balance`.

There is no direct implication at the moment of writing. However, depending on third party DApps or future use cases of KeeperDAO, this may become an issue.

Exploit Scenario: A attacker can deploy a contract, deposit some ether, and then call a function that triggers `selfdestruct`, bypassing the fallback function on the target address (`weth9_address`). For example:

```
contract Destructible {

    function deposit() external payable {  }

    function destruct(address payable weth9_address) public {
        selfdestruct(weth9_address);
    }
}
```

Hence, the resulting balance in `address(weth9_address).balance` will not reflect the sum of all users' balances.

Recommendation: The real total value held by the contract should be tracked by adding the `msg.value` to a state variable when depositing and removing the withdrawn value when withdraw is called.

> Update: `WETH9` is only used for testing purposes. As such, we are classifying this issue as rather Informational, instead of Low severity as initially stated in our first report.

QSP-10 Allowance Double-Spend Exploit

Severity: Informational

Status: Mitigated

File(s) affected: `contracts/protocol/KToken.sol`, `contracts/protocol/StandardToken.sol`

Description: As it presently is constructed, all contracts inheriting from `KToken` (`kEther`, `kWrappedEther`, `kUSDC`, `kWBTC`, `kBTC`, and `kDAI`) are vulnerable to the allowance double-spend exploit, as with any other ERC20 tokens. An example of an exploit goes as follows:

1. Alice allows Bob to transfer `N` amount of Alice's tokens (`N>0`) by calling the `approve()` method on `Token` smart contract (passing Bob's address and `N` as method arguments)
2. After some time, Alice decides to change from `N` to `M` (`M>0`) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve()` method again, this time passing Bob's address and `M` as method arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls the `transferFrom()` method to transfer `N` Alice's tokens somewhere
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer `N` Alice's tokens and will gain an ability to transfer another `M` tokens
5. Before Alice notices any irregularities, Bob calls `transferFrom()` method again, this time to transfer `M` Alice's tokens. The exploit (as described above) is mitigated

through use of functions that increase/decrease the allowance relative to its current value, such as `increaseAllowance` and `decreaseAllowance`.

Pending community agreement on an ERC standard that would protect against this exploit, we recommend that developers of applications dependent on `approve()` / `transferFrom()` should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value. Teams who decide to wait for such a standard should make these recommendations to app developers who work with their token contract.

Recommendation: As a means to mitigate the issue, provide functions to increase and decrease allowance (already the case).

QSP-11 Documentation Is Not In Natspec Format

Severity: *Informational*

Status: Acknowledged

Description: Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec) - see Solidity's official documentation. According to the latter, "It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI)". However, the existing contracts in KeeperDAO are not in Natspec.

Recommendation: Document all external and public functions using Natspec format.

QSP-12 Clone-and-Own

Severity: *Informational*

Status: Acknowledged

Description: The clone-and-own approach involves copying and adjusting open source code at one's own discretion. From the development perspective, it is initially beneficial as it reduces the amount of effort. However, from the security perspective, it involves some risks as the code may not follow the best practices, may contain a security vulnerability, or may include intentionally or unintentionally modified upstream libraries. Rather than the clone-and-own approach, a good industry practice is to use the Truffle framework for managing library dependencies (when possible). This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries.

Recommendation: We suggest documenting where the code has been taken from as a means to trace potential issues with the original source code, allowing one to merge fixes and patches.

QSP-13 Privileged Roles and Ownership

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/LiquidityV2.sol`

Description: The liquidity pool has an `operator` role designating 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 grants to the operator.
The privileges of the operator include the ability to:

- Update the pool deposit fee using LiquidityPoolV2 up to 100%.
- To pause the main function in LiquidityPoolV2.
- Migrate all the underlying funds to a specific contract.

Recommendation: Add public facing documentation explaining under what conditions `operatorOnly` functions are executed, how the private key of the operator is stored and kept safe, and the access policy to such a key. We recommend using a multisig wallet to control `operatorOnly` functions.

QSP-14 Token Renaming Is A Non-Standard ERC20 Operation

Severity: *Undetermined*

Status: Fixed

File(s) affected: `protocol/StandardToken.sol`

Description: The `StandardToken` contract provides a `rename` operation that changes the underlying name of the token. This is a non-standard operation. A token's name should be immutable. Allowing a token's name to change could potentially cause issues in exchanges storing token data linked to a token's name. This issue affects all `KToken` contracts, namely `kEther`, `kWrappedEther`, `kUSDC`, `kWBTC`, `kBTC`, and `kDAI`, as they all have `StandardToken` as a supertype.

Recommendation: Since `rename` is a non-standard operation, we recommend removing this function altogether.

Automated Analyses

Slither

Slither did not finish execution successfully. No result was produced as a consequence.

Adherence to Best Practices

- File `protocol/kRoles.sol` and the contract declared therein have names that do not follow uppercase convention as in other files. Same with `kEther`, `kWrappedEther`, `kUSDC`, `kWBTC`, `kBTC`, and `kDAI` in `protocol/KToken.sol` We recommend standardizing naming conventions across all files. **Not fixed**
- In `protocol/LiquidityPoolV2.sol`, `caluclatePoolFee` is misspelled. It should be `calculatePoolFee`.
- `Interf.sol` is not a descriptive name. We suggest a wiser choice (e.g., `Interfaces.sol`). **Fixed**
- The error message on L53, L65, L149, and L162 have `require` statements whose error messages mention the V1 protocol. Instead, they should be mentioning V2. **Not fixed**
- The names of the functions `updatePoolFee` and `updateFeePool` in `protocol/LiquidityV2.sol` are extremely confusing. We suggest renaming them to allow users to clearly differentiate them. For instance, consider renaming `updateFeePool` to `updateFeePoolAddress` (or `updateFeePoolCollector`). **Not fixed**

- In `protocol/KToken.sol`, a token contract exists for each underlying token. Instead of hard coding the `decimals` value, the latter can be taken directly from the token address, avoiding mistakes that could be introduced during further development and/or source code maintenance. **Not fixed**
- In `protocol/LiquidityPoolV2.sol`, the `initialize` function calls `Pausable.initialize(msg.sender)` twice, which is redundant. Remove the second call. **Not fixed**

Test Results

Test Suite Results

Overall, it seems the test suite covers many test cases. We did not see, however, tests assessing multiple deposits as a means to check the behavior of KToken distribution and potential edge cases. Quantstamp did perform a simulation of multiple deposits and we were able to uncover high risk issues. We suggest that KeeperDAO performs a similar approach.

```
Contract: SimpleLP
    ✓ Can get SimpleLP's Operator
    ✓ Should not register the same token twice (62ms)
    ✓ Should not deposit with an non registered token (50ms)
    ✓ Should receive correct amount of kTokens on first deposit to SimpleLP (231ms)
    ✓ Should receive correct amount of kTokens on first deposit to SimpleLP (83ms)
    ✓ Should receive correct amount of kTokens on second deposit to SimpleLP (183ms)
    ✓ Should not be able to send ETH during an ERC20 deposit (119ms)
    ✓ Should receive correct amount of kTokens on second deposit to SimpleLP (87ms)
    ✓ Should fail to deposit when the eth amount is inconsistent (61ms)
    ✓ Should not allow deposits when paused (108ms)
    ✓ Should not flash lend with an non registered token (43ms)
    ✓ Can flash lend wether (79ms)
    ✓ Can flash lend ether (66ms)
    ✓ Should not lend ether when paused (116ms)
    ✓ Should revert when the borrowerfn reverts (81ms)
    ✓ Can revert on an invalid flash loan (91ms)
    ✓ Should be able to migrate (606ms)
    ✓ Should be able to migrate even when the LP is paused (597ms)
    ✓ Can withdraw partial wEther from SimpleLP (159ms)
    ✓ Can withdraw partial ether from SimpleLP (142ms)
    ✓ Should not allow withdrawals when paused (129ms)
    ✓ Can withdraw all wEther from SimpleLP (139ms)
    ✓ Can withdraw all Ether from SimpleLP (143ms)
    ✓ Should not withdraw Ether from SimpleLP when there are no kTokens (51ms)
    ✓ Should successfully handle profit re-distribution when the profit can be evenly split (379ms)
    ✓ Should handle profit re-distribution when the balance profit cannot be evenly split (365ms)
    ✓ Non operator should not be able to migrate (189ms)
    ✓ Should not withdraw with an unregistered token (125ms)
    ✓ Should not be able to recover liquidity tokens (51ms)

 Contract: LiquidityPoolV2: Deposit With Fees
    ✓ Operator should be able to update deposit fee (62ms)
    ✓ A non-operator should not be able to update deposit fee
    ✓ Fee cannot exceed 10000 (38ms)
    ✓ Can deposit Ether with fees (2806ms)
    ✓ Can deposit ERC20s with fees (3901ms)

 Contract: LiquidityPoolV2: Borrow With Pool Fees
    ✓ Operator should be able to update deposit fee
    ✓ A non-operator should not be able to update deposit fee (40ms)
    ✓ Fee cannot exceed 10000 (42ms)
    ✓ Operator should be able to update fee pool (67ms)
    ✓ A non-operator should not be able to update deposit fee
    ✓ Should send the correct amount of eth to the fee pool when pool fee is 100% (168ms)
    ✓ Should send the correct amount of eth to the fee pool when pool fee is 40% (107ms)
    ✓ Should send the correct amount of eth to the fee pool when pool fee is 0% (108ms)
    ✓ Should send the correct amount of ERC20 tokens to the fee pool when pool fee is 100% (126ms)
    ✓ Should send the correct amount of ERC20 tokens to the fee pool when pool fee is 40% (137ms)
    ✓ Should send the correct amount of ERC20 tokens to the fee pool when pool fee is 60% (129ms)
    ✓ Should send the correct amount of ERC20 tokens to the fee pool when pool fee is 0% (126ms)


 46 passing (17s)
```

Code Coverage

Overall, code coverage appears low. However, one has to account the influence of lacking tests for `LiquidityPool.sol`, which is not relevant for this release of KeeperDAO. Rather, the bulk of tests should be in `LiquidityPoolV2.sol` (which is indeed the case). In there, we see 100% coverage for all metrics, except branch (83%). Hence, we suggest improving that value to as close as possible to 100%.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
common/	42.86	16.67	100	42.86	
CanReclaimTokens.sol	42.86	16.67	100	42.86	29,30,31,33
mock/	100	100	100	100	
MockToken.sol	100	100	100	100	
protocol/	62.71	53.26	74.47	62.92	
BorrowerProxy.sol	100	100	100	100	
Interf.sol	100	100	100	100	
KRoles.sol	100	100	100	100	
KToken.sol	100	100	100	100	
LiquidityPool.sol	0	0	0	0	… 176,177,179
LiquidityPoolV2.sol	100	83.33	100	100	
RookToken.sol	100	100	100	100	
StandardToken.sol	100	100	100	100	
All files	**63.54**	**51.02**	**77.78**	**63.73**	

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

```
a020b3d1ba0c5c9af1277f2a4fa4b0e97ae84140632d61b0f25fc138467710bb ./contracts/Migrations.sol
5af0f7bbe286d00041fe887a6719f348bbd38c864f2fb7f2dca8edf7a134419f ./contracts/common/WETH9.sol
898bc76f2d05e38069823dad29345276e396fc95a60c6fc0f15761c4daa92e07 ./contracts/common/CanReclaimTokens.sol
51883961f29c6d2841f3f2303b0ae0a413d816445cb53dfaf269d396d85a4b52 ./contracts/protocol/BorrowerProxy.sol
79fcb91b175006cdf721a6f99174ecb35cd11f3a1cb721a2d73fe17223dbb594 ./contracts/protocol/StandardToken.sol
44d98d4c4f7a8f636f19462612b9d58859dc11081bc62bef5bc8bbafa3b9cbda ./contracts/protocol/RookToken.sol
d5b870934c13339572b40630be052f45d4400dacb877efc6cf73b99fb2e2f145 ./contracts/protocol/LiquidityPoolV2.sol
7204e230f30e6e6e41049e17a20ccf1f4a84b346ff404f9b43486d643d5858f9 ./contracts/protocol/KToken.sol
e97758cb1dd11fe552e22c1842b2d107ca31516f792f912eaf0fd7386dbb9243 ./contracts/protocol/KRoles.sol
0fed2ab7ac542efed2b2150bffdde9d30145a76bc80cfe1c350991a99118ee55 ./contracts/protocol/Interf.sol
28290b798ba6f1a1a5ba3d544b565c64420030e3f28a8ebcd2b8fae192a7da56 ./contracts/protocol/LiquidityPool.sol
d65d244d4184ba92ac1a5616743b6934c10a6ad2f0425375e167c7308663e75a ./contracts/mock/MockMaliciousBorrower.sol
eb5d794307d58b32c3f0fd4d54b82aacb1a54317ada168653ffaab86e2635c70 ./contracts/mock/MockToken.sol
0ce526c2392b0e7e542022f87944df69778cdb21712e83729def5f6a1308eaa3 ./contracts/mock/MockHonestBorrower.sol
```

Tests

```
0ad235a3f5445cf264619d0dff8a58b55ab34507abff2f43336ceaa16353a261 ./test/LiquidityPoolV2.ts
ba70764086dbd31d7914f7247b37d5fd63111a5292807a47f1e80a32e8087ee7 ./test/LiquidityPoolV1.ts
41b4a99c547fb3d412728e9bc3038215540a35894b12d56785ada00823c65ae7 ./test/helper/logs.ts
21da8977dab78afb50defa8f5d817fdf9456e13bb2653cd1bbe4b50509e247e0 ./test/helper/testUtils.ts
```

Changelog

- 2020-10-31 - Initial report
- 2020-11-02 - Re-audit (1)
- 2020-11-17 - Re-audit (1)

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.