Quantstamp Contract Security Certificate

May 5th 2020 — Quantstamp Verified

Pie DAO

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

Executive Summary

Type	Pooling protocol
Auditors	Kacper Bąk, Senior Research Engineer Martin Derka, Senior Research Engineer Alex Murashkin, Senior Software Engineer
Timeline	2020-04-08 through 2020-04-30
EVM	Muir Glacier
Languages	Solidity, Javascript
Methods	Architecture Review, Unit Testing, Computer-Aided Verification, Manual Review
Specification	Pie DAO

Source Code

Repository	Commit
pie-proxy	ece218f
pie-smart-pools	a17fc73

Goals

- Can users' funds get locked up in the contract?
- Can users withdraw their share of underlying tokens?
- May low-level routines be exploited to steal funds from the contracts?

Changelog

- 2020-04-17 - Initial report
- 2020-04-28 - Report revised based on commits `96b4ccd` (for pie-proxy) and `677dc19` (for pie-smart-pools)
- 2020-04-30 - Added responses from the team.

Overall Assessment

The code contains a lot of assembly and low-level constructs. Both obfuscate the intent and make future development, potentially, more error-prone. We have found a few issues ranging from medium to undetermined severity. We have not found any high-severity issues, however. It is important that any external runtime dependencies do not cause the contracts to fail permanently. We recommend resolving the issues we pointed out and making use of regular Solidity constructions instead of low-level code unless there is a compelling reason not to. The scope of this audit was limited to the following files:

- pie-proxy repo: `PProxyPausable.sol`, `PProxy.sol`, `PProxyStorage.sol`;
- pie-smart-pools repo: `PCappedSmartPool.sol`, `PBasicSmartPool.sol`, `ReentryProtection.sol`, `PCToken.sol`, `PCTokenStorage.sol`.

Update: some of our findings were addressed as of commits `96b4ccd` (for pie-proxy) and `677dc19` (for pie-smart-pools). We recommend addressing all our findings before deploying the contracts into production.

Total Issues	**13**	(4 Resolved)
High Risk Issues	0	(0 Resolved)
Medium Risk Issues	**2**	(2 Resolved)
Low Risk Issues	**4**	(1 Resolved)
Informational Risk Issues	**5**	(1 Resolved)
Undetermined Risk Issues	**2**	(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.

Summary of Findings

ID	Description	Severity	Status
QSP-1	`init()` may be called multiple times upon incorrect initialization	^ Medium	Resolved
QSP-2	Denial-of-Service (DoS)	^ Medium	Resolved
QSP-3	The function `stringToBytes32()` may convert only part of the string	⌄ Low	Resolved
QSP-4	Functions do not check if arguments are non-zero	⌄ Low	Acknowledged
QSP-5	Gas Usage / `for` Loop Concerns	⌄ Low	Acknowledged
QSP-6	Centralization of Power	⌄ Low	Acknowledged
QSP-7	`setCap()` may emit an event when the cap does not change	○ Informational	Resolved
QSP-8	Unlocked Pragma	○ Informational	Acknowledged
QSP-9	Allowance Double-Spend Exploit	○ Informational	Acknowledged
QSP-10	Clone-and-Own	○ Informational	Acknowledged
QSP-11	Race Conditions / Front-Running	○ Informational	Acknowledged
QSP-12	`internalFallback()` may call address without any code and return garbage	? Undetermined	Acknowledged
QSP-13	`setCap()` may set the cap below `totalSupply`	? Undetermined	Acknowledged

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:

- Ganache
- SolidityCoverage
- Mythril
- Securify
- Slither

Steps taken to run the tools:

1. Installed Ganache: `npm install -g ganache-cli`
2. Installed the solidity-coverage tool (within the project's root directory): `npm install --save-dev solidity-coverage`
3. Ran the coverage tool from the project's root directory: `./node_modules/.bin/solidity-coverage`
4. Installed the Mythril tool from Pypi: `pip3 install mythril`
5. Ran the Mythril tool on each contract: `myth -x path/to/contract`
6. Ran the Securify tool: `java -Xmx6048m -jar securify-0.1.jar -fs contract.sol`
7. Installed the Slither tool: `pip install slither-analyzer`
8. Run Slither from the project directory `slither .`

Assessment

Findings

QSP-1 `init()` may be called multiple times upon incorrect initialization

Severity: *Medium Risk*

Status: Resolved

File(s) affected: `PBasicSmartPool.sol`

Description: The function `init()` does not check that `_bPool` is non-zero. Consequently, if `_bPool` is zero, the function may be called multiple times and overwrite other previously initialized fields.

Recommendation: We recommend adding a `require()` statement checking that `_bPool` is non-zero. Furthermore, upon contract deployment and initialization, we recommend verifying that all fields are set to the expected values, especially because the function `approveTokens()` approves tokens returned by `bPool`.

QSP-2 Denial-of-Service (DoS)

Severity: *Medium Risk*

Status: Resolved

File(s) affected: `PBasicSmartPool.sol`

Description: A Denial-of-Service (DoS) attack is a situation which an attacker renders a smart contract unusable. If `_pushUnderlying()` fails entirely when any of the token transfers fails, users won't be able to withdraw any of their tokens. One possible scenario when the failure occurs is when `_pushUnderlying()` attempts to transfer 0 `_amount`.

Recommendation: We recommend redesigning the contract in such a way that allows users to pull their tokens individually, i.e., without requiring a loop. Furthermore, for `_amount` equal 0, do not attempt to do the transfer.
Update: the issue is resolved thru the use of function `exitPoolTakingloss()`.

QSP-3 The function `stringToBytes32()` may convert only part of the string

Severity: *Low Risk*

Status: Resolved

File(s) affected: `PProxyStorage.sol`

Description: The function `stringToBytes32()` converts only the first 32 bytes of string to `bytes32`. For longer strings, the remaining part will not be converted.

Recommendation: We recommend documenting the behavior of the function. Furthermore, we recommend adding a check to ensure that the converted string can fit into 32 bytes.

QSP-4 Functions do not check if arguments are non-zero

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `PProxy.sol`, `PBasicSmartPool.sol`

Description: The functions `PProxy.setProxyOwner()`, `PProxy.setImplementation()`, `PBasicSmartPool.setController()`, `PBasicSmartPool.setPublicSwapSetter()`, `PBasicSmartPool.setTokenBinder()` do not check that arguments of type `address` have non-zero values.
Update: the team informed us that the privileged addresses may be set to 0x0 when all external control is removed from the contracts.

Recommendation: We recommend adding `require()` statements that check if arguments are non-zero.

QSP-5 Gas Usage / `for` Loop Concerns

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `PBasicSmartPool.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. Iteration over `tokens` occurs in lines: 87, 158, 185, and 270.
Update: the team informed us that the underlying Balancer pool has a hard limit of 8 tokens. Consequently, iteration over the loops is was below the block gas limit. Currently joining a 4 token pool costs 618,069 gas and exiting costs 1,150,576 gas, i.e., well within the 10M gas limit. If in a future upgrade this becomes problematic, the team can upgrade the contract through DAO governance to mitigate this potential issue.

Recommendation: We recommend performing gas analysis to find out the limits for which the iteration succeeds. Although, perhaps, unlikely, too many tokens may cause block gas limit exceeded error.

QSP-6 Centralization of Power

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `PBasicSmartPool.sol`

Description: Smart contracts will often have `owner` variables to designate the person with special privileges to make modifications to the smart contract. Specifically, the contracts may feature centralization of power via the following roles: controller, swap setter, and token binder. All these parties are assumed to be trusted.

Recommendation: This centralization of power needs to be made clear to the users, especially depending on the level of privilege the contract allows to the privileged roles.
Update: the team informed us that they will document this contract characteristic.

QSP-7 `setCap()` may emit an event when the cap does not change

Severity: *Informational*

Status: Resolved

File(s) affected: `PCappedSmartPool.sol`

Description: The function `setCap()` emits event `CapChanged` even if one passes cap that is equal to `lpcs().cap`. The event is emitted despite the fact that no cap change happens.

Recommendation: We recommend adding a check that the new cap is different from the old cap.
Update: the team informed us that it is not an issue.

QSP-8 Unlocked Pragma

Severity: *Informational*

Status: Acknowledged

File(s) affected: `PProxyStorage.sol`, `PProxy.sol`, `PProxyPausable.sol`, `ReentryProtection.sol`, `PCTokenStorage.sol`, `PCToken.sol`, `PBasicSmartPool.sol`, `PCappedSmartPool.sol`

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.4.*`. 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".

Recommendation: 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.
Update: the issue appears to be fixed in pie-smart-pools as of commit `677dc19`. The team informed us that they will update pie-proxy contracts as well. Furthermore, compiler is locked in the builder config.

QSP-9 Allowance Double-Spend Exploit

Severity: *Informational*

Status: Acknowledged

File(s) affected: `PCToken.sol`

Description: As it presently is constructed, the contract is vulnerable to the allowance double-spend exploit, as with other ERC20 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`.

Exploit Scenario:

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.

Recommendation: 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.

QSP-10 Clone-and-Own

Severity: *Informational*

Status: Acknowledged

File(s) affected: `PCToken.sol`

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.
Specifically, `PCToken.sol` contains a modified clone of OpenZeppelin's `SafeMath`. Furthermore, the clone contains custom logic.

Recommendation: Rather than the clone-and-own approach, a good industry practice is to use the Truffle framework for managing library dependencies. This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries.
We recommend keeping the custom logic apart from the standard `SafeMath` code. Furthermore, we recommend documenting this custom logic.
Update: the team informed us that they intentionally use the same Math functions as Balancer to avoid having any discrepancies between the smart pool and underlying balancer pool.

QSP-11 Race Conditions / Front-Running

Severity: *Informational*

Status: Acknowledged

File(s) affected: `PCappedSmartPool.sol`

Description: A block is an ordered collection of transactions from all around the network. It's possible for the ordering of these transactions to impact the end result of a block. Specifically, there is a potential race condition between the functions `joinPool()` and `setCap()` due to the modifier `withinCap`.

Recommendation: Race conditions are typically benign issues and are difficult to eliminate. We recommend informing users about the race condition between the functions `joinPool()` and `setCap()`.
Update: they team will document this behavior and will create a proxy contract which can be used to add/remove liquidity to/from any of the PieDAO smart pools. The `setCap()` function is only called by the PieDAO would not be prone to front-running by malicious actors.

QSP-12 `internalFallback()` may call address without any code and return garbage

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `PProxy.sol`

Description: The function `internalFallback()` does not check if `contractAddr` contains any code. Furthermore, the function may return garbage if data returned from the internal call is shorter than the `calldata`, since the function overwrites memory pointed by `ptr` without clearing this memory first. A similar set of deficiencies lead to a temporary shutdown of 0x exchange.

Recommendation: We recommend checking if `contractAddr` contains any code. Furthermore, instead of overwriting, we recommend writing the returned data into a fresh memory slot. Alternatively, clear the memory pointed by `ptr` before overwriting it.
Update: the team informed us that they will not set implementation to non code containing addresses.

QSP-13 `setCap()` may set the cap below `totalSupply`

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `PCappedSmartPool.sol`

Description: The function `setCap()` may set the cap that is below `totalSupply`.

Recommendation: Since cap cannot really be lower than `totalSupply`, we recommend disallowing such caps.
Update: the team informed us that this is a feature to effectively only allow withdrawals from the smart pools.

Automated Analyses

Mythril

Mythril reported:

- the use of `delegatecall()` in `PProxy`.
- integer overflows in `PCappedSmartPool.sol`. Upon closer inspection we classified them as false positives.

Securify

Securify did not report any issues.

Slither

Slither reported that:

- `PProxy` has a payable fallback function, but it does not have a function to withdraw the ether.
- `BasicSmartPool.approveTokens()` ignores return value by external call in line 88. We recommend adding a `require()` statement; furthermore, we recommend adding a function that would approve a single token in case the whole loop fails.
- `BasicSmartPool` ignores return values by external calls in lines 204, 205, 225, 226, 234, 251. We recommend adding `require()` statements.

Code Documentation

Some of the important pieces of code are not well-documented. For example, `PProxyStorage.sol` shall better document `bytes32ToString()`.
Update: resolved (function removed).

Adherence to Best Practices

The code generally adheres to best practices, however,

- the code uses a lot of assembly and low-level constructions which obfuscates the intents and makes it, potentially, more error-prone.
- `PBasicSmartPool`, L204, 225, ignors return value from `transferFrom()`. **Update:** resolved.
- naming could use some improvement. For example, `_denorm` could be `denormalizedWeight`. There should be no shortening of names, e.g., `bPool` => `balancePool`. There is no tax on the number of characters in Solidity.
- `uint256(-1)` should be defined as a constant. It is used in `PBasicSmartPool.sol`, L88, L205, L226.
- in `PBasicSmartPool.sol`, L154, L178, `require()` should specify a reason as a second parameter, to make it clear why a certain condition is required.
- function `PBasicSmartPool.setController()` is marked as `noReentry`, but the other setters are not. This is inconsistent. The functions can be reentered at all. **Update:** resolved.
- for consistency, joining pool should be protected against reentrancy as well. Exiting is. If you expect to work with tokens that contain callbacks, both should be protected. However, the usage seems safe. **Update:** resolved.

Test Results

Test Suite Results

We noted that the file `pBasicSmartPool.ts` contains a bunch of TODO items. We recommend adding the remaining checks.

```
PProxy
  Ownership
     ✓ Owner should be msg.sender
     ✓ Calling setProxyOwner by a different address than the proxyOwner should fail
     ✓ Setting the proxy owner should work (129ms)
  Setting the implementation
     ✓ Setting the implementation contract should work
     ✓ Setting the implementation contract from a non owner should fail
```

```
  Delegating calls to implementation contract
    ✓ Calls should be delegated to implementation contract (200ms)

PProxyPausable
  Pauzer
    ✓ Pauzer should be deployer of the contract
    ✓ Setting the proxy pauzer should work
    ✓ Setting the pauzer from a non proxy owner address should fail
    ✓ Renouncing pauzer should work (87ms)
    ✓ Renouncing pauzer from a non pauzer address should fail
  Pauzing
    ✓ Pausing the contract should work
    ✓ Pausing the contract from a non pauzer address should fail
    ✓ Calling a function when not paused should work (184ms)
    ✓ Calling a function when paused should fail (52ms)
    ✓ Calling a function after unpausing should work (203ms)


16 passing (2s)

PBasicSmartPool
  init
    ✓ Initialising with invalid bPool address should fail (46ms)
    ✓ Initialising with zero supply should fail (42ms)
    ✓ Token symbol should be correct
    ✓ Token name should be correct
    ✓ Initial supply should be correct
    ✓ Controller should be correctly set
    ✓ Public swap setter should be correctly set
    ✓ Token binder should be correctly set
    ✓ bPool should be correctly set
    ✓ Tokens should be correctly set (66ms)
    ✓ calcTokensForAmount should work
    ✓ Calling init when already initialized should fail
    ✓ Smart pool should not hold any non balancer pool tokens after init (153ms)
  Controller functions
    ✓ Setting a new controller should work
    ✓ Setting a new controller from a non controller address should fail
    ✓ Setting public swap setter should work
    ✓ Setting public swap setter from a non controller address should fail
    ✓ Setting the token binder should work
    ✓ Setting the token binder from a non controller address should fail
    ✓ Setting public swap should work (110ms)
    ✓ Setting public swap from a non publicSwapSetter address should fail
    ✓ Setting the swap fee should work (107ms)
    ✓ Setting the swap fee from a non controller address should fail
  Joining and Exiting
    ✓ Adding liquidity should work (2117ms)
    ✓ Adding liquidity when a transfer fails should fail (85ms)
    ✓ Adding liquidity when a token transfer returns fails should fail (47ms)
    ✓ Removing liquidity should work (2411ms)
    ✓ Removing all liquidity should fail
    ✓ Removing liquidity should fail when removing more than balance (2260ms)
    ✓ Removing liquidity when a token transfer fails should fail
    ✓ Removing liquidity when a token transfer returns false should fail (38ms)
    ✓ Removing liquidity leaving a single token should work (1786ms)
    ✓ Removing all liquidity leaving a single token should fail
    ✓ Removing liquidity leaving a single token should fail when removing more than balance (2096ms)
  Token binding
    ✓ Binding a new token should work (393ms)
    ✓ Binding a token when transferFrom returns false should fail (87ms)
    ✓ Binding from a non token binder address should fail (74ms)
    ✓ Rebinding a token should work (375ms)
    ✓ Rebinding a token reducing the balance should work (347ms)
    ✓ Rebinding a token reducing the balance when the the token token transfer returns false should fail
    ✓ Rebinding a token from a non token binder address should fail
    ✓ Unbinding a token should work
    ✓ Unbinding a token from a non token binder address should fail
  ready modifier
    ✓ should revert when not ready (44ms)

PCappedSmartPool
  ✓ Cap should initially zero
  ✓ Setting the cap should work
  ✓ Setting the cap from a non controller address should fail
  ✓ JoinPool with less than the cap should work (2457ms)
  ✓ JoinPool with more than the cap should fail (195ms)

PCToken
  token metadata
    ✓ Should have 18 decimals
    ✓ Token name should be correct
    ✓ Symbol should be correct
    ✓ Initial supply should be zero
    ✓ After minting total supply should go up by minted amount (159ms)
    ✓ Burning tokens should lower the total supply (158ms)
    ✓ Burning more than an address's balance should fail (79ms)
  balanceOf
    ✓ Should return zero if no balance
    ✓ Should return correct amount if account has some tokens (79ms)
  transfer
    ✓ Should fail when the sender does not have enought balance (83ms)
    ✓ Sending the entire balance should work (156ms)
    ✓ Should emit transfer event (144ms)
    ✓ Sending 0 tokens should work (100ms)
  approve
    ✓ Should emit event
    ✓ Should work when there was no approved amount before
    ✓ Should work when there was a approved amount before
    ✓ Setting approval back to zero should work (60ms)
  increaseApproval
    ✓ Should emit event
    ✓ Should work when there was no approved amount before
```

```
      ✓ Should work when there was an approved amount before
      ✓ Increasing approval beyond max uint256 should fail
    decreaseApproval
      ✓ Should emit event (58ms)
      ✓ Decreasing part of the approval should work
      ✓ Decreasing the entire approval should work (57ms)
      ✓ Decreasing more than the approval amount should set approval to zero (52ms)
    transferFrom
      ✓ Should emit event (86ms)
      ✓ Should work when sender has enough balance and approved spender (103ms)
      ✓ Should fail when not enough allowance is set
      ✓ Should fail when sender does not have enough balance
      ✓ Should not change approval amount when it was set to max uint256 (84ms)

  PProxiedFactory
    ✓ Creating a new proxied pool should work (1715ms)

  ReentryProtection
    ✓ Should prevent reentry


  81 passing (3m)
```

Code Coverage

The first table shows coverage for the repo `pie-proxy`. The second for `pie-smart-pools`. Overall, the contracts feature reasonable coverage. We recommend adding test cases to cover else branches (negative scenarios) of the contract `PBasicSmartPool.sol`. Exercising negative scenarios may lead to discovering unintended interactions and outcomes.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
PProxy.sol	100	100	100	100	
PProxyPausable.sol	100	100	100	100	
PProxyStorage.sol	100	100	100	100	
contracts/interfaces/	100	100	100	100	
IPProxyOverrides.sol	100	100	100	100	
contracts/test/	50	100	50	50	
TestImplementation.sol	50	100	50	50	10,14
All files	**93.75**	**100**	**93.1**	**94.74**	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	97.33	70	96.3	97.53	
Math.sol	95.45	56.25	100	95.45	23
Ownable.sol	80	50	75	85.71	26
PCToken.sol	100	90	100	100	
PCTokenStorage.sol	100	100	100	100	
ReentryProtection.sol	100	100	100	100	
contracts/factory/	100	50	100	100	
PProxiedFactory.sol	100	50	100	100	
contracts/interfaces/	100	100	100	100	
IBFactory.sol	100	100	100	100	
IBPool.sol	100	100	100	100	
IERC20.sol	100	100	100	100	
IKyberNetwork.sol	100	100	100	100	
IPSmartPool.sol	100	100	100	100	
IUniswapExchange.sol	100	100	100	100	
IUniswapFactory.sol	100	100	100	100	
contracts/smart-**pools/**	100	81.82	100	100	
PBasicSmartPool.sol	100	80.95	100	100	
PCappedSmartPool.sol	100	100	100	100	
contracts/test/	100	100	100	100	
TestPCToken.sol	100	100	100	100	
TestReentryProtection.sol	100	100	100	100	
All files	**99.19**	**76.32**	**98.63**	**99.23**	

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

```
7675534ef96767c2120b352596749f602a422d70efc2981789fbeec72475c901 ./contracts/PProxyPausable.sol

f6ae5334aafea931ff0b03465f459f6c67974694c0a44d9d6d1dae182372236f ./contracts/PProxyStorage.sol

aadd4e7de1664a8bb1b88bfa31bc288bd36f8bfcd1eb0fca212049dc8e798d00 ./contracts/PProxy.sol

3b9ec77d90b8dd639dd5cf1dffe16713be110d740c267f2b1a7ac921dd06fb98 ./contracts/interfaces/IPProxyOverrides.sol

98e1216b45f73040cf1001ea182919a42f49acfeba04bcdf9b8169f0def52f39 ./contracts/test/TestImplementation.sol
```

Tests

```
b514148e53f2850db8640ec90c73a8184e5f54af5fcfa762135552144ca36c0d ./test/pProxy.ts

8f81153d9198a5a66d19d7f948f83cecfa3e824e0f98115304dbe17428db7e56 ./test/pProxyPausable.ts
```

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.