Quantstamp Security Assessment Certificate

November 4th 2021 — Quantstamp Verified

Sommelier Finance

This audit report was prepared by Quantstamp, the leader in blockchain security.

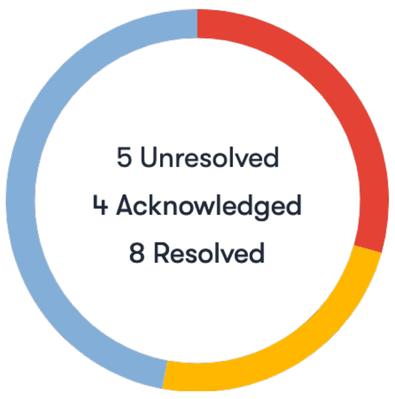
Executive Summary

Type	DeFi protocol
Auditors	Jake Bunce, Research Engineer Jan Gorzny, Blockchain Researcher Hisham Galal, Research Engineer
Timeline	2021-10-21 through 2021-11-04
EVM	London
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Documentation Quality	Low
Test Quality	Medium

Source Code

Repository	Commit
cellars	b887510
cellars	09d2e7b

Total Issues	**17** (8 Resolved)
High Risk Issues	**3** (2 Resolved)
Medium Risk Issues	**5** (4 Resolved)
Low Risk Issues	**4** (1 Resolved)
Informational Risk Issues	**1** (0 Resolved)
Undetermined Risk Issues	**4** (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

Quantstamp has reviewed the Sommelier Finance code base, which is a DeFi protocol using Uniswap V3. Although ambitious, the code suffers from a lack of clarity, an excessive amount of code repetition, and a failure to adhere to best practices. Our review found 17 issues, which cover the range of severity for our reports. In some cases, these are likely by design (e.g., QSP-12 - the use of `for`-loops), but in other cases, the source of the issues are less clear (e.g., QSP-17 - no deadlines, or QSP-6, an incomplete function). Quantstamp strongly recommends refactoring the code base to increase readability and maintenance, as well as providing useful, detailed documentation, so that the intent of the code can be clearly understood. Moreover, as Quantstamp was unable to compute the code coverage for the provided tests, we also recommend documenting the tests or providing instructions to compute coverage. These measures should be considered required before this protocol is deploy in a production setting.

ID	Description	Severity	Status
QSP-1	Inconsistent Implementations for Liquidity Addition	High	Fixed
QSP-2	Integer Overflow / Underflow	High	Unresolved
QSP-3	Multiple Instances of Reentrancy	High	Fixed
QSP-4	Greedy Contract	Medium	Fixed
QSP-5	Management Function Argument Validation	Medium	Fixed
QSP-6	Incomplete Function	Medium	Fixed
QSP-7	Code Duplication	Medium	Fixed
QSP-8	`amountOutMin` is Set to Zero	Medium	Unresolved
QSP-9	Unlocked Pragma	Low	Fixed
QSP-10	Own Implementation of Reentrancy Guard	Low	Acknowledged
QSP-11	Privileged Roles and Ownership	Low	Acknowledged
QSP-12	Gas Usage / `for` Loop Concerns	Low	Acknowledged
QSP-13	Clone-and-Own	Informational	Unresolved
QSP-14	Unclear Operation/Lack of Documentation	Undetermined	Unresolved
QSP-15	Clarify Flash Loan Protection	Undetermined	Unresolved
QSP-16	`addLiquidityForUniV3` Accepts Too Much	Undetermined	Acknowledged
QSP-17	No Deadlines	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 v0.6.6

Steps taken to run the tools:

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

Findings

QSP-1 Inconsistent Implementations for Liquidity Addition

Severity: *High Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: `addLiquidityForUniV3()` is inconsistent between `CellarPoolShare.sol` and `CellarPoolShareLimitETHUSDT.sol` & `CellarPoolShareLimitUSDCETH` with the total supply and pricing.

Recommendation: Move to a single function to remove code duplication and clarify why there are limits for some assets but not others.

QSP-2 Integer Overflow / Underflow

Severity: *High Risk*

Status: Unresolved

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: Integer overflow/underflow occur when an integer hits its bit-size limit. Every integer has a set range; when that range is passed, the value loops back around. A clock is a good analogy: at 11:59, the minute hand goes to 0, not 60, because 59 is the largest possible minute.
Integer overflow and underflow may cause many unexpected kinds of behavior and was the core reason for the `batchOverflow` attack. Here's an example with `uint8` variables, meaning unsigned integers with a range of `0..255`.

```
```
function under_over_flow() public {
    uint8 num_players = 0;
    num_players = num_players - 1; // 0 - 1 now equals 255!
    if (num_players == 255) {
        emit LogUnderflow();   // underflow occurred
    }
    uint8 jackpot = 255;
    jackpot = jackpot + 1;   // 255 + 1 now equals 0!
    if (jackpot == 0) {
        emit LogOverflow();   // overflow occurred
    }
}
```
```

Recommendation: Use the OpenZeppelin library `SafeMath` with version < 0.8.0 of Solidity, or move to Solidity version >= 0.8.0 where such protection is built into the compiler.

Update: Partially resolved, however there are still outstanding occurrences of arithmetic operations that do not use the SafeMath library and are therefore susceptible to integer over/underflow conditions.

QSP-3 Multiple Instances of Reentrancy

Severity: *High Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: Functions `invest()`, `invest()`, and `invest()` (i.e., the same function in each file) are all susceptible to reentrancy.

Recommendation: Implement reentrancy protection for this function. Suggested to use an open source library that is well audited and enjoys a deployment from a number of projects, such as this one.

QSP-4 Greedy Contract

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`

Description: A greedy contract is a contract that can receive ether which can never be redeemed.

Recommendation: Prevent Ether from being deposited into the contract by adding a `revert()` in the fallback function.

QSP-5 Management Function Argument Validation

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`

Description: Functions `setValidator()`, `transferOwnership()`, `setManagementFee()`, and `setPerformanceFee()` do not check for the validity of the address argument is sound.

Moreover, there are functions that change contract state don't emit events: `setValidator()`, `transferOwnership()`, `setManagementFee()`, and `setPerformanceFee()`.

Finally, the return values from `transfer()`, `approve()`, and `transferFrom()` is always `true` instead of returning the correct values from `_transfer()`, `_approve()`, and `_transferFrom()`, respectively.

Recommendation: Add various `require` statements to ensure that the right values are used, or at least that incorrect values (like the zero address) are not used. Consider emitting events on important state changes

QSP-6 Incomplete Function

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`

Description: `_beforeTokenTransfer()` appears to be an incomplete function.

Recommendation: Implement the function or clarify the intention behind this function.

QSP-7 Code Duplication

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: `CellarPoolShareLimitETHUSDT.sol` and `CellarPoolShareLimitUSDCETH.sol` are two large files with about 1076 lines of code each. They share the entire code except at three lines only: 45, 123, and 125. Also, `CellarPoolShare.sol` shares almost the entire code where the major difference is the code comments. Code duplication is dangerous because there is a chance that one copy of the code will be updated without further updating the other instances of the same code

Recommendation: Remove code duplication by creating a base contract that has all shared functionality, then inherit from it, and add the minimal overidden functions/assignments.

QSP-8 `amountOutMin` is Set to Zero

Severity: *Medium Risk*

Status: Unresolved

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: As per https://docs.uniswap.org/protocol/guides/swaps/single-swaps, "this is a significant risk in production. For a real deployment, this value should be calculated using our SDK or an on-chain price oracle - this helps protect against getting an unusually bad price for a trade due to a front running sandwich or another type of price manipulation."
This occurs on lines 335 and 367 of `CellarPoolShare.sol`, for example.

Recommendation: Predict a value using e.g., an oracle.

Update: We're unable to validate this fix as there's still an outstanding issue (QSP-2) to resolve that affects the implementation of the mitigation.

QSP-9 Unlocked Pragma

Severity: *Low Risk*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`, `interfaces.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.

QSP-10 Own Implementation of Reentrancy Guard

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `CellarPoolShare.sol`

Description: Reentrancy protection has been defined via a custom implementation.

Recommendation: Consider using a library that is well audited and enjoys wide deployment instead of rolling your own. Further reading.

QSP-11 Privileged Roles and Ownership

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: Smart contracts will often have `owner` variables to designate the person with special privileges to make modifications to the smart contract.
In this case, the functions `setValidator`, `transferOwnership`, `transferOwnership`, `setManagementFee`, `setPerformanceFee`, and `rebalance` can only be executed by the `owner`. Such privileges rely on some level of centralization, which poses an attack vector if:
The private key of a privileged actor is stollen, or; The privileged actor is malicious and attempts to make contracts work in an expected manner, or; The privileged actor is not malicious, but performs an incorrect state change in the given contracts (e.g., by calling a set-like function) such that the platform misbehaves, making it misbehave or be prone to attacks.

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 owner. This can also be mitigated by ensuring that privileged actors rely on multisig wallets, or governance smart contract that takes actions based on approved proposals from a DAO community.

QSP-12 Gas Usage / `for` Loop Concerns

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.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. It is best to break such loops into individual functions as possible.
The following functions have loops (in all three non-interface files): `constructor()`, `reinvest()`, `_removeLiquidity()`, `_addLiquidity(), and`rebalance()` .

Recommendation: Ensure that the loop can always be used to iterate over the entire set of values and calls, either by allowing sub-ranges to be executed, or by conducting a gas usage analysis.

QSP-13 Clone-and-Own

Severity: *Informational*

Status: Unresolved

File(s) affected: `interfaces.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. 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.
All interfaces defined in `interface.sol` are cloned from existing open-source libraries such as OpenZeppelin and Uniswap v3.

Recommendation: Instead of cloning code, one should rather import it and lock its version.

QSP-14 Unclear Operation/Lack of Documentation

Severity: *Undetermined*

Status: Unresolved

File(s) affected: `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol`, `CellarPoolShareLimitUSDCETH.sol`

Description: `invest()` and weight operations `_getWeightInfo()` & `_modifyWeightInfo()` are complicated and there's no supporting documentation. Additionally there is duplication of these functions across the Solidity files and there should only be one.

Recommendation: Please provide documentation to describe the intended outcome of these functions.

QSP-15 Clarify Flash Loan Protection

Severity: *Undetermined*

Status: Unresolved

File(s) affected: `interfaces.sol`

Description: Contract `BlockLock` is used for protection against flash loans.

Recommendation: Please provide documentation on the intended behaviour of this protection.

QSP-16 `addLiquidityForUniV3` Accepts Too Much

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `CellarPoolShare.sol`

Description: The function `addLiquidityForUniV3` appears to refund or return some values - this is not ideal. Most simply, it may waste gas; more seriously, it may introduce accounting complications.
This issue may be present in the other two similar files.

Recommendation: Instead of refunding tokens, take only what is necessary.

QSP-17 No Deadlines

Severity: *Undetermined*

Status: Fixed

File(s) affected: `CellarPoolShare.sol`

Description: There are lots of lines with `deadline: type(uint256).max`, effectively removing deadlines.

Recommendation: If deadlines are not required or enforced, this should be documented. If deadlines which are finite are expected, these should be implemented in the code.

Code Documentation

1. There is a lack of docstrings: functions should be well documented with docstrings defining intended operation.

Adherence to Best Practices

1. `interfaces.sol` should be renamed to `Interfaces.sol` for consistency.
2. `Interfaces.sol` also contains libraries, and as such, the name is not entirely accurate.
3. Each file should contain one contract, library, or interface, for ease of readability and maintainability.
4. Unclear error codes: many functions have `require()` clauses to evaluate conditions prior to execution. The error returned to the user is in the format `RX`, where `X` is an integer. It would be better for users to have a meaningful error message returned instead of a number.
5. **[fixed]** Given the similarity of `CellarPoolShare.sol`, `CellarPoolShareLimitETHUSDT.sol` and `CellarPoolShareLimitUSDCETH.sol`, it's likely that the code could be refactored so that it is easier to read and maintain, taking the common parts and placing them in a separate file.
6. `CellarPoolShare.sol`: the numeric value on line 32 could be written as `1 year` to improve readability.
7. **[fixed]** `CellarPoolShare.sol`: the overridden ERC20 functions that call `internal` functions should `require` that those internal functions return `true`.
8. `CellarPoolShare.sol`: `transferFrom` does not follow the `check-effects-interaction` design pattern.
9. `CellarPoolShare.sol`: The `if` statement starting on line 307 is followed by another complementary one on line 342, but these do not cover all cases (in particular, what happens if there is equality?). At the very least, this should be documented.

Test Results

Test Suite Results

```
================================ test session starts ================================
platform linux -- Python 3.6.9, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: /app
plugins: eth-brownie-1.16.4, web3-5.23.1, xdist-1.34.0, forked-1.3.0, hypothesis-6.21.6
collected 16 items
Attached to local RPC client listening at '127.0.0.1:8545'...

tests/test_00_add_liquidity.py ..                                          [ 12%]
tests/test_01_transfer.py ...                                              [ 31%]
tests/test_02_remove_liquidity.py ...                                      [ 50%]
tests/test_03_reinvest.py ...                                              [ 68%]
tests/test_04_rebalance.py ...                                             [ 87%]
tests/test_05_weight.py .                                                  [ 93%]
tests/test_06_add_liquidity_limit.py .                                     [100%]

================================= warnings summary ==================================
../home/ethsec/.local/pipx/venvs/eth-brownie/lib/python3.6/site-packages/brownie/network/main.py:46
  /home/ethsec/.local/pipx/venvs/eth-brownie/lib/python3.6/site-packages/brownie/network/main.py:46: BrownieEnvironmentWarning: Development network has a block height of 13416770
    BrownieEnvironmentWarning,

-- Docs: https://docs.pytest.org/en/stable/warnings.html
====================== 16 passed, 1 warning in 359.90s (0:05:59) ======================
```

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

```
90ee9dbbcbf22e10fc56d247c0b6f60f9a01d67ba1848bf60d47f322e16e2db6 ./contracts/CellarPoolShare.sol
118144af98f39c72e4a733f1a54d9865f877163ce65bde5b6b4cb9201f698a99 ./contracts/CellarPoolShareLimitETHUSDT.sol
72dbd4cdab5a139a1ab2e336df9ea73d355b2504f6ffeb06bc40f5f66033a6b9 ./contracts/CellarPoolShareLimitUSDCETH.sol
ac5befac7cc3193b2c88ce8a84883d07aeef100b7bffa8c67cb5c3c247b1a998 ./contracts/interfaces.sol
```

Tests

```
4757e9792d7513d3782b628c7bf20a7390ab7c98dce364440ea31312ceb61efe ./tests/conftest.py
a875e08607ca57657f8970075328b69b45215397a5c54bbb161f2948141d34ec ./tests/test_00_add_liquidity.py
5e0c98f49403d09bd834676bcd953c1d9e6fe7bf473973f23a04b2c7b849cc51 ./tests/test_01_transfer.py
49e9c7bba5d2bb694b346dca4ec126de7f69cb686e8cf76440e37fe33fd21974 ./tests/test_02_remove_liquidity.py
ee0154e57d7eca984d0c97f6393b481dd29957bc460eb922e82648aaa9006af3 ./tests/test_03_reinvest.py
341dcc6e886559ae9f87676e12f67f9953f34c69209939246b4cc33860e32061 ./tests/test_04_rebalance.py
8a88ccc36296056e3ee5f575d3e7660d13f4fb70640b7b19e84b9a7b4dc7a813 ./tests/test_05_weight.py
a297fba2b1bbd74109eae54078c5bf319b6e2db453e706ab2c4431f666ccb53e ./tests/test_06_add_liquidity_limit.py
```

Changelog

- 2021-10-21 - Initial report [`b887510`]
- 2021-11-04 - Final report [`09d2e7b`]

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.