Quantstamp

Venus WstETH Oracle

Executive Summary

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

Type	Oracle
Timeline	2024-02-07 through 2024-02-13
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Protocol Documentation
Source Code	• VenusProtocol/oracle #f638cef
Auditors	• Shih-Hung Wang Auditing Engineer • Hytham Farah Auditing Engineer • Faycal Lalidji Senior Auditing Engineer • Mostafa Yassin Auditing Engineer

Documentation quality	High
Test quality	High
Total Findings	6 Acknowledged: 5 Mitigated: 1
High severity findings	0
Medium severity findings	0
Low severity findings	3 Acknowledged: 2 Mitigated: 1
Undetermined severity findings	0
Informational findings	3 Acknowledged: 3

Summary of Findings

Venus Protocol is a DeFi lending protocol deployed on multiple chains, including Binance Smart Chain and Ethereum. This audit focused on the `WstETHOracle` contract, which will be configured in the `ResilientOracle` as the MAIN oracle for the `wstETH` token on Ethereum.

This audit report identified two low-severity issues related to the oracle design. Specifically, the oracle assumes a fixed 1:1 ratio between the `stETH` token and `ETH` (VWST-1). Also, there may be a risk of using stale price data returned from the oracle (VWST-2).

The code is well-documented and has 100% test coverage. The audit team has strictly covered the files in the Scope section, and any other files were out of the scope of this audit. It is strongly recommended the Venus team to address all issues outlined in this report.

Update: All issues have been either mitigated or acknowledged by the Venus team. For VWST-1, please see the issue details below for the Venus team's response on how this issue will be mitigated.

ID	DESCRIPTION	SEVERITY	STATUS
VWST-1	Oracle Assumes a Fixed 1:1 Ratio Between stETH and ETH	Low	Mitigated
VWST-2	Oracle with Multiple Fallbacks Does Not Fully Mitigate Stale Price Risk	Low	Acknowledged
VWST-3	Missing Input Validation	Low	Acknowledged
VWST-4	Enhancing Reliability in stETH to wstETH Conversion	Informational	Acknowledged
VWST-5	Redundant Parameter	Informational	Acknowledged
VWST-6	Use of Solidity Version with Known Compiler Bugs	Informational	Acknowledged

Assessment Breakdown

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

> **Disclaimer**
> Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

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

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. 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:
    1. 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.
    2. 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, clarity, 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.

Scope

Files Included

contracts/oracles/WstETHOracle.sol

Findings

VWST-1
Oracle Assumes a Fixed 1:1 Ratio Between stETH and ETH

Low | Mitigated

> **Update**
> The Venus team added a constructor parameter, `ASSUME_STETH_ETH_EQUIVALENCE`, to the `WstETHOracle` contract, indicating whether the oracle should assume a 1:1 ratio between `stETH` and `ETH`. If not, the market price of `stETH` will be obtained through the `ResilientOracle`.
>
> This change allows the Venus team to deploy two `wstETH` oracles and configure which oracle (or both) to use in the `ResilientOracle` based on their assessment of the `stETH` market conditions. Note that the correctness of the configuration in the `ResilientOracle` and how the market price of `stETH` is obtained was out of scope and not examined through this audit.
>
> We also note that this mitigation increases centralization risk as it would be up to the team to identify what constitutes a "long-term depeg" and act accordingly.

> **Update**

Marked as "Mitigated" by the client. Addressed in: `dd0fba403de31545b979dd1513eaf53e085ffd7d`. The client provided the following explanation:

> We agree that in some cases assuming 1:1 ratio between `stETH/ETH` might be wrong. There is an option to add an `stETH/USD` price feed as pivot oracle in our `ResilientOracle` in order to have an on-chain kill switch in case of depeg. However we believe that if there is depeg of this ratio, it will be a short term depeg. In case of a short term depeg and configuring a pivot price feed (`stETH/USD`) in our `ResilientOracle` users will not be able to use the protocol, since the resilient oracle will return invalid price, meaning borrowing, repaying and liquidations will be not possible. Another option is to use only the `stETH/USD` price feed and not assume 1:1 ratio at all. This solution is again not really optimal. Usually `ETH` and `stETH` are on peg, meaning that users will borrow near the maximum allowed amount they can. Meaning that in case of a short term depeg a lot of false liquidations will be forced which will end up in users loosing their positions. Short term depegs are expected happen from time to time in case of big `stETH` redemptions or network congestion, but the peg always tends to restore. In a case of a long term depeg (or a black swan event) we propose the following mitigation:
>
> 1. We will have 2 deployed `wstETH` oracles on-chain:
>   - One oracle will return price based on 1:1 ratio assumption between `stETH/ETH`
>   - One oracle will return price based on `stETH/USD` market price feed
> 2. By default in the `ResilientOracle` we will have only configured the oracle assuming 1:1 ratio between `stETH/ETH` , as main oracle
> 3. The other oracle (getting price from `stETH/USD` price feed and not assuming 1:1 ratio) will not be configured in our `ResilientOracle`
> 4. We will have an off chain monitoring system in place, monitoring the prices returned from both oracles. In case of a big deviation, our team will decide if to replace the oracle assuming 1:1 ratio with the oracle not assuming it, or to add the latter as a pivot oracle for the time being. In order to maintain the same code base, we have implemented a logic with a boolean flag, based on it the oracle will either assume 1:1 ratio, or will check `stETH/USD` price. We have added also this mitigation plan in our documentation

File(s) affected: `WstETHOracle.sol`

Description: The `WstETHOracle` contract returns the price of the `wstETH` token in USD, which is calculated as follows:

$$\mathrm{P}_{\mathrm{WSTETH/USD}} = \mathrm{P}_{\mathrm{WSTETH/STETH}} \times \mathrm{P}_{\mathrm{WETH/USD}}$$

where $\mathrm{P}_{\mathrm{WSTETH/STETH}}$ represents the amount of `stETH` equivalent to 1 `wstETH` token, and $\mathrm{P}_{\mathrm{WETH/USD}}$ represents the price of `WETH` in USD. The former is fetched from the on-chain `stETH` token contract and reflects the actual exchange rate between `wstETH` and `stETH` in the Lido protocol when querying the price. The latter is fetched and returned from the `ResilientOracle` .

This pricing formula assumes a 1:1 ratio between `stETH` and `ETH` , i.e., `stETH` is pegged to `ETH` . Such a design raises the following security concerns:

1. The oracle is unable to reflect the market price of `stETH` . Suppose `stETH` depegs from `ETH` , which may be due to reasons such as a large amount of `ETH` being removed from the Curve `stETH/ETH` pool. The market price of `stETH` has dropped, while the protocol may overvalue `stETH` temporarily during the depeg event. Assuming the depeg does not recover in time (note that withdrawals from Lido are not confirmed instantly compared to deposits), positions with `wstETH` as collateral will have an advantage in terms of borrowing power during this period.
2. Suppose the depeg continues so that the spread between the exchange rate and the market price is larger than $1 - \mathrm{LT}$ (where LT represents the liquidation threshold). In that case, it will become profitable to deposit `stETH` as collateral to the protocol, borrow `ETH` , and sell it in the secondary market. The utilization rate of `ETH` may increase, leading to potential issues such as lacking liquidity for withdrawals.
3. In an extreme scenario where the depeg cannot be recovered, the protocol may accrue bad debts if underwater positions with `wstETH` as collateral cannot be liquidated in time.

Recommendation: Consider implementing off-chain monitoring of the `stETH/ETH` exchange rate and react promptly if `stETH` depegs to an abnormal threshold. Possible actions could be pausing the `wstETH` market or adjusting the market parameters (e.g., the liquidation threshold), depending on the protocol choice and acceptance of associated risks.

An alternative approach is to obtain the `stETH/ETH` price from Chainlink price feeds or on-chain oracles to ensure that the market price of `stETH` is within an acceptable range.

VWST-2
Oracle with Multiple Fallbacks Does Not Fully Mitigate Stale Price Risk

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client. The client provided the following explanation:
>
> > The accuracy of `WETH` feed depends on the configuration in `ResilientOracle` we can configure Main and Pivot price feed in order to detect false price and return invalid price. When it comes to stale price, all of our oracles check for price staleness and return invalid price if the feed hasn't been updated in a time more than the price feed's usual heartbeat.

File(s) affected: `WstETHOracle.sol`

Description: The `ResilientOracle` contract attempts to manage extreme scenarios where a substantial number of dependent oracles fail. Nonetheless, the oracle function used does not return the last update time, meaning that the risk of utilizing stale price data still remains with a low likelihood.

We also note, the oracle also assumes that the `ResilientOracle` will accurately return the price of a `WETH` in terms of USD. In case this assumption fails, the oracle will return inaccurate results.

Recommendation: It is crucial to acknowledge that using an oracle that implements multiple fallback mechanisms reduces the risk of stale data, the possibility of encountering such issues cannot be entirely eliminated and always exists with a low likelihood.

VWST-3 Missing Input Validation

• Low Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

> We do not want to rely on this check for the simple reason that it can easily be bypassed (by calling the function from contract constructor). Moreover, before any configuration/deployment happens on-chain we use fork tests to validate that the contract is configured properly and works as expected.

File(s) affected: `WstETHOracle.sol`

Description: While there are non-zero address checks, consider increasing the robustness of the input validation by using OpenZeppelin's `isContract()` function to ensure that all the addresses provided are smart contract addresses and not EOA.

Recommendation: Implement the above reccomendation or consider hardcoding the addresses directly into the contract itself to eliminate the risk of an incorrect address being used in deployment.

VWST-4
Enhancing Reliability in stETH to wstETH Conversion

• Informational Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

> In `wstETH` contract we have the same function:
>
> ```
> /**
> * @notice Get amount of stETH for a one wstETH
> * @return Amount of stETH for 1 wstETH
> */
> function stEthPerToken() external view returns (uint256) {
>     return stETH.getPooledEthByShares(1 ether);
> }
> ```
>
> In our code what we are doing is we directly call `stETH.getPooledEthByShares(1 ether);` which is practically the same and moreover we are saving gas. In our opinion there are not risks because:
> 1. `wstETH` is non-upgradeable and this function will be always like that
> 2. in case `stETH` implementation of this function changes, so the result in both `wstETH.stEthPerToken()` and `stETH.getPooledEthByShares(1 ether);` will be the same.

File(s) affected: `WstETHOracle.sol`

Description: `WstETH` contract contains a specific function that allows getting the amount of `stETH` from `wstETH`:

```
contract WstETH is ERC20Permit {
    ...
    function getStETHByWstETH(uint256 _wstETHAmount) external view returns (uint256) {
        return stETH.getPooledEthByShares(_wstETHAmount);
    }
    ...
}
```

Using the `stETH` contract directly to compute the `stETH` value from a fixed share value by calling `STETH.getPooledEthByShares(1`

ether) in the WstETHOracle.getPrice() function is correct. However, any future upgrade might create unknown results so we strongly recommend using WstETH.getStETHByWstETH() instead of the stETH contract.

Recommendation: Consider implementing the recommendation above.

VWST-5 Redundant Parameter

• Informational Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

> The main reason to leave the parameter as it is, is to be compliant with OracleInterface. Otherwise if we remove the parameter, we need to adjust the logic in ResilientOracle which is not really preferred by us.

File(s) affected: WstETHOracle.sol

Description: The getPrice() function accepts a parameter but reverts unless it is WSTETH_ADDRESS. In this case the parameter is not necessary, however, may still be kept in order to conform to the OracleInterface.

Recommendation: Consider returning the wstETH price no matter what input is provided. This however relies on a correct token configuration in the ResilientOracle.

VWST-6
Use of Solidity Version with Known Compiler Bugs

• Informational Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

> Since the vulnerabilities are not applicable for our code (most of them are related to using Yul in your smart contracts), we don’t think this is crucial for fixing.

File(s) affected: WstETHOracle.sol

Description: The in-scope contract is compiled using Solidity version 0.8.13. According to the Solidity official's compiler bug list, this version contains several known compiler bugs, specifically 1 medium/high-severity, 2 medium-severity, and 6 low-severity bugs. According to our examination, these bugs seem to not affect the code in scope.

Recommendation: As the known compiler bugs do not affect the code, no action is required. As security best practice, consider updating the Solidity version to the latest or a more recent version to avoid the code from potentially being affected by the compiler bugs. If new compiler bugs are discovered in the future, ensure that they do not affect the code.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – 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.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **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).

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

- `c5e...f7a ./contracts/oracles/WstETHOracle.sol`

Tests

- `5f7...8ff ./test/WstETHOracle.ts`

Toolset

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

Setup

Tool Setup:

- Slither v0.10.0

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

Non-false positive findings have been included in the report.

Test Suite Results

The test suites were run by calling `npx hardhat test`. Only the relevant tests have been included in the following output. All tests passed.

```
WstETHOracle unit tests
  deployment
    ✔ revert if wstETH address is 0
    ✔ revert if WETH address is 0
    ✔ revert if stETH address is 0
    ✔ revert if ResilientOracle address is 0
    ✔ should deploy contract
getPrice
    ✔ revert if wstETH address is wrong
    ✔ should get correct price
```

Code Coverage

Coverage was gathered by running `npx hardhat coverage`. Only the files in scope are included in the following output.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
WstETHOracle.sol	100	100	100	100	

Changelog

- 2024-02-13 - Initial report
- 2024-02-20 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Quantstamp