Quantstamp

Vusd-Stablecoin

Executive Summary

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

Type	Stablecoin
Timeline	2025-03-24 through 2025-03-31
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• https://github.com/hemilabs/vusd-stablecoin #7b66754
Auditors	• Paul Clemson Auditing Engineer • Leonardo Passos Senior Research Engineer • Tim Sigl Auditing Engineer

Documentation quality	Medium
Test quality	Medium
Total Findings	10 Fixed: 3 Acknowledged: 7
High severity findings	1 Fixed: 1
Medium severity findings	1 Fixed: 1
Low severity findings	4 Fixed: 1 Acknowledged: 3
Undetermined severity findings	0
Informational findings	4 Acknowledged: 4

Summary of Findings

VUSD is a stablecoin protocol that allows users to mint VUSD tokens by depositing a whitelisted stablecoin into the project's treasury. The amount of VUSD tokens minted is based on the current oracle price of the deposited stablecoin. Users can later redeem their VUSD and receive one of underlying stablecoins in return. The assets held by the treasury are deposited into the Compound protocol allowing them to earn a yield in COMP tokes. These COMP tokens are later converted back into one of the whitelisted stablecoins, with the intention of slowly growing the collateralization ratio of VUSD token.

Fix Review Update: The team fixed or acknowledged all issues and the suggestions provided by the audit team. As some of the contracts in scope are already live, the client intends to apply fixes for some of the non-urgent issues found in a future update.

ID	DESCRIPTION	SEVERITY	STATUS
VUS-1	Stablecoin Arbitrage Leads to Potential for Vusd to Become Undercollateralized	High	Fixed
VUS-2	Stale Oracle Price Data Not Checked Before Token Operations May Drain Assets From Protocol	Medium	Fixed
VUS-3	Lack of Error Handling for Oracle Calls May Cause Temporary Service Disruption	Low	Acknowledged
VUS-4	Lack of Slippage Protection on Mint and Redeem Functions	Low	Fixed
VUS-5	Proof of Reserves Invariant Can Be Broken by Treausry Updates	Low	Acknowledged
VUS-6	Unfair Token Redemptions when an Underlying Stablecoin Depegs	Low	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
VUS-7	`oracles` Mapping Not Cleared in `Removewhitelistedtoken()` Function	Informational	Acknowledged
VUS-8	No Validation that `_newMintLimit` Is at Least Total Supply	Informational	Acknowledged
VUS-9	Hardcoded Decimal Conversion May Not Support All Erc-20 Tokens	Informational	Acknowledged
VUS-10	Infinite Token Approvals Create Additional Security Risk	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

Repo: https://github.com/hemilabs/vusd-stablecoin(7b66754aa4e7e79aefd47361902759464bf1da54) Files: /contracts/*.sol
Repo: `hemilabs/vusd-stablecoin/7b66754aa4e7e79aefd47361902759464bf1da54`

Files Excluded

Repo: https://github.com/hemilabs/vusd-stablecoin(7b66754aa4e7e79aefd47361902759464bf1da54) Files: /contracts/test/*

Operational Considerations

- Users must be aware that the `governor` role has a lot of authority within the protocol, including the ability to mint unbacked VUSD tokens as well as being able to arbitrarily transfer tokens out of the treasury contract.
- The protocol inherits all of the security considerations of the Compound protocol.
- The protocol inherits all of the security considerations of the stablecoins it chooses to whitelist.
- The protocol will be significantly impacted if any of the whitelisted stablecoins in its treasury were to lose their peg.

Key Actors And Their Capabilities

There are three main actors within the protocol:

Governor: The governor is the main admin role within the protocol. It has a large number of permissions, including but not limited to:

- Minting VUSD without backing.
- Withdrawing tokens from the treasury.
- Migrating the treasury to a new contract.
- Adding/removing tokens from the whitelist.
- Converting earned COMP into one of the whitelisted stable coins.
- Changing fee amounts, and the accepted price tolerance.
- Adding/removing other roles.

Keeper: The keeper role has the ability to convert earned COMP tokens into one of the whitelisted stablecoins.

Redeemer: The redeemer role has the power to withdraw tokens out of the treasury. It is expected that this role will only be held by the `Redeemer` smart contract.

Findings

VUS-1

Stablecoin Arbitrage Leads to Potential for Vusd to Become Undercollateralized

High Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `655b2efc2704480b32a13a2a52886a71520eb8ea` .

File(s) affected: `Redeemer.sol, Minter.sol`

Description: The core invariant of the protocol is that the VUSD token should always be backed at least 1:1 by the stablecoins held in its treasury. However because VUSD tokens are minted to users based on the oracle price of the stablecoin being deposited, there is an opportunity for arbitrageurs to mint more VUSD with stablecoins that have temporarily deviated slightly above one dollar. They can then immediately redeem this VUSD for another of the whitelisted stablecoins that price has not deviated from one dollar.

This scenario allows the arbitrageur to make a small profit, but more importantly means that when the deviated price of the first stable coin returns to one dollar the protocol will now be slightly undercollateralized. Over time this could be repeated whenever such a price deviation occurs to slowly force the protocol to become more and more undercollateralized.

Exploit Scenario:
Consider the following scenario:

1. There is $250k VUSD minted back by 150k USDT/100k USDC in the treasury
2. The price of USDC deviates to $1.007 (within the default accepted `priceTolerance`)
3. A user deposits $100k USDC getting back 100,700 VUSD token
4. The user then immediately redeems these for USDT, receiving ~ 100,400 USDT after paying the `redeemFee`
5. The price of USDC returns back to $1
6. The protocol treasury only has 200k USDC and 49,600 USDT despite there still being 250k VUSD in circulation

Adding the following test to the codebase's foundry test suite highlights this:

```
    function test_TreasuryUndercollat() public {
        // 100k minted in USDC, 150k minted in USDT
        address whale = makeAddr("whale");
        deal(USDC, whale, 100_000e6);
        deal(USDT, whale, 150_000e6);

        vm.startPrank(whale);

        address treasury = minter.treasury();
```

```
        IERC20(USDC).approve(address(minter), 100_000e6);
        IUSDT(USDT).approve(address(minter), 150_000e6);

        minter.mint(USDC, 100_000e6);
        minter.mint(USDT, 150_000e6);
        uint256 treasuryCUSDCBalSt = IcToken(cUSDC).balanceOfUnderlying(treasury) * 10**12;
        uint256 treasuryCUSDTBalSt = IcToken(cUSDT).balanceOfUnderlying(treasury) * 10**12;

        console2.log("Treasury Bal Stt", treasuryCUSDCBalSt + treasuryCUSDTBalSt);
        vm.stopPrank();

        // USDC price adjusted $1.007
        usdcOracle.setPrice(1e8 + 7e5);

        // Alice mints with USDC
        deal(USDC, alice, 100_000e6);
        vm.startPrank(alice);
        IERC20(USDC).approve(address(minter), 100_000e6);
        minter.mint(USDC, 100_000e6);

        // Alice redeems USDT
        uint256 aliceBalance = IERC20(vusd).balanceOf(alice);
        IERC20(vusd).approve(address(redeemer), aliceBalance);
        redeemer.redeem(USDT, aliceBalance);

        // USDC price moves back to $1.00
        usdcOracle.setPrice(1e8);

        uint256 vusdSupply = IERC20(vusd).totalSupply();

        // Get CTokenAmounts
        uint256 treasuryCUSDCBal = IcToken(cUSDC).balanceOfUnderlying(treasury) * 10**12;
        uint256 treasuryCUSDTBal = IcToken(cUSDT).balanceOfUnderlying(treasury) * 10**12;
        console2.log("Treasury Bal End", treasuryCUSDCBal + treasuryCUSDTBal);

        assert(treasuryCUSDCBal + treasuryCUSDTBal < vusdSupply);
    }
```

Recommendation: Consider lowering the default `priceTolerance` and setting a non-zero default `mintingFee` to ensuring that `mintingFee + redeemFee >= priceTolerance`. This will make it significantly harder for any potential arbitrage to undercollateralize the treasury.

VUS-2

Stale Oracle Price Data Not Checked Before Token Operations May Drain Assets From Protocol

Medium | Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `5f55a01c23fde5764bbef05ca11593c430df854c` .

Further Update

Additional fixes applied in `b3dc7e4233ce5818d9f01bc73ba72ed047b04659` and `e9a6380cc1888e6ae96b44f11e838b4aa52fa8b8` .

File(s) affected: `Redeemer.sol, Minter.sol`

Description: The Redeemer and Minter contracts use Chainlink oracles to fetch price data for stablecoins to calculate redemption and minting amounts. However, these contracts only check if the price falls within a tolerance range but don not verify the freshness of the price data. In the `_calculateRedeemable()` function of the Redeemer contract and the `_calculateMintage()` function of the Minter contract, the code calls `latestRoundData()` but only uses the price value without checking the timestamp of when this price was last updated.

Exploit Scenario:

1. The Chainlink oracle for a stablecoin stops updating for an extended period due to technical issues or other reasons.
2. During this time, the actual market price of the stablecoin deviates significantly from its pegged value (e.g., during a depeg event).
3. An attacker notices this situation and uses the stale price data to their advantage.
4. The attacker can mint VUSD using the depegged stablecoin at its previous pegged price.

5. The attacker then redeems this VUSD for a different, non-depegged stablecoin.
6. This arbitrage opportunity allows the attacker to drain assets from the protocol and make significant profits.

Recommendation: Add staleness checks when getting price data from Chainlink oracles by:

Retrieve and check the `updatedAt` timestamp from the `latestRoundData()` function. Define a maximum acceptable staleness threshold based on the heartbeat (the update frequency) of the oracle. Require that the time difference between the current timestamp and `updatedAt` is less than this threshold. Implement this check in both the `_calculateRedeemable()` function in the Redeemer contract and the `_calculateMintage()` function in the Minter contract.

VUS-3
Lack of Error Handling for Oracle Calls May Cause Temporary Service Disruption

Low | Acknowledged

Update

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

```
Will fix it in future release when we have fallback oracle ready.
```

File(s) affected: `Redeemer.sol, Minter.sol`

Description: The Redeemer and Minter contracts interact with Chainlink oracles to fetch price data for stablecoins, but they do not implement proper error handling mechanisms for these external calls. In the `_calculateRedeemable()` function of the Redeemer contract and the `_calculateMintage()` function of the Minter contract, the code directly calls `latestRoundData()` without any try-catch block or fallback mechanism. If Chainlink bans the contract address from accessing price feeds or if the oracle service experiences downtime, these functions will fail, leading to a temporary disruption of minting and redemption services.

Exploit Scenario:

1. Chainlink decides to ban the contract address from accessing their price feeds due to policy changes or abuse detection.
2. All redemption and minting functions in the protocol immediately stop working since they depend on oracle data.
3. Users are temporarily unable to redeem their VUSD tokens until the governor intervenes.
4. The governor must identify the issue, remove the problematic token from the whitelist, and then re-add it with a new oracle address.
5. This process may take time, during which the protocol's core functionality is impaired, potentially causing user frustration and loss of confidence.

Recommendation: Consider taking one of the following actions to fix this issue:

1. Implement a try-catch block around oracle calls to gracefully handle any failures.
2. Create a fallback mechanism with alternative oracle providers that can be used temporarily in case of main oracle failure.
3. Add monitoring systems to alert the team when oracle calls start failing so they can take proactive measures.
4. Document the recovery process for oracle failures so the governor can quickly restore functionality by removing and re-adding the affected token with an updated oracle address.

VUS-4 Lack of Slippage Protection on Mint and Redeem Functions

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `5f55a01c23fde5764bbef05ca11593c430df854c` .

Description: When minting and redeeming tokens within the protocol there is an expectation that the user will know how many tokens they are going to receive. However in certain scenarios the user may receive less tokens than they expected when they submitted the transaction. This could happen in a number of scenarios including:

- The oracle price is updated, changing the value of the underlying stable coins.
- The governor changes the `mintingFee` while a user's transaction is pending.

In the current implementation the user has no control over this and their transaction would succeed even if the number of tokens they receive is unfavorable.

Recommendation: Consider allowing the user to specify a minimum amount of tokens they wish to receive when minting or redeeming tokens and revert if the actual amount is less than this.

VUS-5
Proof of Reserves Invariant Can Be Broken by Treausry Updates

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Planned to fix it in future release of treasury contract
> ```

File(s) affected: `VUSD.sol`

Description: Description: From the VUSD documentation, it says:

```
VUSD is open source, designed for simplicity and security, peg stability, self-sustaining, backed by
over-collateralized, yield-generating cryptocurrencies. VUSD is fully auditable on-chain, with real-
time proof of reserves available at all times.
```

Having a proof of reserve means that all VUSD should be backed by at least the corresponding amount in USD, as given the collateral assets in the treasury.

This invariant, however, is not enforced in `VUSD.updateTreasury()`. Hence, a malicious governor may set a new treasury with no matching reserve to back the value of all the VUSD in circulation.

Recommendation: In the `updateTreasury()` function, check that the new treasury holds enough collateral to back the value of all the VUSD current supply.

VUS-6
Unfair Token Redemptions when an Underlying Stablecoin Depegs

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Planned to fix it in future release of treasury contract
> ```

Description: The protocol currently has no clear contingency plan on how it would handle the event where one of the stablecoins backing VUSD depegs. Currently, `redeem()` function allows the user to specify which token they wish to withdraw for their USDC, therefore in a depeg scenario holders of VUSD would race to withdraw their tokens for one of the whitelisted stablecoins that has not depegged, leaving the last users holding a token that is now only backed by the depegged stablecoin.

Recommendation: Consider having the `redeem()` function return to users a percentage share of the underlying treasury. This would mean that in the event of one of the whitelisted stablecoins depegging all users would be able to redeem their fair share of the treasury, rather than only rewarding the users who are fastest to withdraw.

VUS-7
`oracles` Mapping Not Cleared in `Removewhitelistedtoken()` Function

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Planned to fix it in future release of treasury contract
> ```

File(s) affected: `Treasury.sol`

Description: In the Treasury contract, when a token is removed from the whitelist via the `removeWhitelistedToken()` function a number of key state changes are made to clear any state related to the token to be removed. However, in the current implementation of the function, the `cTokens[_token]` mapping is mistakenly deleted twice, while the `oracles[_token]` mapping is not cleared. This means some stale state is left behind that refers to the now removed token.

Recommendation: Consider the following change to ensure all state related to the now removed token is correctly cleaned up:

```
        delete cTokens[_token];
+       delete oracles[_token];
-       delete cTokens[_token];
```

VUS-8
No Validation that `_newMintLimit` Is at Least Total Supply

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> `Governor may want to set lower limit or 0 limit to disable minting`

File(s) affected: `Minter.sol`

Description: The `updateMaxMintAmount()` function allows the governor to set a new maximum amount of VUSD tokens that can be minted. However there is no validation that the new mint limit is not less than the amount of VUSD already in circulation. If this were to happen it would immediately block the ability to mint tokens in the protocol and leave a potentially confusing mismatch between the mint limit and the currently circulating total supply of VUSD.

Recommendation: Consider adding a validation to ensure that the new mint limit is not less than the current total supply of VUSD:

```
require(_newMintLimit >= vusd.totalSupply(), "limit-too-low");
```

VUS-9
Hardcoded Decimal Conversion May Not Support All Erc-20 Tokens

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> `We dont care about ERC20 token that has decimal greater than 18.`

File(s) affected: `Redeemer.sol, Minter.sol`

Description: In the Redeemer contract's `_calculateRedeemable()` function, the code converts the redeemable amount from 18 decimals (VUSD's precision) to the destination token's decimal precision. This conversion uses a hardcoded formula that assumes tokens will have at most 18 decimals:

```
return _redeemable / 10**(18 - IERC20Metadata(_token).decimals());
```

While this approach works for tokens with 18 or fewer decimals, it would fail if a token with more than 18 decimals were added to the protocol in the future.

Exploit Scenario:

1. The protocol adds support for a new token that has more than 18 decimals of precision (though currently rare in practice).
2. When users try to redeem VUSD for this token, the formula `10**(18 - token.decimals())` would result in a negative exponent.
3. This would cause the redemption calculation to revert due to the invalid arithmetic operation.
4. Users attempting to redeem VUSD for this token would be unable to complete their transactions until the formula is updated.

Recommendation: Consider applying one of the following recommendations:

1. Add a validation check when adding new tokens to ensure they have at most 18 decimals.
2. Modify the conversion formula to gracefully handle tokens with any decimal precision.
3. Consider avoiding magic numbers in the code by defining constants for the decimal precision values.

VUS-10
Infinite Token Approvals Create Additional Security Risk

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Minter and Redeemer doesn't hold any erc20 token. It is immediately deposited to protocol when user interact for mint.  Router has approval of COMP token only not stable coins.
> ```

Description: The Minter contract uses infinite token approvals (`type(uint256).max`) when adding a new token to the whitelist. This occurs in the `_addToken()` function with the line `IERC20(_token).safeApprove(_cToken, type(uint256).max)` . Similarly, in the Treasury contract, infinite approvals are granted in multiple places: when adding tokens via `_addToken()` and when updating the swap manager via `_approveRouters()` . While this approach eliminates the need for subsequent approval transactions, it exposes the protocol to greater risk if any of the approved contracts (cToken contracts or routers) were to be compromised.

Exploit Scenario:

1. A vulnerability is discovered in one of the cToken contracts or swap routers that have been granted infinite approval.
2. An attacker exploits this vulnerability to drain all tokens from the Minter or Treasury contracts.
3. Since the attacker has access to the maximum possible allowance, they can transfer all available tokens in a single transaction.
4. The protocol loses all funds held in the affected contract, potentially causing significant financial damage.

Recommendation: Consider replacing infinite approvals with exact amounts needed for each transaction, particularly for critical operations.

Auditor Suggestions

S1 No Validation that `priceTolerance` Not Set to Zero

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> We would prefer to keep it as is.  If governor set 0 price tolerance, it will be done for a reason in exceptional scenario. Assuming governor wont set it 0 in normal scenario.
> ```

Description: If the `updatePriceTolerance()` function is called to set the maximum price tolerance to zero, it will block minting and redemption because the price returned from the oracle will very rarely be exactly one dollar.

Recommendation: Consider adding an input validation to ensure `_newPriceTolerance` cannot be zero.

S2 Treasury Contract Missing Events Around Key State Updates

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Currently no plan to fix informational item of Treasury.sol. This will be taken care future update of Treasury.sol
> ```

Description: Best practices dictate that events should be emitted whenever key changes are made to a contract's state. This ensures that actors off-chain can easily follow these changes as they happen. However, in the Treasury contract many of the key actions occur without events being emitted.

Recommendation: Consider emitting events when key state updates are made.

S3 Inconsistent Type Expected for `vusd` when Migrating to a New Treasury Contract

Acknowledged

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

> Currently no plan to fix informational item of Treasury.sol. This will be taken care future update of Treasury.sol

File(s) affected: `Treasury.sol`

Description: When migrating funds from the current treasury contract to a new contract there is the following check in the `migrate()` function:

```
require(address(vusd) == ITreasury(_newTreasury).vusd(), "vusd-mismatch");
```

This converts the current treasury contract's `vusd` variable from type `IVUSD` to type `address`, while expecting that the type of `vusd` in the `_newTreasury` is of type `address`. This change in the expected interface of the treasury contract could cause potential issues with other systems that attempt to interact with the system.

Recommendation: Consider ensuring the type expected for `vusd` is consistent across implementations of the treasury contract.

S4 Avoid Use of Magic Numbers

Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `41e2527f1a15cd7912768c31f6526ab0b58717ea`.
The client provided the following explanation:

> 41e2527f1a15cd7912768c31f6526ab0b58717ea

File(s) affected: `Redeemer.sol, Minter.sol`

Description: In the Redeemer contract's `_calculateRedeemable()` function, the conversion formula hardcodes VUSD's decimals as 18 when calculating token amounts:

```
return _redeemable / 10**(18 - IERC20Metadata(_token).decimals());
```

Even if there are no intentions of ever changing the number of decimals for the VUSD token, the readability of the code could be is improved by a more descriptive declaration.

Recommendation: Consider creating a state variable to store VUSD's number of decimals like so:

```
uint8 public immutable vusdDecimals;

constructor(address _vusd) {
    require(_vusd != address(0), "vusd-address-is-zero");
    vusd = IVUSD(_vusd);
    vusdDecimals = IVUSD(_vusd).decimals();
}
```

Then query this value when making the calculation:

```
return _redeemable / 10**(vusdDecimals - IERC20Metadata(_token).decimals());
```

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.

Files

- `3b4...6e1 ./contracts/Governed.sol`
- `db8...ae3 ./contracts/Redeemer.sol`
- `c6f...9ba ./contracts/VUSD.sol`
- `9b7...fab ./contracts/Minter.sol`
- `61c...0e0 ./contracts/Treasury.sol`
- `c82...dd3 ./contracts/interfaces/ITreasury.sol`
- `69c...aa0 ./contracts/interfaces/IVUSD.sol`
- `28b...672 ./contracts/interfaces/bloq/ISwapManager.sol`
- `7bd...ec8 ./contracts/interfaces/compound/ICompound.sol`
- `7f3...3ae ./contracts/interfaces/chainlink/IAggregatorV3.sol`
- `199...11a ./contracts/interfaces/curve/ICurveFactory.sol`
- `623...88d ./contracts/interfaces/curve/ICurveMetapool.sol`
- `f2b...74c ./contracts/interfaces/uniswap/IUniswap.sol`
- `915...5b6 ./contracts/test/IUniswapRouterTest.sol`

Toolset

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

Setup

Tool Setup:

- Slither v0.11.0

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

The Slither static analysis tool analysed 29 contracts with 115 detectors and found 401 results. Most of the findings were false positives, however any valid issues were included as part of the report.

Test Suite Results

The protocol's test suite is split over both hardhat and foundry tests. To run the tests the following is required:

1. Run the command `npm install`
2. Create a `.env` file and provide `NODE_URL` and `FORK_BLOCK_NUMBER` values 3.1. To run the foundry tests run the command `forge test` 3.2. To run the hardhat tests run the command `npm test`

It is worth noting that two of the hardhat tests remain stuck in the "pending" state, and this should be rectified by the team.

```
Foundry Test Output:
Ran 6 tests for test/foundry/Redeemer.t.sol:RedeemerTest
[PASS] testRedeemWithPriceToleranceExceeded() (gas: 56530)
[PASS] testRedeemWithUpdatedFee() (gas: 156443)
[PASS] testRedeemWithinPriceTolerance() (gas: 148337)
[PASS] testRedeemable() (gas: 67377)
[PASS] testUpdatePriceTolerance() (gas: 23387)
[PASS] testUpdateRedeemFee() (gas: 23391)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 3.13s (955.54ms CPU time)

Ran 11 tests for test/foundry/Minter.t.sol:MinterTest
[PASS] testAddAndRemoveWhitelistedToken() (gas: 99399)
[PASS] testCalculateMintage() (gas: 65617)
[PASS] testGovernorCanMint() (gas: 74821)
[PASS] testMintFailWithPriceDeviation() (gas: 316853)
[PASS] testMintMaxAvailableVUSD() (gas: 359845)
[PASS] testMintVUSD() (gas: 578975)
[PASS] testMintVUSDWithFee() (gas: 598040)
[PASS] testNonGovernorCannotMint() (gas: 17886)
[PASS] testUpdateMaxMintAmount() (gas: 23370)
[PASS] testUpdateMintingFee() (gas: 40565)
[PASS] testUpdatePriceDeviationLimit() (gas: 23520)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 14.53s (14.51s CPU time)

Ran 2 test suites in 14.56s (17.65s CPU time): 17 tests passed, 0 failed, 0 skipped (17 total tests)

╭---------------------------+------------------+------------------+----------------+----------------╮
| File                      | % Lines          | % Statements     | % Branches     | % Funcs        |
+=============================================================================================+
| contracts/Governed.sol    | 42.86% (6/14)    | 45.45% (5/11)    | 0.00% (0/6)    | 50.00% (2/4)   |
|---------------------------+------------------+------------------+----------------+----------------|
| contracts/Minter.sol      | 93.07% (94/101)  | 97.14% (102/105) | 57.89% (22/38) | 73.68% (14/19) |
|---------------------------+------------------+------------------+----------------+----------------|
| contracts/Redeemer.sol    | 92.31% (48/52)   | 98.18% (54/55)   | 62.50% (10/16) | 75.00% (9/12)  |
|---------------------------+------------------+------------------+----------------+----------------|
| contracts/Treasury.sol    | 0.00% (0/102)    | 0.00% (0/96)     | 0.00% (0/56)   | 0.00% (0/26)   |
|---------------------------+------------------+------------------+----------------+----------------|
| contracts/VUSD.sol        | 56.52% (13/23)   | 47.37% (9/19)    | 25.00% (4/16)  | 66.67% (4/6)   |
|---------------------------+------------------+------------------+----------------+----------------|
| test/foundry/Redeemer.t.sol | 66.67% (10/15) | 83.33% (5/6)     | 100.00% (0/0)  | 55.56% (5/9)   |
|---------------------------+------------------+------------------+----------------+----------------|
| Total                     | 55.70% (171/307) | 59.93% (175/292) | 27.27% (36/132) | 44.74% (34/76) |
╰---------------------------+------------------+------------------+----------------+----------------╯

  Live mint and redeem test
    ✓ Should verify mint and redeem

  VUSD Treasury
    Check Withdrawable
      ✓ Should return zero withdrawable when no balance
      ✓ Should return valid withdrawable
      ✓ Should return zero withdrawable for non supporting token
    Withdraw token
      ✓ Should revert if caller is neither governor nor redeemer
      ✓ Should revert if token is not supported
      ✓ Should allow withdraw by redeemer
      ✓ Should allow withdraw to another address by redeemer
      ✓ Should allow withdraw by governor
    WithdrawMulti by governor
      ✓ Should allow withdrawMulti by governor
      ✓ Should revert withdrawMulti if inputs are bad
    WithdrawAll by governor
      ✓ Should allow withdrawAll by governor
      ✓ Should revert withdrawAll if token is not supported
    Claim COMP
      - Should claim comp from all cToken markets
      - Should claim comp via keeper call
      ✓ Should revert if token is not supported
      ✓ Should revert if caller is not authorized
    Migrate to new treasury
```

```
    ✓ Should revert if new treasury address is zero
    ✓ Should revert if vusd doesn't match
    ✓ Should transfer all cTokens to new treasury
  Sweep token
    ✓ Should sweep token
    ✓ Should revert if trying to sweep cToken
  Update redeemer
    ✓ Should revert if caller is not governor
    ✓ Should revert if setting zero address as redeemer
    ✓ Should add new redeemer
    ✓ Should revert if setting same redeemer
  Update keeper
    ✓ Should revert if caller is not governor
    ✓ Should revert if setting zero address as keeper
    ✓ Should add new keeper
    ✓ Should remove a keeper
  Update swap manager
    ✓ Should revert if caller is not governor
    ✓ Should revert if setting zero address as swap manager
    ✓ Should add new swap manager
  Update token whitelist
    Add token in whitelist
      ✓ Should revert if caller is not governor
      ✓ Should revert if setting zero address for token
      ✓ Should revert if setting zero address for cToken
      ✓ Should add token address in whitelist
      ✓ Should revert if address already exist in list
    Remove token address from whitelist
      ✓ Should revert if caller is not governor
      ✓ Should remove token from whitelist
      ✓ Should revert if token not in list


VUSD
  Update miner address
    ✓ Should revert if caller is not governor
    ✓ Should revert if minter address is zero
    ✓ Should add new minter
    ✓ Should revert when same minter added again
  Update treasury address
    ✓ Should revert if caller is not governor
    ✓ Should revert if treasury address is zero
    ✓ Should add new treasury
    ✓ Should revert when same treasury added again
  Mint VUSD
    ✓ Should revert if caller is not minter
  Multi transfer
    ✓ Should revert if arity mismatch
    ✓ Should revert if no balance to transfer
    ✓ Should transfer VUSD to multiple recipients

·---------------------------------------|----------------------------|-------------|-----------------------·
|         Solc version: 0.8.3           ·  Optimizer enabled: false  ·  Runs: 200  ·  Block limit: 30000000 gas  │
|·····································|····························|·····|·····|·····|·····|·····|·····|·····|·····|
|  Methods                                                                                                      │
|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Contract  ·  Method  ·  Min  ·  Max  ·  Avg  ·  # calls  ·  eur (avg)  │
|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  ERC20  ·  approve  ·  46158  ·  55582  ·  50156  ·  12  ·  -  │
|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
|  Minter  ·  mint  ·  301547  ·  370852  ·  357759  ·  11  ·  -  │
|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|·····|
```

| Redeemer · redeem · – · – · 243392 · 1 ·

| Treasury · addKeeper · 77697 · 77709 · 77709 · 40 ·

| Treasury · addWhitelistedToken · – · – · 204187 · 1 ·

| Treasury · claimCompAndConvertTo · – · – · 508818 · 1 ·

| Treasury · migrate · – · – · 345203 · 1 ·

| Treasury · removeKeeper · – · – · 35083 · 1 ·

| Treasury · removeWhitelistedToken · – · – · 63873 · 1 ·

| Treasury · sweep · – · – · 44571 · 1 ·

| Treasury · updateRedeemer · – · – · 51896 · 5 ·

| Treasury · updateSwapManager · – · – · 83674 · 2 ·

| Treasury · withdraw · – · – · 188483 · 1 ·

| Treasury · withdraw · 198434 · 200627 · 199531 · 2 ·

| Treasury · withdrawAll · – · – · 172038 · 1 ·

| Treasury · withdrawMulti · – · – · 187991 · 1 ·

| VUSD · multiTransfer · – · – · 76517 · 1 ·

| VUSD · updateMinter · 48163 · 48175 · 48173 · 42 ·

| VUSD · updateTreasury · 31173 · 31185 · 31184 · 42 ·

| Deployments · · % of limit ·

```
|  Minter                                ·     3848720  ·    3848732  ·    3848731  ·       12.8 %  ·
-  |
·····································|·············|·············|·············|··············|
·············
|  Treasury                              ·     5125992  ·    5126004  ·    5126003  ·       17.1 %  ·
-  |
·····································|·············|·············|·············|··············|
·············
|  VUSD                                  ·           -  ·          -  ·    3457359  ·       11.5 %  ·
-  |
·-------------------------------------|-------------|-------------|-------------|--------------|-----
--------·

  51 passing (2m)
  2 pending
```

Changelog

- 2025-03-28 - Initial report
- 2025-05-06 - 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

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.