Algo Ssstablecoinsss Audit Report

Version 1.0

aayku

December 10, 2024

Algo Ssstablecoinsss Audit Report

aayku

December 10, 2024

Prepared by: aayku

Table of Contents

Protocol Summary

This protocol is meant to be a stablecoin where users can deposit WETH and WBTC in exchange for a token that will be pegged to the USD. The system is meant to be such that someone could fork this codebase, swap out WETH & WBTC for any basket of assets they like, and the code would work the same.

Disclaimer

aayku makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by aayku is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Vyper implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Repository: https://github.com/Cyfrin/2024-12-algo-ssstablecoinsss

Commit hash: 4cc3197b13f1db728fd6509cc1dcbfd7a2360179

Scope

```
src/
--- decentralized_stable_coin.vy
--- dsc_engine.vy
--- oracle_lib.vy
```

Executive Summary

Issues found

Severity	Number of Issues Found
High	2
Medium	0
Low	3
Info	2
Total	7

Findings

High

[H-1] Incorrect Health Factor Check Allows Full Collateral Withdrawal

Description: The protocol uses a health factor mechanism to ensure that collateral remains sufficient, preventing users from redeeming collateral when prices drop significantly. This mechanism is enforced through the `_revert_if_health_factor_is_broken` function, which reverts transactions if the user's health factor falls below a safe threshold.

However, the correct placement of this health factor check depends on the specific process being executed:

- Depositing Collateral When depositing collateral, the health factor should be checked after updating the user's collateral balance.
- Redeeming Collateral When redeeming collateral, the health factor check should be performed before any state changes, such as reducing collateral balances or burning tokens. This prevents the system from being manipulated by intermediate states where balances are reduced or set to zero. For this protocol, the redeeming process is implemented incorrectly because the health factor check happens after these critical state changes.

For example, in the `redeem_collateral_for_dsc` function, if a user withdraws all collateral and burns all tokens, the system will see `total_dsc_minted = 0` when performing the health factor check, returning the maximum possible value due to this logic:

```
  if total_dsc_minted == 0:
    return max_value(uint256)
```

This logic flaw allows bypassing the critical health factor check, enabling users to redeem full collateral even after a severe price drop, putting the protocol at risk of insolvency.

Note: The current implementation might assume `max_value(uint256)` as a valid return for users **without any DSC debt**. The critical risk occurs **only when users have non-zero DSC debt** but can still withdraw collateral due to bypassing the health factor check.

Impact: This logic flaw allows users with outstanding DSC debt to potentially redeem more collateral than intended after a severe price drop, risking protocol insolvency due to miscalculated health factors.

Proof of Concept:

This can be proved by running the following unit test:

```python
def test_redeem_collateral_for_dsc_works_if_eth_price_1usd(dsce, weth,
    dsc, some_user, eth_usd):
    # Ensure ETH price is in place
    eth_usd_updated_price = 2000 * 10**8 # 1 ETH = $2000
    eth_usd.updateAnswer(eth_usd_updated_price)

    with boa.env.prank(some_user):
        # Checks to ensure user has no value in the dsc_engine
        dsc_minted, collateral_in_value = dsce.get_account_information(
            some_user)
        assert dsc_minted == 0
        assert collateral_in_value == 0

        # Deposit collateral and mint stablecoins
        initial_user_weth_balance = weth.balanceOf(some_user)
        weth.approve(dsce, COLLATERAL_AMOUNT)
        dsc.approve(dsce, AMOUNT_TO_MINT)
        dsce.deposit_collateral_and_mint_dsc(weth, COLLATERAL_AMOUNT,
            AMOUNT_TO_MINT)
        assert weth.balanceOf(some_user) < initial_user_weth_balance

        # Adjust ETH price to $1
        eth_usd_updated_price = 1 * 10**8 # 1 ETH = $1
        eth_usd.updateAnswer(eth_usd_updated_price)

        # Redeem collateral for DSC - this should not be possible...
        dsce.redeem_collateral_for_dsc(weth, COLLATERAL_AMOUNT,
            AMOUNT_TO_MINT)
        user_dsc_balance = dsc.balanceOf(some_user)
        assert user_dsc_balance == 0
        user_weth_balance = weth.balanceOf(some_user)
        assert user_weth_balance == initial_user_weth_balance
```

Recommended Mitigation:

The health factor check should happen before calculations that are related with collateral redemption or token burning:

```
def redeem_collateral_for_dsc(
    token_collateral_address: address,
    amount_collateral: uint256,
    amount_dsc_to_burn: uint256,
):
+    self._revert_if_health_factor_is_broken(msg.sender)
    self._burn_dsc(amount_dsc_to_burn, msg.sender, msg.sender)
    self._redeem_collateral(
        token_collateral_address, amount_collateral, msg.sender, msg.
            sender
    )
-    self._revert_if_health_factor_is_broken(msg.sender)

...
@external
def redeem_collateral(
    token_collateral_address: address, amount_collateral: uint256
):
+    self._revert_if_health_factor_is_broken(msg.sender)
    self._redeem_collateral(
        token_collateral_address, amount_collateral, msg.sender, msg.
            sender
    )
-    self._revert_if_health_factor_is_broken(msg.sender)

...
@external
def burn_dsc(amount_dsc_to_burn: uint256):
+    self._revert_if_health_factor_is_broken(msg.sender)
    self._burn_dsc(amount_dsc_to_burn, msg.sender, msg.sender)
-    self._revert_if_health_factor_is_broken(msg.sender)
```

[H-2] Incorrect Liquidation Bonus Calculation Causes Liquidation Failures

Description: The liquidation mechanism in the protocol miscalculates the liquidation bonus by subtracting the bonus collateral from the user's deposited collateral without considering its effect on the user's health factor. This miscalculation results in a consistently reduced health factor for the user after liquidation, even when the intended goal is to improve it. Additionally, if the user's collateral balance is insufficient to cover both the debt and the liquidation bonus, the function reverts due to underflow, rendering the entire liquidation process non-functional.

Impact:

- The incorrect bonus calculation prevents successful liquidations, making it impossible for liq-

uidators to operate effectively.
- If the user's collateral balance is lower than `token_amount_from_debt_covered + bonus_collateral`, the protocol reverts due to underflow.

Proof of Concept:

Protocol's own test is a proof of this - run the following test with a small change, and it will revert due to not improving health factor:

```
def test_must_improve_health_factor_on_liquidation(
    some_user, liquidator, weth, eth_usd, wbtc, btc_usd
):
    # Setup mock DSC
    mock_dsc = mock_more_debt_dsc.deploy(eth_usd)
    token_addresses = [weth, wbtc]
    feed_addresses = [eth_usd, btc_usd]
    dsce = dsc_engine.deploy(token_addresses, feed_addresses, mock_dsc)
    mock_dsc.set_minter(dsce.address, True)
    mock_dsc.transfer_ownership(dsce)

    # Setup user position
    with boa.env.prank(some_user):
        weth.approve(dsce, COLLATERAL_AMOUNT)
        dsce.deposit_collateral_and_mint_dsc(weth, COLLATERAL_AMOUNT,
           AMOUNT_TO_MINT)

    # Setup liquidator
    collateral_to_cover = to_wei(1, "ether")
    weth.mint(liquidator, collateral_to_cover)

    with boa.env.prank(liquidator):
        weth.approve(dsce, collateral_to_cover)
        debt_to_cover = to_wei(10, "ether")
        dsce.deposit_collateral_and_mint_dsc(weth, collateral_to_cover,
            AMOUNT_TO_MINT)
        mock_dsc.approve(dsce, debt_to_cover)

        # Update price to trigger liquidation
        eth_usd_updated_price = 18 * 10**8  # 1 ETH = $18
        eth_usd.updateAnswer(eth_usd_updated_price)

-         with boa.reverts("DSCEngine__HealthFactorNotImproved"):
            dsce.liquidate(weth, some_user, debt_to_cover)
```

This test proves that underflow can also happen:

```
def test_liquidation_underflow(
    some_user, liquidator, weth, eth_usd, dsc, dsce
):
    # User setup
    with boa.env.prank(some_user):
        weth.approve(dsce, COLLATERAL_AMOUNT)
        dsce.deposit_collateral_and_mint_dsc(weth, COLLATERAL_AMOUNT,
            AMOUNT_TO_MINT)

    # Liquidator setup
    weth.mint(liquidator, COLLATERAL_TO_COVER)
    with boa.env.prank(liquidator):
        weth.approve(dsce, COLLATERAL_TO_COVER)
        dsce.deposit_collateral_and_mint_dsc(weth, COLLATERAL_TO_COVER,
            AMOUNT_TO_MINT)

    # Drop ETH Price to $10 per ETH
    eth_usd.updateAnswer(10 * 10**8)

    # Expected Revert: Liquidation Bonus Causes Underflow
    debt_to_cover = AMOUNT_TO_MINT
    with boa.env.prank(liquidator), boa.reverts("safesub"):
        dsce.liquidate(weth, some_user, debt_to_cover)
```

Recommended Mitigation: Consider removing the bonus mechanism entirely. If bonus is a must, consider creating a governance-controlled reward system, and do not tie bonus to user's collateral.

Low

[L-1] Missing Token Address in `DSCEngine__CollateralDeposited` Event

Description: The `DSCEngine__CollateralDeposited` event lacks the token address, making it difficult to determine which asset was used as collateral. Including the token address would enhance the event's completeness, allowing external systems to track collateral deposits accurately.

Impact: Without the token address, off-chain systems relying on this event cannot reconstruct the correct state of the contract, potentially causing inconsistencies in portfolio tracking, analytics, and state synchronization.

Recommended Mitigation: Add the token field to the event definition:

```
event CollateralDeposited:
     user: indexed(address)
     amount: indexed(uint256)
+    token: address
```

[L-2] Irrelevant health factor check for liquidator is a waste of gas

Description: In `DSCEngine__liquidate` function a health check is performed at the very end for the liquidator itself. As no new DSC tokens are being minted for the liquidator, this check is irrelevant.

Impact: The function does waste gas as providing no value.

Recommended Mitigation:

```
@external
def liquidate(collateral: address, user: address, debt_to_cover:
    uint256):
     assert debt_to_cover > 0, "DSCEngine__NeedsMoreThanZero"
     starting_user_health_factor: uint256 = self._health_factor(user)
     assert (
         starting_user_health_factor < MIN_HEALTH_FACTOR
     ), "DSCEngine__HealthFactorOk"
     token_amount_from_debt_covered: uint256 = self.
        _get_token_amount_from_usd(
         collateral, debt_to_cover
     )
     bonus_collateral: uint256 = (
         token_amount_from_debt_covered * LIQUIDATION_BONUS
     ) // LIQUIDATION_PRECISION
     self._redeem_collateral(
         collateral,
         token_amount_from_debt_covered + bonus_collateral,
         user,
         msg.sender,
     )
     self._burn_dsc(debt_to_cover, user, msg.sender)
     ending_user_health_factor: uint256 = self._health_factor(user)
     assert (
         ending_user_health_factor > starting_user_health_factor
     ), "DSCEngine__HealthFactorNotImproved"
-     self._revert_if_health_factor_is_broken(msg.sender)
```

[L-3] Re-entrancy attacks could be possible if `WETH` or `WBTC` contracts get upgraded or go rogue

Description: Both tokens used for collateral are upgradeable, and in case they go rogue or get upgraded with malicious functionality, they open up re-entrancy attacks in several functions. For example, an attacker could call `DSCEngine__liquidate` and start liquidating all the users that have lower health factor and absorb all the collateral without burning his own tokens.

Impact: If this situation would happen, the consequences of the protocol are disastrous. However, the likelihood is very minimal.

Informational

[I-1] Missing zero address checks in the `DSCEngine____init__` function

Recommended Mitigation:

```
@deploy
def __init__(
    token_addresses: address[2],
    price_feed_addresses: address[2],
    dsc_address: address,
):
+    assert token_addresses[0] != ZERO_ADDRESS, "DSCEngine__ZeroAddress"
+    assert token_addresses[1] != ZERO_ADDRESS, "DSCEngine__ZeroAddress"
+    assert price_feed_addresses[0] != ZERO_ADDRESS, "DSCEngine__ZeroAddress"
+    assert price_feed_addresses[1] != ZERO_ADDRESS, "DSCEngine__ZeroAddress"
+    assert dsc_address != ZERO_ADDRESS, "DSCEngine__ZeroAddress"

    DSC = i_decentralized_stable_coin(dsc_address)
    COLLATERAL_TOKENS = token_addresses
    # This is gas inefficient!
    self.token_address_to_price_feed[token_addresses[0]] = price_feed_addresses[
        0
    ]
    self.token_address_to_price_feed[token_addresses[1]] = price_feed_addresses[
        1
    ]
```

[I-2] All state-changing external functions are missing NATSPEC documentation making their description less verbose

Description: It is good practice to include basic NATSPEC documentation for every important state-changing function to allow party reading the code get a better and faster understanding of what each function is intended to do.

Impact: Not having ANY NATSPEC documentation might cause confusion for developers in terms of functionality, intended business logic and can take longer to review and understand the code.