SMART CONTRACT AUDIT REPORT

for

OLYMPUSDAO

Prepared By: Shuxiao Wang

PeckShield

April 9, 2021

Document Properties

Client	OlympusDAO
Title	Smart Contract Audit Report
Target	OlympusDAO
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 9, 2021	Xuxian Jiang	Final Release
1.0-rc	April 3, 2021	Xuxian Jiang	Release Candidate #1
0.3	April 1, 2021	Xuxian Jiang	Additional Findings #2
0.2	March 28, 2021	Xuxian Jiang	Additional Findings #1
0.1	March 25, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the OlympusDAO protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About OlympusDAO

Olympus is an algorithmic currency protocol based on the OHM token. It introduces unique economic and game-theoretic dynamics into the market through asset-backing and protocol owned value. It is a value-backed, self-stabilizing, and decentralized stablecoin with unique collateral backing and algorithmic incentive mechanism. Different from existing stablecoin solutions, it is proposed as a non-pegged stablecoin by exploring a radical opportunity to achieve stability while eliminating dependence on fiat currencies.

The basic information of the OlympusDAO protocol is as follows:

Table 1.1: Basic Information of The OlympusDAO Protocol

Item	Description
Issuer	OlympusDAO
Website	https://olympusdao.eth.link/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 9, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/OlympusDAO/olympus.git (cdd4afe)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the OlympusDAO implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	2	■■
Undetermined	1	■
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, 2 informational recommendations, and 1 issue with undetermined severity.

Table 2.1: Key OlympusDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Caller Authentication Of sOlympusERC20::rebase()	Security Features	Fixed
PVE-002	Undetermined	Potential Rebasing Perturbation	Time And State	Confirmed
PVE-003	Informational	Simplified Logic In BondingCalculator::_principleValuation()	Coding Practices	Fixed
PVE-004	Medium	Proper Initialization Enforcement In sOlympus::setStakingContract()	Security Features	Fixed
PVE-005	Low	Improved Decimal Conversion in depositReserves()	Business Logic	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-007	Informational	Redundant Code Removal	Coding Practices	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Caller Authentication Of sOlympusERC20::rebase()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: sOlympusERC20
- Category: Security Features [7]
- CWE subcategory: CWE-282 [2]

Description

In the Olympus protocol, one core component is the Staking contract that allows participants to stake OHM tokens and get sOHM in return. The sOHM token is a rebasing, ERC20-compliant one that evenly distributes profits to staking users. While examining the rebasing logic, we notice an authentication issue that needs to be resolved.

To elaborate, we show below the rebase() implementation. This function follows a similar implementation from AmpleForth [1] with internal Gon-based representation. However, we notice this function is protected with a onlyMonetaryPolicy() modifier. This modifier has the requirement of require(msg.sender == monetaryPolicy), which in essence restricts the caller to be from monetaryPolicy.

```
    function rebase(uint256 olyProfit) public onlyMonetaryPolicy() returns (uint256) {
        uint256 _rebase;

        if (olyProfit == 0) {
            emit LogRebase(block.timestamp, _totalSupply);
            return _totalSupply;
        }

        if(circulatingSupply() > 0 ){
            _rebase = olyProfit.mul(_totalSupply).div(circulatingSupply());
        }
```

[1] The AmpleForth protocol can be accessed at https://www.ampleforth.org/

```
        else {
            _rebase = olyProfit;
        }

        _totalSupply = _totalSupply.add(_rebase);


        if (_totalSupply > MAX_SUPPLY) {
            _totalSupply = MAX_SUPPLY;
        }

        _gonsPerFragment = TOTAL_GONS.div(_totalSupply);

        emit LogRebase(block.timestamp, _totalSupply);
        return _totalSupply;
    }
```

Listing 3.1: sOlympusERC20::rebase()

Meanwhile, our analysis shows that the only possible caller of `rebase()` is the `Staking` contract (line 723). With that, there is a need to adjust the modifier to be `onlyStakingContract`. Certainly, a possible solution will require the `Staking` contract to be the same as `monetaryPolicy`.

Recommendation Properly authenticating the caller of `rebase` to be `stakingContract`, not `monetaryPolicy`. Or consider the merge of `stakingContract` and `monetaryPolicy` as the same entity.

Status This issue has been fixed for v2.

3.2 Potential Rebasing Perturbation

- ID: PVE-002
- Severity: Undetermined
- Likelihood: -
- Impact: -
- Target: `OlympusStaking`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

As mentioned earlier, the `Olympus` protocol implements a unique expansion and contraction mechanism in order to be a stablecoin. In the following, we examine the rebasing mechanism implemented in the protocol.

To elaborate, we show below the `_distributeOHMProfits()` routine that triggers `sOHM`-rebasing so that the accumulated profits can be evenly distributed to circulating sOHM. Note that the rebasing operation will not be triggered until the current block height reaches the specified `nextEpochBlock` number.

```
    // triggers rebase to distribute accumulated profits to circulating sOHM
    function _distributeOHMProfits() internal {
        if( nextEpochBlock <= block.number ) {
            IOHMandsOHM(sOHM).rebase(ohmToDistributeNextEpoch);
            uint256 _ohmBalance = IOHMandsOHM(ohm).balanceOf(address(this));
            uint256 _sohmSupply = IOHMandsOHM(sOHM).circulatingSupply();
            ohmToDistributeNextEpoch = _ohmBalance.sub(_sohmSupply);
            nextEpochBlock = nextEpochBlock.add( epochLengthInBlocks );
        }
    }
```

Listing 3.2: OlympusStaking::_distributeOHMProfits()

With that, it is possible that right before `nextEpochBlock` is reached, a user may choose to stake (or unstake) to increase (decrease) the circulating supply of `sOHM`. Either way, the current rebasing operation as well as the `ohmToDistributeNextEpoch` amount may be influenced.

Note that this is a common sandwich-based arbitrage behavior plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user. We need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich arbitrage behavior to better protect the rebasing operation in `Olympus`.

Status The issue has been confirmed.

3.3 Simplified Logic In BondingCalculator::_principleValuation()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BondingCalculator`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

Besides staking, the `Olympus` protocol provides the bond mechanism as the secondary strategy to provide a more conservative and reliable return. Specifically, this mechanism quotes the bonder with terms for a trade at a future date and the actual bond amount depends on a bonding curve. There are two main factors, `BCV` and `vesting term`. The first factor allows to scale the rate at which bond premiums increase. A higher `BCV` means a lower discount for bonders and less inflation. A lower `BCV` means a higher capacity for bonders and less protocol profit. The `vesting term` determines how long it takes for bonds to become redeemable. A longer term means lower inflation and lower bond demand.

While analyzing the bonding curve, we observe an optimization in the internal helper `_principleValuation()`. This helper is used to determine the LP share values according to a conservative formula. In the actual calculation at line 628, the ending scaling factor of `div(1e10).mul(10)` can be simplified as `div(1e9)`.

```
621     // Values LP share based on formula
622     // returns principleValuation = 2sqrt(constant product) * (% ownership of total LP)
623     // uint k_ = constant product of liquidity pool
624     // uint amountDeposited_ = amount of LP token
625     // uint totalSupplyOfTokenDeposited = total amount of LP
626     function _principleValuation( uint k_, uint amountDeposited_, uint
            totalSupplyOfTokenDeposited_ ) internal pure returns ( uint principleValuation_
            ) {
627         // *** When deposit amount is small does not pick up principle valuation *** \\
628         principleValuation_ = k_.sqrrt().mul(2).mul( FixedPoint.fraction(
                amountDeposited_, totalSupplyOfTokenDeposited_ ).decode112with18().div( 1e10
                 ).mul( 10 ) );
629     }
```

Listing 3.3: BondingCalculator :: _principleValuation ()

Recommendation Simplify the scaling operation on the helper routine to calculate the principle valuation.

Status This issue has been fixed for v2.

3.4 Proper Initialization Enforcement In sOlympus::setStakingContract()

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: sOlympus
- Category: Security Features [7]
- CWE subcategory: CWE-282 [2]

Description

As mentioned in Section 3.1, one core component of Olympus is the Staking contract that allows participants to stake OHM tokens and get sOHM in return. While examining the sOHM token contract, we notice a privileged operation setStakingContract() that is designed to initialize the stakingContract address and its internal Gon balance.

To elaborate, we show below the setStakingContract() implementation from the sOHM token contract, i.e., sOlympus. While it indeed properly sets up the stakingContract address and initializes the Gon balance, this initialization operation should only occur once. Otherwise, the sOHM supply may go awry, resulting in protocol-wide instability.

```
    function setStakingContract( address newStakingContract_ ) external onlyOwner() {
      stakingContract = newStakingContract_;
      _gonBalances[stakingContract] = TOTAL_GONS;
    }
```

Listing 3.4: sOlympus::setStakingContract()

Recommendation Ensure the setStakingContract() can only be initialized once.

Status This issue has been fixed for v2.

3.5 Improved Decimal Conversion in depositReserves()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Olympus` protocol has a treasury contract, i.e., `Vault`, that allows for taking reserve tokens (e.g., `DAI`) and minting managed tokens (e.g., `OHM`). The treasury contact can also take the principle tokens (e..g, `OHM-DAI SLP`) and mint the managed tokens according to the bonding curve-based principle evaluation. In the following, we examine the conversions from reserve tokens to managed tokens.

The conversion logic is implemented in the `depositReserves()` routine. To elaborate, we show below its code. It comes to our attention that the conversion logic is coded as `amount_.div(10 ** IERC20(getManagedToken).decimals())`. Note that the given amount is denominated at the reserve token `DAI` and the minted amount is in the unit of managed token (`OHM`). With that, the proper calculation of the converted amount should be the following: `amount_.mul(10 ** IERC20(getManagedToken).decimals()).div(10**IERC20(getReserveToken).decimals())`.

```
function depositReserves( uint amount_ ) external returns ( bool ) {
    require( isReserveDepositor[msg.sender] == true, "Not allowed to deposit" );
    IERC20( getReserveToken ).safeTransferFrom( msg.sender, address(this), amount_ );
    IERC20Mintable( getManagedToken ).mint( msg.sender, amount_.div( 10 ** IERC20(
        getManagedToken ).decimals() ) );
    return true;
}
```

Listing 3.5: Vault :: isReserveDepositor ()

Fortunately, the managed token `OHM` has the decimal of 9 and the reserve token `DAI` has the decimal of 18. As a result, it still results in the same converted (absolute) amount. However, the revised conversion logic is generic in accommodating other token setups, especially when the managed token does not have 9 as its decimal.

Recommendation Revise the `isReserveDepositor()` logic by following the correct decimal conversion.

Status This issue has been fixed for v2.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: OlympusERC20
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the OlympusDAO protocol, there is a privileged owner account plays a critical role in governing the treasury contract (Vault) and regulating the OHM token contract. In the following, we show representative privileged operations in the Olympus protocol.

```
function setDAOWallet( address newDAOWallet_ ) external onlyOwner() returns ( bool ) {
  daoWallet = newDAOWallet_;
  return true;
}

function setStakingContract( address newStakingContract_ ) external onlyOwner()
    returns ( bool ) {
  stakingContract = newStakingContract_;
  return true;
}

function setLPRewardsContract( address newLPRewardsContract_ ) external onlyOwner()
    returns ( bool ) {
  LPRewardsContract = newLPRewardsContract_;
  return true;
}

function setLPProfitShare( uint newDAOProfitShare_ ) external onlyOwner() returns (
    bool ) {
  LPProfitShare = newDAOProfitShare_;
  return true;
}
```

Listing 3.6: Example Privileged Operations in Vault

```
function setVault( address vault_ ) external onlyOwner() returns ( bool ) {
  _vault = vault_;

  return true;
}


function mint(address account_, uint256 amount_) external onlyVault() {
  _mint(account_, amount_);
```

```
}
```

Listing 3.7: Example Privileged Operations in OlympusERC20Token

We emphasize that the privilege assignment with various factory contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure.

We point out that a compromised `owner` account would allow the attacker to change current `vault` to mint arbitrary number of `OHM` or change other settings (e.g., `stakingContract`) to steal funds of currently staking users, which directly undermines the integrity of the `Olympus` protocol.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. It is in place with the purpose as being a helper function to facilitate reward distribution. Note this functionality has been offloaded to a separate contract. And all of these have been removed for v2.

3.7 Redundant Code Removal

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `OlympusERC20`, `Vault`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

OlympusDAO makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the `Vault` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine the `isReserveToken` state variable, it is designed to determine whether a given token is a reserve token. However, apparently, the current version does not make use of this state variable.

```
contract TWAPOracleUpdater is ERC20Permit, VaultOwned {

  using EnumerableSet for EnumerableSet.AddressSet;

```

```
event TWAPOracleChanged( address indexed previousTWAPOracle, address indexed newTWAPOracle );
event TWAPEpochChanged( uint previousTWAPEpochPeriod, uint newTWAPEpochPeriod );
event TWAPSourceAdded( address indexed newTWAPSource );
event TWAPSourceRemoved( address indexed removedTWAPSource );

EnumerableSet.AddressSet private _dexPoolsTWAPSources;

ITWAPOracle public twapOracle;

uint public twapEpochPeriod;

constructor(
      string memory name_,
      string memory symbol_,
      uint8 decimals_
  ) ERC20(name_, symbol_, decimals_) {
  }

function changeTWAPOracle( address newTWAPOracle_ ) external onlyOwner() {
  emit TWAPOracleChanged( address(twapOracle), newTWAPOracle_);
  twapOracle = ITWAPOracle( newTWAPOracle_ );
}

function changeTWAPEpochPeriod( uint newTWAPEpochPeriod_ ) external onlyOwner() {
  require( newTWAPEpochPeriod_ > 0, "TWAPOracleUpdater: TWAP Epoch period must be greater than 0." );
  emit TWAPEpochChanged( twapEpochPeriod, newTWAPEpochPeriod_ );
  twapEpochPeriod = newTWAPEpochPeriod_;
}

function addTWAPSource( address newTWAPSourceDexPool_ ) external onlyOwner() {
  require( _dexPoolsTWAPSources.add( newTWAPSourceDexPool_ ), "OlympusERC20TOken: TWAP Source already stored." );
  emit TWAPSourceAdded( newTWAPSourceDexPool_ );
}

function removeTWAPSource( address twapSourceToRemove_ ) external onlyOwner() {
  require( _dexPoolsTWAPSources.remove( twapSourceToRemove_ ), "OlympusERC20TOken: TWAP source not present." );
  emit TWAPSourceRemoved( twapSourceToRemove_ );
}

function _uodateTWAPOracle( address dexPoolToUpdateFrom_, uint twapEpochPeriodToUpdate_ ) internal {
  if ( _dexPoolsTWAPSources.contains( dexPoolToUpdateFrom_ )) {
    twapOracle.updateTWAP( dexPoolToUpdateFrom_, twapEpochPeriodToUpdate_ );
  }
}

function _beforeTokenTransfer( address from_, address to_, uint256 amount_ ) internal override virtual {
```

```
      if( _dexPoolsTWAPSources.contains( from_ ) ) {
        _uodateTWAPOracle( from_, twapEpochPeriod );
      } else {
        if ( _dexPoolsTWAPSources.contains( to_ ) ) {
          _uodateTWAPOracle( to_, twapEpochPeriod );
        }
      }
    }
}
```

Listing 3.8: The TWAPOracleUpdater Contract

Moreover, the current implementation includes a contract `TWAPOracleUpdater` that is supposed to be inherited by the `OHM` token contract. However, this `TWAPOracleUpdater` contract is currently not used and thus can be safely removed.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been confirmed. The team has integrated TWAP code, which will be utilized in future versions.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Olympus`, which utilizes the protocol owned value to enable price consistency and scarcity within an infinite supply system. During the audit, we notice that the current implementation still remains to be completed, though the overall code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[11] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[13] PeckShield. PeckShield Inc. https://www.peckshield.com.