SMART CONTRACT AUDIT REPORT

for

Wombat Volatile Pool

Prepared By: Xiaomi Huang

PeckShield

August 10, 2024

Document Properties

Client	Wombat Exchange
Title	Smart Contract Audit Report
Target	Wombat Volatile Pool
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 10, 2024	Xuxian Jiang	Final Release
1.0-rc	April 10, 2024	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Volatile Pool support in Wombat, 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Wombat

Wombat is a BNB-native stableswap protocol with open-liquidity pools, low slippage and single-sided staking. It brings greater capital efficiency to fuel DeFi growth and adoption. This audit covers the latest volatile pool support with additional features, including efficient re-pegging, dynamic haircut, and arbitrage block protection. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Wombat Volatile Pool

Item	Description
Name	Wombat Exchange
Website	https://www.wombat.exchange/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 10, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the following contracts, i.e., DynamicFeeHelper.sol, RepegHelper.sol, and VolatilePool.sol. Moreover, this audit assumes the correctness of the underlying AMM invariant and its implementation.

- https://github.com/wombat-exchange/wombat.git (843e30d)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/wombat-exchange/wombat.git (7f15cac)

1.2 About PeckShield

PeckShield Inc. [9] 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 the OWASP Risk Rating Methodology [8]:

- 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

To evaluate the risk, we go through a checklist of 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 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) [7], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 implementation of the volatile pool support in `Wombat`. 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 logic, 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	3	■■■
Informational	0	
Total	5	

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 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 and 3 low-severity vulnerabilities.

Table 2.1: Key Wombat Volatile Pool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in Current Pools	Coding Practices	Confirmed
PVE-002	Low	Inconsistent Adjustment Step Calculation in RepegHelper	Coding Practices	Resolved
PVE-003	Low	Lack of Token Support With Large Decimals in _findUpperBound()	Business Logic	Resolved
PVE-004	Medium	Inaccurate Tip Bucket Balance Calculation in VolatilePool	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Constructor/Initialization Logic in Current Pools

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, each `Pool` contract is instantiated as a proxy with an actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function initialize(uint256 ampFactor_, uint256 haircutRate_) public override {
        super.initialize(ampFactor_, haircutRate_);

        repegData.adjustmentStep = 0.0005e18;
        repegData.oracleEmaHalfTime = 600;
        repegData.psi = 5;
        repegData.lastOracleTimestamp = uint128(block.timestamp);

        poolData.reserveRateForRepegging = 0.5e18;

        lastSwapTimestamp = block.timestamp;
    }
```

Listing 3.1: `VolatilePool::initialize()`

Recommendation Improve the above-mentioned constructor routines in all existing upgradeable pools.

Status The issue has been confirmed.

3.2 Inconsistent Adjustment Step Calculation in RepegHelper

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RepegHelper`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Wombat` protocol is well-documented with the extensive use of `NatSpec` comments to provide rich documentation for functions, return variables and others. In the process of analyzing current `NatSpec` comments, we notice the presence of an inconsistency with the code implementation.

To elaborate, we show below the `_getNormalizedAdjustmentStep()` function from the `RepegHelper` contract. This function is designed to calculate the relative distance of change of `priceScale` towards `oraclePrice` after re-pegging. Our analysis shows it in essence computes the minimum between `(uint256(myStruct.adjustmentStep)).wdiv(norm)` (line 304) and `WAD / myStruct.psi` (line 305). However, the related white paper indicates the maximum between the two is needed (pp.4).

```
303     function _getNormalizedAdjustmentStep(RepegData storage myStruct, uint256 norm)
            internal view returns (uint256) {
304         uint256 value = (uint256(myStruct.adjustmentStep)).wdiv(norm);
305         if (value > WAD / myStruct.psi) {
306             // upper bounded by WAD / psi, which is 0.2 WAD by default
307             return WAD / myStruct.psi;
308         } else {
309             return value;
310         }
311     }
```

Listing 3.2: `RepegHelper::_getNormalizedAdjustmentStep()`

Recommendation Revisit the above mentioned logic to consistently compute the intended relative distance of change.

Status The issue has been fixed by the following commit: `1d4d18d`.

3.3 Lack of Token Support With Large Decimals in _findUpperBound()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CoreV4`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Wombat` protocol by design supports underlying tokens without restriction on their liquidity and associated decimals. While reviewing this design, we notice current implementation may not support underlying tokens with decimals larger than 18.

Specifically, we show below the related routine `_findUpperBound()`, which is used to perform binary search to find the upper bound of `fromAmount` required to swap `fromAsset` to `toAmount` of `toAsset`. We notice the use of `toWadFactor`, a local variable to record the scaling factor based on the `fromAsset`. However, when `fromAsset` has a large decimals (>18), the computed `toWadFactor` becomes 0, which fails the intended computation in `_findUpperBound()`.

```
    function _findUpperBound(
        PoolV4Data storage poolData,
        IAsset fromAsset,
        IAsset toAsset,
        uint256 toAmount,
        uint256 scaleFactor,
        uint256 haircutRate
    ) private view returns (uint256 upperBound) {
        uint8 decimals = fromAsset.underlyingTokenDecimals();
        uint256 toWadFactor = DSMath.toWad(1, decimals);
        // the search value uses the same number of digits as the token
        uint256 high = (uint256(fromAsset.liability()).wmul(poolData.endCovRatio) -
            fromAsset.cash()).fromWad(decimals);
        uint256 low = 1;

        // verify 'high' is a valid upper bound
        uint256 quote;
        (quote, ) = quoteSwapForHighCovRatioPool(
            poolData,
            fromAsset,
            toAsset,
            (high * toWadFactor).toInt256(),
            scaleFactor,
            haircutRate
        );
        if (quote < toAmount) revert CORE_COV_RATIO_LIMIT_EXCEEDED();
```

```

        // Note: we might limit the maximum number of rounds if the request is always
            rejected by the RPC server
        while (low < high) {
            uint256 mid = (low + high) / 2;
            (quote, ) = quoteSwapForHighCovRatioPool(
                poolData,
                fromAsset,
                toAsset,
                (mid * toWadFactor).toInt256(),
                scaleFactor,
                haircutRate
            );
            if (quote >= toAmount) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return high * toWadFactor;
    }
```

Listing 3.3: `CoreV4::_findUpperBound()`

Recommendation Revisit the above routine to adjust the `toWadFactor` calculation.

Status The issue has been fixed by the following commit: `7f15cac`.

3.4 Inaccurate Tip Bucket Balance Calculation in VolatilePool

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VolatilePool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `VolatilePool`support allows for efficient re-pegging, which by design allocates certain reserve from the accumulated fee. The reserve fee is saved in the pool. However, this reserve fee is not excluded from the tip bucket balance calculation.

To elaborate, we show below the code snippet from the `tipBucketBalance()` routine. This routine has a rather straightforward logic in deducting the `cash` amount (line 667) and `collected fee` (line 668) from the current balance. Notice the reserve fee is also part of the current balance. With that, there is a need to deduct the repeg-related reserve fee as well.

```
    function tipBucketBalance(PoolV4Data storage poolData, IERC20 token) external view
        returns (uint256 balance) {
        IAsset asset = poolData.assets.assetOf(token);
        return
            asset.underlyingTokenBalance().toWad(asset.underlyingTokenDecimals()) -
            asset.cash() -
            poolData.feeAndReserve[asset].feeCollected;
    }
```

Listing 3.4: VolatilePool::tipBucketBalance()

Recommendation Revisit the above logic to compute the intended tip bucket balance.

Status The issue has been fixed by the following commit: `7f15cac`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Wombat` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the protocol-wide operations (e.g., manage pool assets, configure pool parameters, collect tip bucket, and execute other privileged operations). Our analysis shows that this privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

```
    function setPriceAnchor(IVolatileAsset priceAnchor_) external onlyOwner {
        RepegData storage myStruct = repegData;
        if (address(priceAnchor_) == address(0)) revert POOL__INVALID_PARAMETER();
        if (address(myStruct.priceAnchor) != address(0)) revert POOL__INVALID_PARAMETER
            ();

        myStruct.priceAnchor = priceAnchor_;
        emit SetPriceAnchor(priceAnchor_);
    }

    function setAdjustmentStep(uint64 adjustmentStep_) external onlyOwner {
        if (adjustmentStep_ > WAD) revert POOL__INVALID_PARAMETER();
        RepegData storage myStruct = repegData;
        myStruct.adjustmentStep = adjustmentStep_;
        emit SetAdjustmentStep(adjustmentStep_);
    }
```

```
    function setOracleEmaHalfTime(uint128 oracleEmaHalfTime_) external onlyOwner {
        if (oracleEmaHalfTime_ < 60) revert POOL__INVALID_PARAMETER();
        RepegData storage myStruct = repegData;
        myStruct.oracleEmaHalfTime = oracleEmaHalfTime_;
        emit SetOracleEmaHalfTime(oracleEmaHalfTime_);
    }

    function setPsi(uint32 psi_) external onlyOwner {
        if (psi_ == 0) revert POOL__INVALID_PARAMETER();
        repegData.psi = psi_;
        emit SetPsi(psi_);
    }

    function setReserveRateForRepegging(uint256 reserveRate_) external onlyOwner {
        if (reserveRate_ > 1e18) revert POOL__INVALID_PARAMETER();
        poolData.reserveRateForRepegging = reserveRate_;
        emit SetReserveRateForRepegging(reserveRate_);
    }

    function setDynamicFeeParam(
        uint128 haircutVolatilityMax_,
        uint128 haircutImbalanceMax_,
        int128 haircutVolatilityKV1_,
        int128 haircutVolatilityBetaV1_,
        int128 haircutVolatilityKV2_,
        int128 haircutVolatilityBetaV2_,
        int128 haircutImbalanceSmallTheta_
    ) external onlyOwner {
        dynamicFeeConfig.haircutVolatilityMax = haircutVolatilityMax_;
        dynamicFeeConfig.haircutImbalanceMax = haircutImbalanceMax_;

        dynamicFeeConfig.haircutVolatilityKV1 = haircutVolatilityKV1_;
        dynamicFeeConfig.haircutVolatilityBetaV1 = haircutVolatilityBetaV1_;
        dynamicFeeConfig.haircutVolatilityKV2 = haircutVolatilityKV2_;
        dynamicFeeConfig.haircutVolatilityBetaV2 = haircutVolatilityBetaV2_;
        dynamicFeeConfig.haircutImbalanceSmallTheta = haircutImbalanceSmallTheta_;
        // Not implemented
        // dynamicFeeConfig.haircutImbalanceBigTheta = haircutImbalanceBigTheta_;

        emit SetDynamicFeeParam(
            haircutVolatilityMax_,
            haircutImbalanceMax_,
            haircutVolatilityKV1_,
            haircutVolatilityBetaV1_,
            haircutVolatilityKV2_,
            haircutVolatilityBetaV2_,
            haircutImbalanceSmallTheta_
        );
    }
```

Listing 3.5: Example Privileged Operations in `VolatilePool`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It will be worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 mitigated as the team confirmed they use multi-sig for the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the volatile pool support in Wombat. The support adds additional features, including efficient re-pegging, dynamic haircut, and arbitrage block protection. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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