



SMART CONTRACT AUDIT REPORT

for

GEARBOX



Prepared By: Shuxiao Wang

PeckShield
May 3, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 3, 2021	Xuxian Jiang	Final Release
1.0-rc1	April 24, 2021	Xuxian Jiang	Release Candidate #1
0.3	April 18, 2021	Xuxian Jiang	Add More Findings #2
0.2	April 13, 2021	Xuxian Jiang	Add More Findings #1
0.1	April 9, 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	4
1.1	About Gearbox	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Accommodation of Non-ERC20-Compliant Tokens	12
3.2	Denial-Of-Service Against of WETHGateway	14
3.3	Overwritten Virtual Accounts Without Repayment	15
3.4	Improved Logic Of allowToken()	17
3.5	Potential DoS Against PoolService	18
3.6	Improved Sanity Checks For System Parameters	19
3.7	Possible Sandwich/MEV Attacks For Reduced Conversion	20
3.8	Improper Interest Collection In lendVirtualAccount()	23
3.9	Asset Consistency Between VAMFilter And PoolService	24
4	Conclusion	26
	References	27

1 | Introduction

Given the opportunity to review the **Gearbox** design document and related smart contract source code, 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 Gearbox

Gearbox protocol is aiming to enhance capital efficiency in the DeFi space with the introduction of new DeFi primitives for undercollateralized interactions with other DeFi lego blocks which can be used for decentralized margin trading, leveraged yield farming, etc. This is made possible with **Virtual Accounts**, which are agent-oriented isolated smart contracts, and liquidation thresholds protecting the bottom line of the deposited capital. Overall, **Gearbox** does not only give users and other decentralized protocols access to leverage their trading and farming positions (and therefore maximize their profit), but also ensures a non-custodial, transparent, and composable design of the protocol.

The basic information of Gearbox is as follows:

Table 1.1: Basic Information of Gearbox

Item	Description
Issuer	Gearbox
Website	https://gearbox.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 3, 2021

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

in this audit. Note that Gearbox assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- <https://github.com/Gearbox-protocol/gearbox-v2.git> (229de35)

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

- <https://github.com/Gearbox-protocol/gearbox-v2.git> (db4a5cf)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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.

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

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

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.




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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Gearbox protocol. 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	1	
Medium	4	
Low	4	
Informational	0	
Total	9	

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 1 high-severity vulnerability, 4 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key Gearbox Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-002	Medium	Denial-Of-Service Against of WETHGateway	Business Logic	Fixed
PVE-003	High	Overwritten Virtual Accounts Without Re-payment	Business Logic	Fixed
PVE-004	Low	Improved Logic Of allowToken()	Business Logic	Fixed
PVE-005	Medium	Potential DoS Against PoolService	Business Logic	Fixed
PVE-006	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-007	Medium	Possible Sandwich/MEV Attacks For Reduced Conversion	Time and State	Fixed
PVE-008	Medium	Improper Interest Collection In lendVirtualAccount()	Business Logic	Fixed
PVE-009	Low	Asset Consistency Between VAMFilter And PoolService	Coding Practices	Fixed

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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VanillaVirtualAccount
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195  * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
196  * @param _spender The address which will spend the funds.
197  * @param _value The amount of tokens to be spent.
198  */
199  function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201      // To change the approve amount you first have to reduce the addresses '
202      // allowance to zero by calling 'approve(_spender, 0)' if it is not
203      // already 0 to mitigate the race condition described here:
204      // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205      require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

```

```

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }

```

Listing 3.1: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `VanillaVirtualAccount::approveTokenForContract()` routine as an example. This routine is designed to approve a specific token for swap contract. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 112): the first one reduces the allowance to 0; and the second one sets the new allowance.

```

101     /**
102     * Approves particular token for swap contract
103     * @param token ERC20 token for allowance
104     * @param swapContract Swap contract address
105     */
106     function approveTokenForContract(address token, address swapContract)
107     external
108     override
109     virtualAccountManagerOnly
110     {
111         // For audit: is it okay to use approve instead of safeApprove here?
112         IERC20(token).approve(swapContract, Constants.MAX_INT);
113     }

```

Listing 3.2: VanillaVirtualAccount :: approveTokenForContract()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of USDT's `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a `using SafeERC20 for IERC20`. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `2efeb2e`.

3.2 Denial-Of-Service Against of WETHGateway

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: WETHGateway
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Gearbox, there is a convenient contract WETHGateway that provides automatic wrapping and unwrapping of ETH to add/remove liquidity and open/repay virtual accounts. In the following, we examine this logic behind this contract and report a denial-of-service issue against the virtual account creation.

The issue stems from the logic behind the virtual account creation. To elaborate, we show below the `openVirtualAccount()` routine that is responsible for the opening of a virtual account. We notice this routine supports a parameter i.e., `onBehalfOf`, which indicates the true owner of the created virtual account. However, in order to successfully create the virtual account, it requires `msg.sender` does not have a virtual account.

```

177     function openVirtualAccount(
178         uint256 amount,
179         address payable onBehalfOf,
180         uint256 leverageFactor,
181         uint256 referralCode
182     ) external override whenNotPaused nonReentrant {
183         // Checks that amount is in limits
184         require(
185             amount >= minAmount && amount <= maxAmount,
186             Errors.VAM_INCORRECT_AMOUNT
187         );
188
189         // Checks that user has no opened accounts
190         require(
191             !hasOpenedVirtualAccount(msg.sender),
192             Errors.VAM_YOU_HAVE_ALREADY_OPEN_VIRTUAL_ACCOUNT
193         );
194         ...
195     }

```

Listing 3.3: AbstractVirtualAccountManager::openVirtualAccount()

As a result, a malicious actor may intentionally create a virtual account by specifying `onBehalfOf` to be the WETHGateway portal. By doing so, WETHGateway may not be able to create any virtual account for others, hence a denial-of-service for legitimate users.

Recommendation Instead of requiring that the `msg.sender` cannot have a virtual account, the `openVirtualAccount()` routine should be revised to require `onBehalfOf` does not have a virtual account.

Status The issue has been fixed by this commit: `0e90f26`.

3.3 Overwritten Virtual Accounts Without Repayment

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `AbstractVirtualAccountManager`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Section 3.2, we have examined the `WETHGateway` contract and reported a denial-of-service issue. In this section, we further analyze the virtual account creation logic and report a different scenario that makes use of the same `onBehalfOf` support to avoid paying borrowed funds.

```

177     function openVirtualAccount(
178         uint256 amount,
179         address payable onBehalfOf,
180         uint256 leverageFactor,
181         uint256 referralCode
182     ) external override whenNotPaused nonReentrant {
183         // Checks that amount is in limits
184         require(
185             amount >= minAmount && amount <= maxAmount,
186             Errors.VAM_INCORRECT_AMOUNT
187         );
188
189         // Checks that user has no opened accounts
190         require(
191             !hasOpenedVirtualAccount(msg.sender),
192             Errors.VAM_YOU_HAVE_ALREADY_OPEN_VIRTUAL_ACCOUNT
193         );
194
195         // Checks that leverage factor is in limits
196         require(
197             leverageFactor > 0 && leverageFactor <= maxLeverageFactor,
198             Errors.VAM_INCORRECT_LEVERAGE_FACTOR
199         );
200
201         // borrowedAmount = amount * leverageFactor
202         uint256 borrowedAmount =
203             amount.mul(leverageFactor).div(Constants.LEVERAGE_DECIMALS);
204
205         // Get Reusable Virtual Account virtualAccount

```

```

206     address virtualAccount =
207         _accountFactory.takeVirtualAccount(address(this), onBehalfOf);

209     // Transfer pool tokens to new virtual account
210     IPoolService(poolService).lendVirtualAccount(
211         borrowedAmount,
212         virtualAccount
213     );

215     // Transfer borrower own fund to virtual account
216     IERC20(underlyingToken).safeTransferFrom(
217         msg.sender,
218         virtualAccount,
219         amount
220     );

222     // Set parameters for new virtual account
223     IVirtualAccount(virtualAccount).setGenericParameters(
224         borrowedAmount,
225         IPoolService(poolService).calcLinearCumulative_RAY()
226     );

228     // link virtual account address with borrower address
229     _virtualAccounts[onBehalfOf] = virtualAccount;

231     // emit new event
232     emit OpenVirtualAccount(
233         msg.sender,
234         onBehalfOf,
235         virtualAccount,
236         amount,
237         borrowedAmount,
238         referralCode
239     );
240 }

```

Listing 3.4: AbstractVirtualAccountManager::openVirtualAccount()

Specifically, we show above the full implementation of the `openVirtualAccount()` routine. Suppose a borrowing user *Malice* already opened a virtual *account_A* and is having a debt of 10K ETH on the account. The protocol accordingly maintains a mapping from *Alice* to *account_A*, i.e., `_virtualAccounts[Malice] = account_A` (line 229). However, *Malice* may use another fresh identity to call `openVirtualAccount()` (by providing *Malice* as `onBehalfOf`), resulting in the creation of another virtual *account_B*. This virtual account *account_B* does not have any debt. As a result, *Malice* is now associated with *account_B*, wiping out his previous debt in *account_A*.

Recommendation Similar to Section 3.2, there is a need to revise `openVirtualAccount()` to ensure `onBehalfOf` may not have a virtual account.

Status The issue has been fixed by this commit: [0e90f26](#).

3.4 Improved Logic Of allowToken()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VAMFilter
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Gearbox protocol has developed a VAMFilter contract that encapsulates the logic to vet supported tokens. This is necessary to ensure only legitimate, safe tokens may be listed.

To elaborate, we show below the core allowToken() routine that adds the given token to the list of allowed tokens. We note this routine is permissioned with the duringConfigOnly modifier. This modifier ensures that the list can only be modified when the configuration change is permitted. Moreover, tokens can only be added into the list of allowed tokens. It comes to our attention that currently it does not allow an allowed token to be removed. To accommodate convenient management and flexible adjustment, it is suggested to permit current allowed tokens to be adjusted as far as the same duringConfigOnly holds.

```

70  /**
71   * @dev Adds token to the list of allowed tokens
72   * @param token Address of allowed token
73   * @param liquidityThreshold
74   */
75  function allowToken(address token, uint256 liquidityThreshold)
76  public
77    override
78    duringConfigOnly
79    onlyOwner
80  {
81    require(token != address(0), Errors.ZERO_ADDRESS_IS_NOT_ALLOWED);
82    require(!_allowedTokensMap[token], Errors.VF_TOKEN_IS_ALREADY_ALLOWED);
83
84    _allowedTokensMap[token] = true;
85    tokensLiquidityThreshold[token] = liquidityThreshold;
86    allowedTokens.push(token);
87  }

```

Listing 3.5: VAMFilter::allowToken()

Recommendation Add necessary support to permit current allowed tokens to be modified when the duringConfigOnly modifier holds.

Status The issue has been fixed by this commit: 984ad73.

3.5 Potential DoS Against PoolService

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Low
- Target: PoolService
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Gearbox protocol is designed for decentralized margin trading and leveraged yield farming with a liquidity pool and anyone can become a liquidity provider by depositing funds into the pool. In the following, we examine the pool logic, especially when new liquidity is being added into the pool.

The liquidity-adding logic is implemented in the `addLiquidity()` routine. To elaborate, we show below its implementation. This routine has a rather straightforward logic: it firstly transfers in the new liquidity, then mints corresponding pool share to the liquidity provider, and finally updates the new borrow rate due to the new liquidity.

```

144     function addLiquidity(
145         uint256 amount,
146         address onBehalfOf,
147         uint256 referralCode
148     ) external override whenNotPaused nonReentrant {
149         IERC20(underlyingToken).safeTransferFrom(
150             msg.sender,
151             address(this),
152             amount
153         );
154
155         DieselToken(dieselToken).mint(onBehalfOf, _toDiesel(amount));
156
157         _expectedLiquidityLU = _expectedLiquidityLU.add(amount);
158         _updateBorrowRate();
159         emit AddLiquidity(msg.sender, onBehalfOf, amount, referralCode);
160     }

```

Listing 3.6: PoolService :: addLiquidity()

If we follow the execution flow, the new borrow rate is updated in the following `_updateBorrowRate()` routine. It comes to our attention that the new borrow APY requires the calculation of `availableLiquidity()` (line 351), which basically returns `IERC20(underlyingToken).balanceOf(address(this))` (line 217).

```

340     function _updateBorrowRate() internal {
341         // Update total _expectedLiquidityLU
342
343         _expectedLiquidityLU = expectedLiquidity();

```

```

345     // Update cumulativeIndex
346     _cumulativeIndex_RAY = calcLinearCumulative_RAY();

348     // update borrow APY
349     borrowAPY_RAY = _interestRateModel.calcBorrowRate(
350         _expectedLiquidityLU,
351         availableLiquidity()
352     );
353     _timestampLU = block.timestamp;
354 }

```

Listing 3.7: PoolService::_updateBorrowRate()

```

216     function availableLiquidity() public view override returns (uint256) {
217         return IERC20(underlyingToken).balanceOf(address(this));
218     }

```

Listing 3.8: PoolService::availableLiquidity ()

With that, a possible denial-of-service situation may be introduced. For simplicity, suppose a malicious actor is the first liquidity provider. The actor may choose to directly transfer assets into the pool without going through the normal `addLiquidity()` entry. As far as the transferred amount is larger than the `_expectedLiquidityLU` (line 350), the borrow rate calculation at line 349 will be reverted as there is an arithmetic underflow. As a result, new liquidity providers may not be able to add liquidity, hence causing a denial-of-service.

Recommendation Revise the borrow rate calculation to avoid the above denial-of-service issue.

Status The issue has been fixed by this commit: 984ad73.

3.6 Improved Sanity Checks For System Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Gearbox protocol is no exception. Specifically, if we examine the `VAMFilter` contract, it has defined a number of protocol-wide risk parameters, such as `tokensLiquidityThreshold`, `minAmount`, and `maxAmount`. In the following, we show an example routine that allow for their changes.

```

70  /**
71   * @dev Adds token to the list of allowed tokens
72   * @param token Address of allowed token
73   * @param liquidityThreshold
74   */
75  function allowToken(address token, uint256 liquidityThreshold)
76  public
77      override
78      duringConfigOnly
79      onlyOwner
80  {
81      require(token != address(0), Errors.ZERO_ADDRESS_IS_NOT_ALLOWED);
82      require(!_allowedTokensMap[token], Errors.VF_TOKEN_IS_ALREADY_ALLOWED);
83
84      _allowedTokensMap[token] = true;
85      tokensLiquidityThreshold[token] = liquidityThreshold;
86      allowedTokens.push(token);
87  }

```

Listing 3.9: VAMFilter::allowToken()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `minAmount` may impose a restrictive barrier to open a virtual account, hence affecting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: 984ad73.

3.7 Possible Sandwich/MEV Attacks For Reduced Conversion

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: AbstractVirtualAccountManager
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

As mentioned in Section 3.2, the Gearbox protocol opens a new virtual account for each participating user. We have examined the logic behind the virtual account creation. In the following, we further

examine the logic behind the virtual account termination.

To elaborate, we show below the `closeVirtualAccount()` function in the `AbstractVirtualAccountManager`. It basically converts all assets to the underlying one and then delegates the close operation to `_closeVirtualAccountImpl()` handler (line 269).

```

256     function closeVirtualAccount(address to, uint256 amountOutTolerance)
257     external
258     override
259     hasOpenedVirtualAccountOnly
260     whenNotPaused
261     nonReentrant
262     {
263         // Converts all assets to underlying one. _convertAllAssetsToUnderlying is
           virtual
264         // and should be implemented in derived classes
265         _convertAllAssetsToUnderlying(msg.sender, amountOutTolerance);
266
267         // Load virtual account details: amount & borrowed amount
268         uint256 remainingFunds =
269             _closeVirtualAccountImpl(
270                 Constants.OPERATION_CLOSURE,
271                 msg.sender,
272                 address(0),
273                 to
274             );
275
276         emit CloseVirtualAccount(msg.sender, to, remainingFunds);
277     }

```

Listing 3.10: `AbstractVirtualAccountManager::closeVirtualAccount()`

The conversion of all assets to the underlying one is implemented in inheriting contracts. Using the `TraderVirtualAccountManager` as an example, we show below the related conversion routine, i.e., `convertAllAssetsToUnderlying`.

```

214     function _convertAllAssetsToUnderlying(
215         address trader,
216         uint256 amountOutTolerance
217     ) internal override {
218         TraderVirtualAccount traderAccount =
219             TraderVirtualAccount(getVirtualAccountAddress(trader));
220
221         for (uint256 i = 0; i < _vamFilter.getAllowedTokensCount(); i++) {
222             (address tokenAddr, uint256 amount) =
223                 getVirtualAccountTokenById(trader, i);
224
225             if (tokenAddr != underlyingToken && amount > 0) {
226                 // Sell on vault
227                 address[] memory path = new address[](2);
228                 path[0] = tokenAddr;
229                 path[1] = underlyingToken;
230

```

```

231         _provideVirtualAccountAllowance(
232             trader ,
233             _defaultSwapContract ,
234             tokenAddr
235         );
236
237         uint256[] memory amountsOut =
238             IUniswapV2Router02(_defaultSwapContract).getAmountsOut(
239                 amount ,
240                 path
241             );
242
243         traderAccount.swapExactTokensForTokens(
244             _defaultSwapContract ,
245             amount ,
246             amountsOut[1].mul(amountOutTolerance).div(
247                 Constants.PERCENTAGE_FACTOR
248             ) ,
249             path ,
250             block.timestamp
251         );
252     }
253 }
254

```

Listing 3.11: TraderVirtualAccountManager::convertAllAssetsToUnderlying()

We notice all assets are routed to UniswapV2 in order to swap them to the underlying one. And the swap operation does not specify a valid restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller converted amount.

Note that this is a common issue 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 or the virtual account in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of UniswapV2. Nevertheless, 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 attack to better protect the interests of protocol users.

Status The issue has been fixed by this commit: 984ad73.

3.8 Improper Interest Collection In `lendVirtualAccount()`

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: PoolService
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.5, the Gearbox protocol is designed for decentralized margin trading and leveraged yield farming with a liquidity pool and anyone can become a liquidity provider by depositing funds into the pool. In the following, we examine another pool logic when funds are being borrowed to a farming user.

In the following, we show the `PoolService::lendVirtualAccount()` routine. As the name indicates, this routine is designed to lend funds to a virtual account. Since the lend operation affects the available liquidity, there is a need to timely accrue the interests from existing borrows, then account for the new borrow amount, and finally update the new borrow rate.

```

224  /**
225   * @dev Lends funds to virtual account manager
226   * and updates the pool parameters
227   *
228   * More: https://dev.gearbox.fi/developers/pool/abstractpoolservice#
229         lendvirtualaccount
230   *
231   * @param borrowedAmount Borrowed amount for virtual account
232   * @param virtualAccount Virtual account address
233   */
234  function lendVirtualAccount(uint256 borrowedAmount, address virtualAccount)
235      external
236      override
237      virtualAccountManagerOnly
238      whenNotPaused
239  {
240      // Increase total borrowed amount
241      totalBorrowed = totalBorrowed.add(borrowedAmount);
242
243      // Transfer funds to virtual account
244      IERC20(underlyingToken).safeTransfer(virtualAccount, borrowedAmount);
245
246      // Update borrow Rate
247      _updateBorrowRate();
248      emit Borrow(msg.sender, virtualAccount, borrowedAmount);

```

249

}

Listing 3.12: PoolService :: lendVirtualAccount()

To elaborate, we show above the full implementation of `lendVirtualAccount()`. It comes to our attention that the logic directly increases total borrowed amount (line 240) without firstly accruing the interest from existing borrows. With that, it may incorrectly over-charge existing borrowers for additional interest (as the total borrow amount is prematurely increased). As a solution, there is a need to split the `_updateBorrowRate()` into two parts: upon the entry, we need to properly collect interest and update index; and upon exit, we only need to update the new borrow rate.

Recommendation Properly revise the `lendVirtualAccount()` to collect due interest.

Status The issue has been fixed by this commit: [984ad73](#).

3.9 Asset Consistency Between VAMFilter And PoolService

- ID: PVE-009
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [\[4\]](#)
- CWE subcategory: CWE-1126 [\[1\]](#)

Description

In the Gearbox protocol, there is an implicit requirement between two core contracts, i.e., `VAMFilter` and its `PoolService`. In particular, both have an internal state to keep record of the current underlying token. Naturally, their underlying tokens need to be identical. Otherwise, the entire protocol may deviate the intended design and potentially cause fund loss.

To elaborate, we show below the constructors of these two contracts. Each constructor properly initializes current setup with required parameters. Note that both require the input of the underlying token. It is suggested to delegate the role of the common `addressProvider` that is designed to keep all protocol-wide configurations and parameters.

```

108     constructor(
109         address addressProvider ,
110         address _underlyingToken ,
111         address _dieselAddress ,
112         address interestRateModelAddress
113     ) PausableTrait(addressProvider) {
114         _addressProvider = AddressProvider(addressProvider);
115         _interestRateModel = IInterestRateModel(interestRateModelAddress);
116         underlyingToken = _underlyingToken;
117         dieselToken = _dieselAddress;

```



```

118     _treasuryAddress = _addressProvider.getTreasuryContract();
119
120     _cumulativeIndex_RAY = WadRayMath.RAY;
121     _timestampLU = block.timestamp;
122 }

```

Listing 3.13: PoolService::constructor()

```

58     constructor(address addressProvider, address underlyingTokenAddress) {
59         _priceOracle = IPriceOracle(
60             AddressProvider(addressProvider).getPriceOracle()
61         );
62
63         _underlyingTokenAddress = underlyingTokenAddress;
64         allowToken(
65             _underlyingTokenAddress,
66             Constants.UNDERLYING_TOKEN_LIQUIDATION_THRESHOLD
67         );
68     }

```

Listing 3.14: VAMFilter::constructor()

Similarly, in the `StableVirtualAccountManager` contract, there is also a need to ensure that the underlying token is one of those coins supported in the Curve pool.

Recommendation Ensure the consistency of the underlying asset between `VAMFilter` and `PoolService`.

Status The issue has been fixed by this commit: [6aae166](#).

4 | Conclusion

In this audit, we have analyzed the Gearbox design and implementation. The system presents a unique, robust offering as a decentralized non-custodial protocol that enables decentralized margin trading and leveraged yield farming. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.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>.