SMART CONTRACT AUDIT REPORT
for
VALUE DEFI

Prepared By: Shuxiao Wang

Hangzhou, China
November 28, 2020
Document Properties

<table>
<thead>
<tr>
<th>Client</th>
<th>Value DeFi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Smart Contract Audit Report</td>
</tr>
<tr>
<td>Target</td>
<td>Vaults</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Author</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Auditors</td>
<td>Xuxian Jiang, Huaguo Shi</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>Shuxiao Wang</td>
</tr>
<tr>
<td>Approved by</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Classification</td>
<td>Public</td>
</tr>
</tbody>
</table>

Version Info

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>November 28, 2020</td>
<td>Xuxian Jiang</td>
<td>Final Release</td>
</tr>
</tbody>
</table>

Contact

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Shuxiao Wang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>+86 173 6454 5338</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:contact@peckshield.com">contact@peckshield.com</a></td>
</tr>
</tbody>
</table>
Contents

1 Introduction 5
   1.1 About Value DeFi .......................... 5
   1.2 About PeckShield ......................... 6
   1.3 Methodology ............................... 6
   1.4 Disclaimer ................................. 8

2 Findings 10
   2.1 Summary ................................ 10
   2.2 Key Findings .............................. 11

3 Detailed Results 12
   3.1 Revisited Assumption on Trusted Governance And Operator ......... 12
   3.2 Suggested Adherence of Checks-Effects-Interactions ............... 13
   3.3 Incompatibility with Deflationary/Rebasing Tokens ................ 14
   3.4 Improved Sanity Checks For System Parameters .................. 16
   3.5 Unused Code Removal ........................ 17
   3.6 Possible Sandwich/MEV Attack For Reduced Returns ............. 18
   3.7 Precise Conversion of WETH-USDC For Liquidity Addition .......... 19
   3.8 Improved claim() in WETHMultiPoolStrategy ..................... 21
   3.9 Asset Consistency Check Between Bank And Strategy ............. 22

4 Conclusion 24

5 Appendix 25
   5.1 Basic Coding Bugs .......................... 25
      5.1.1 Constructor Mismatch ...................... 25
      5.1.2 Ownership Takeover ...................... 25
      5.1.3 Redundant Fallback Function ............ 25
      5.1.4 Overflows & Underflows ................. 25
      5.1.5 Reentrancy ................................ 26
5.1.6 Money-Giving Bug .................................................... 26
5.1.7 Blackhole ............................................................. 26
5.1.8 Unauthorized Self-Destruct ...................................... 26
5.1.9 Revert DoS ............................................................ 26
5.1.10 Unchecked External Call ....................................... 27
5.1.11 Gasless Send ....................................................... 27
5.1.12 Send Instead Of Transfer ....................................... 27
5.1.13 Costly Loop .......................................................... 27
5.1.14 (Unsafe) Use Of Untrusted Libraries ....................... 27
5.1.15 (Unsafe) Use Of Predictable Variables .................... 28
5.1.16 Transaction Ordering Dependence ......................... 28
5.1.17 Deprecated Uses .................................................. 28
5.2 Semantic Consistency Checks ................................. 28
5.3 Additional Recommendations .................................... 28
  5.3.1 Avoid Use of Variadic Byte Array .......................... 28
  5.3.2 Make Visibility Level Explicit ............................... 29
  5.3.3 Make Type Inference Explicit ............................... 29
  5.3.4 Adhere To Function Declaration Strictly .................. 29

References ........................................................................ 30
1 Introduction

Given the opportunity to review the design document and related smart contract source code of Vaults in the Value DeFi protocol, we in the report outline 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 Value DeFi

The Value DeFi protocol is a platform containing a suite of products that aims to bring fairness, value, and innovation to DeFi. It operates on four core tenets: 1) increase accessibility to yield farming, 2) provide on-chain voting for governance, 3) reward our stakeholders with flexible, optimized and profitable vault strategies, and 4) protect the community’s funds through the integration of an insurance treasury. The audited Vaults are yield aggregators that deploy your assets using multiple strategies to keep the returns as high as possible.

The basic information of the Vaults is as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issuer</td>
<td>Value DeFi</td>
</tr>
<tr>
<td>Website</td>
<td><a href="https://valuedefi.io">https://valuedefi.io</a></td>
</tr>
<tr>
<td>Type</td>
<td>Ethereum Smart Contract</td>
</tr>
<tr>
<td>Platform</td>
<td>Solidity</td>
</tr>
<tr>
<td>Audit Method</td>
<td>Whitebox</td>
</tr>
<tr>
<td>Latest Audit Report</td>
<td>November 28, 2020</td>
</tr>
</tbody>
</table>

In the following, we show the Git repository of reviewed files and the commit hash value
used in this audit. Note that the repository contains a number of contracts and sub-directories and this audit covers only the following ones: ValueVaultMaster, ValueVaultBank, ValueVaultV2, and WETHMultiPoolStrategy.

- https://github.com/valuedefi/vaults.git (dab536c)

### 1.2 About PeckShield

PeckShield Inc. [20] 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).

![Vulnerability Severity Classification Table]

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Impact</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Critical</td>
<td>High</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

### 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [15]:

- **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 List of Check Items

<table>
<thead>
<tr>
<th>Category</th>
<th>Check Item</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Coding Bugs</strong></td>
<td>Constructor Mismatch&lt;br&gt;Ownership Takeover&lt;br&gt;Redundant Fallback Function&lt;br&gt;Overflows &amp; Underflows&lt;br&gt;Reentrancy&lt;br&gt;Money-Giving Bug&lt;br&gt;Blackhole&lt;br&gt;Unauthorized Self-Destruct&lt;br&gt;Revert DoS&lt;br&gt;Unchecked External Call&lt;br&gt;Gasless Send&lt;br&gt;Send Instead Of Transfer&lt;br&gt;Costly Loop&lt;br&gt;(Unsafe) Use Of Untrusted Libraries&lt;br&gt;(Unsafe) Use Of Predictable Variables&lt;br&gt;Transaction Ordering Dependence&lt;br&gt;Deprecated Uses</td>
</tr>
<tr>
<td><strong>Semantic Consistency Checks</strong></td>
<td>Semantic Consistency Checks</td>
</tr>
<tr>
<td><strong>Advanced DeFi Scrutiny</strong></td>
<td>Business Logs Review&lt;br&gt;Functionality Checks&lt;br&gt;Authentication Management&lt;br&gt;Access Control &amp; Authorization&lt;br&gt;Oracle Security&lt;br&gt;Digital Asset Escrow&lt;br&gt;Kill-Switch Mechanism&lt;br&gt;Operation Trails &amp; Event Generation&lt;br&gt;ERC20 Idiosyncrasies Handling&lt;br&gt;Frontend-Contract Integration&lt;br&gt;Deployment Consistency&lt;br&gt;Holistic Risk Management</td>
</tr>
<tr>
<td><strong>Additional Recommendations</strong></td>
<td>Avoiding Use of Variadic Byte Array&lt;br&gt;Using Fixed Compiler Version&lt;br&gt;Making Visibility Level Explicit&lt;br&gt;Making Type Inference Explicit&lt;br&gt;Adhering To Function Declaration Strictly&lt;br&gt;Following Other Best Practices</td>
</tr>
</tbody>
</table>
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 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) [14], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>Weaknesses in this category are typically introduced during the configuration of the software.</td>
</tr>
<tr>
<td>Data Processing Issues</td>
<td>Weaknesses in this category are typically found in functionality that processes data.</td>
</tr>
<tr>
<td>Numeric Errors</td>
<td>Weaknesses in this category are related to improper calculation or conversion of numbers.</td>
</tr>
<tr>
<td>Security Features</td>
<td>Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)</td>
</tr>
<tr>
<td>Time and State</td>
<td>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.</td>
</tr>
<tr>
<td>Error Conditions, Return Values, Status Codes</td>
<td>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.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Weaknesses in this category are related to improper management of system resources.</td>
</tr>
<tr>
<td>Behavioral Issues</td>
<td>Weaknesses in this category are related to unexpected behaviors from code that an application uses.</td>
</tr>
<tr>
<td>Business Logics</td>
<td>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.</td>
</tr>
<tr>
<td>Initialization and Cleanup</td>
<td>Weaknesses in this category occur in behaviors that are used for initialization and breakdown.</td>
</tr>
<tr>
<td>Arguments and Parameters</td>
<td>Weaknesses in this category are related to improper use of arguments or parameters within function calls.</td>
</tr>
<tr>
<td>Expression Issues</td>
<td>Weaknesses in this category are related to incorrectly written expressions within code.</td>
</tr>
<tr>
<td>Coding Practices</td>
<td>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.</td>
</tr>
</tbody>
</table>
2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of Vaults. 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.

<table>
<thead>
<tr>
<th>Severity</th>
<th># of Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>0</td>
</tr>
<tr>
<td>High</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>3</td>
</tr>
<tr>
<td>Low</td>
<td>5</td>
</tr>
<tr>
<td>Informational</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>9</td>
</tr>
</tbody>
</table>

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 3 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Vaults

<table>
<thead>
<tr>
<th>ID</th>
<th>Severity</th>
<th>Title</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVE-001</td>
<td>Medium</td>
<td>Revisited Assumption on Trusted Governance And Operator</td>
<td>Security Features</td>
</tr>
<tr>
<td>PVE-002</td>
<td>Low</td>
<td>Suggested Adherence of Checks-Effects-Interactions</td>
<td>Time and State</td>
</tr>
<tr>
<td>PVE-003</td>
<td>Informational</td>
<td>Incompatibility With Deflationary/Rebasing Tokens</td>
<td>Business Logic</td>
</tr>
<tr>
<td>PVE-004</td>
<td>Low</td>
<td>Improved Sanity Checks For System Parameters</td>
<td>Coding Practices</td>
</tr>
<tr>
<td>PVE-005</td>
<td>Low</td>
<td>Unused Code Removal</td>
<td>Coding Practices</td>
</tr>
<tr>
<td>PVE-006</td>
<td>Medium</td>
<td>Possible Sandwich/MEV Attack For Reduced Returns</td>
<td>Time and State</td>
</tr>
<tr>
<td>PVE-007</td>
<td>Low</td>
<td>Precise Conversion of WETH-USDC For Liquidity Addition</td>
<td>Business Logic</td>
</tr>
<tr>
<td>PVE-008</td>
<td>Medium</td>
<td>Improved claim() in WETHMultiPool-Strategy</td>
<td>Business Logic</td>
</tr>
<tr>
<td>PVE-009</td>
<td>Low</td>
<td>Improved Asset Consistency Among Bank And Strategies</td>
<td>Coding Practices</td>
</tr>
</tbody>
</table>

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 Revisited Assumption on Trusted Governance And Operator

- **ID:** PVE-001
- **Severity:** Medium
- **Likelihood:** Medium
- **Impact:** Medium
- **Target:** Multiple Contracts
- **Category:** Security Features [9]
- **CWE subcategory:** CWE-287 [4]

**Description**

In Vaults, the governance/operator account plays a critical role in governing and regulating the system-wide operations (e.g., vault/strategy addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., vault, bank, and strategy.

![Diagram of privilege management in Vaults](image)

*Figure 3.1: The Privilege Management Chain in Vaults*

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we examine the current privilege management graph in the
Vaults protocol (Figure 3.1).

We emphasize that the privilege assignment among bank, vault, and strategy is properly administered. However, it could be worrisome if governance is not governed by a DAO-like structure.

We point out that a compromised governance account would allow the attacker to add a malicious strategy, which directly undermines the assumption of the entire protocol.

**Recommendation** Promptly transfer the governance privilege to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

### 3.2 Suggested Adherence of Checks-Effects-Interactions

- **ID:** PVE-002
- **Severity:** Low
- **Likelihood:** Low
- **Impact:** Low
- **Target:** Multiple Contracts
- **Category:** Time and State [12]
- **CWE subcategory:** CWE-663 [6]

**Description**

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [23] exploit, and the recent Uniswap/Lendf.Me hack [21].

We notice there are several occasions the checks-effects-interactions principle is violated. Using the ValueVaultBank as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 193) starts before effecting the update on internal states (lines 194 − 195), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching re-entrancy via the very same `emergencyWithdraw()` function.

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _poolId) public {
    uint256 amount = stakers[_poolId][msg.sender].stake;
    poolMap[_poolId].token.safeTransfer(address(msg.sender), amount);
```
Another similar violation can be found in the make_profit() routine within the same contract, the deposit() routine in WETHSodaPoolStrategy, and the emergencyWithdraw()/depositOnBehalf()/withdrawOnBehalf() routines in ValueMinorPool.

In the meantime, we should mention that the UniswapV2’s LP tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

**Recommendation** Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. The above emergencyWithdraw() can be revised as follows:

```solidity
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _poolId) public {
    uint256 amount = stakers[_poolId][msg.sender].stake;
    stakers[_poolId][msg.sender].stake = 0;
    global[_poolId].total_stake = global[_poolId].total_stake.sub(amount);
    poolMap[_poolId].token.safeTransfer(address(msg.sender), amount);
}
```

**3.3 Incompatibility with Deflationary/Rebasing Tokens**

- **ID:** PVE-003
- **Severity:** Informational
- **Likelihood:** N/A
- **Impact:** N/A
- **Target:** ValueVaultBank
- **Category:** Business Logics [11]
- **CWE subcategory:** CWE-841 [8]

**Description**

In Vaults, the ValueVaultBank contract is designed to be the main entry for interaction with farming users. In particular, one entry routine, i.e., deposit(), accepts user deposits of supported assets (e.g., DAI). Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the ValueVaultBank contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault’s internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```solidity
// Deposit tokens to Bank. If we have a strategy, then tokens will be moved there.
function deposit(uint256 _poolId, uint256 _amount, bool _farmMinorPool, address _referrer) public discountCHI {
```
However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer` or `transferFrom()`. (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary/rebasing tokens.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the `ValueVaultBank` contract. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

**Recommendation** If current codebase needs to support deflationary tokens, it is better to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.
3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ValueVaultMaster
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [3]

**Description**

In Vaults, there are a number of home-made strategy contracts to invest farmers’ assets (held in ValueVaultBank) and collect reward tokens. The distribution of reward tokens requires proper setup of certain fee parameters. The protocol also defines related risk parameters, e.g., gasFee and govVaultProfitShareFee. The first parameter defines the deduction from the gains for gas reimbursement while the second parameter specifies the deduction that is charged on the yields to the governance vault.

Our result shows the update logic on this fee parameter and other system-wide 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 a large fee parameter (say more than 100%) will revert every harvest() operation.

To elaborate, we show below its code snippet of two related routines, i.e., setGovVaultProfitShareFee() and setGasFee(). These two functions update the percentages for governance profit and gas reimbursement respectively. However, they can be improved to validate that the given govVaultProfitShareFee (or gasFee) falls in an appropriate range.

```solidity
function setGovVaultProfitShareFee(uint256 _govVaultProfitShareFee) public {
    require(msg.sender == governance, "!governance");
    govVaultProfitShareFee = _govVaultProfitShareFee;
}

function setGasFee(uint256 _gasFee) public {
    require(msg.sender == governance, "!governance");
    gasFee = _gasFee;
}

function setMinStakeTimeToClaimVaultReward(uint256 _minStakeTimeToClaimVaultReward) public {
    require(msg.sender == governance, "!governance");
    minStakeTimeToClaimVaultReward = _minStakeTimeToClaimVaultReward;
}
```

Listing 3.4: ValueVaultMaster.sol
**Recommendation**  Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. An example validation is shown below. Note the sanity checks can also be applied to other routines, such as `setMinStakeTimeToClaimVaultReward()`.

```solidity
function setGovVaultProfitShareFee(uint256 _govVaultProfitShareFee) public {
    require(msg.sender == governance, "!governance");
    require(_govVaultProfitShareFee < FEE_DENOMINATOR, "!governance")
    govVaultProfitShareFee = _govVaultProfitShareFee;
}

function setGasFee(uint256 _gasFee) public {
    require(msg.sender == governance, "!governance");
    require(_gasFee < FEE_DENOMINATOR, "!governance")
    gasFee = _gasFee;
}
```

Listing 3.5: ValueVaultMaster.sol

### 3.5 Unused Code Removal

- **ID:** PVE-005
- **Severity:** Informational
- **Likelihood:** N/A
- **Impact:** N/A
- **Target:** Multiple Contracts
- **Category:** Coding Practices [10]
- **CWE subcategory:** CWE-563 [5]

#### Description

Vaults make good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, Context, and Ownable, to facilitate its code implementation and organization. For example, the ValueVaultBank smart contract has so far imported at least four 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 closely the `ValueVaultV2` contract, there is a state variable that is defined, but not used anymore: `lockedAmount`. This variable is apparently left behind from a deprecated feature.

```solidity
contract ValueVaultV2 is IValueVault, ERC20 {
    using SafeMath for uint256;

    address public governance;

    mapping (address => uint256) public lockedAmount;

    IStrategyV2 public strategy;
}
```
In addition, there are a number of unused state variables and these unused variables can be removed as well. Examples include `totalBalance` that is defined in various strategies under the v2 sub-directory, `sodaVault` in `WETHSodaPoolStrategy`, and `discountCHI` in `ValueVaultBank`.

**Recommendation**  Consider the removal of the unused code and the unused constants.

### 3.6 Possible Sandwich/MEV Attack For Reduced Returns

- **ID:** PVE-006
- **Severity:** Medium
- **Likelihood:** Medium
- **Impact:** Medium
- **Target:** Strategies
- **Category:** Time and State [13]
- **CWE subcategory:** CWE-682 [7]

**Description**

In Vaults, a number of different strategy contracts have been designed and implemented to invest farmers’ assets (held in `ValueVaultBank`) and harvest growing yields. To elaborate, we show below the `harvest()` routine from the `Univ2ETHUSDCMultiPoolStrategy` strategy. By calling this routine, the strategy can collect any pending rewards and swap them to the designated WETH tokens for the next round of investment.

```solidity
/**
 * @dev See {IStrategyV2-harvest}.
 */

function harvest(uint256 _bankPoolId, uint256 _poolId) external override {
    address bank = valueVaultMaster.bank();
    address _vault = msg.sender;
    require(valueVaultMaster.isVault(_vault), "!vault"); // additional protection so we don’t burn the funds

    poolMap[_poolId].targetPool.getReward();
    IERC20 targetToken = poolMap[_poolId].targetToken;
    uint256 targetTokenBal = targetToken.balanceOf(address(this));

    if (targetTokenBal < poolMap[_poolId].minHarvestForTakeProfit) return;

    _swapTokens(address(targetToken), address(weth), targetTokenBal);
    uint256 wethBal = weth.balanceOf(address(this));
}
```

Listing 3.6: `ValueVaultV2.sol`
Listing 3.7: Univ2ETHUSDCMultiPoolStrategy.sol

We notice the collected yields are routed to UniswapV2 in order to swap them to WETH as rewards. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding. A similar issue also exists in the harvest() routine of other strategies.

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 strategy contract 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 farming users.

3.7 Precise Conversion of WETH-USDC For Liquidity Addition

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Univ2ETHUSDCMultiPoolStrategy
- CWE subcategory: CWE-841 [8]

Description

The Vaults protocol contains a strategy Univ2ETHUSDCMultiPoolStrategy that takes Uniswap LP tokens (e.g., ETHUSDC_UNIv2) and invests in specified reward pools. In the following, we show the code snippet of the harvest() routine.

The harvest() routine is used to collect possible yields and convert them into target tokens. Our analysis shows that the rewards (in targetToken - lines 272 – 275) are converted into WETHs as an intermediary token. The converted WETHs are used to pay gasFee and govVaultProfitShareFee and the remaining amount is further re-invested back for increased LP tokens (lines 298 – 299).
```solidity
function harvest(uint256 _bankPoolId, uint256 _poolId) external override {
    address bank = valueVaultMaster.bank();
    address _vault = msg.sender;
    require(valueVaultMaster.isVault(_vault), "!vault"); // additional protection so we don’t burn the funds

    poolMap[_poolId].targetPool.getReward();
    IERC20 targetToken = poolMap[_poolId].targetToken;
    uint256 targetTokenBal = targetToken.balanceOf(address(this));

    if (targetTokenBal < poolMap[_poolId].minHarvestForTakeProfit) return;

    _swapTokens(address(targetToken), address(weth), targetTokenBal);
    uint256 wethBal = weth.balanceOf(address(this));

    if (wethBal > 0) {
        uint256 _reserved = 0;
        uint256 _gasFee = 0;
        uint256 _govVaultProfitShareFee = 0;

        if (valueVaultMaster.gasFee() > 0) {
            _gasFee = wethBal.mul(valueVaultMaster.gasFee()).div(FEE_DENOMINATOR);
            _reserved = _reserved.add(_gasFee);
        }

        if (valueVaultMaster.govVaultProfitShareFee() > 0) {
            _govVaultProfitShareFee = wethBal.mul(valueVaultMaster.govVaultProfitShareFee()).div(FEE_DENOMINATOR);
            _reserved = _reserved.add(_govVaultProfitShareFee);
        }

        uint256 wethToBuyTokenA = wethBal.sub(_reserved).div(2); // we have TokenB (WETH) already, so use 1/2 bal to buy TokenA (USDC)
        ...
    }
}
```

Listing 3.8: Univ2ETHUSDCMultiPoolStrategy.sol

However, the current method in converting the remaining amount back to liquidity can be improved. The reason is that the current pool reserve may not be balanced in requiring the equally same value of WETH and USDT. A better approach is to make user of an external helper (deployed at 0x3AF045FD63Afc040aE9FD8C5b380d2DF2BB04cfc) to better maximize the liquidity return.

**Recommendation**  Optimize the allocation of rewarded assets for maximized liquidity return. An example implementation is located in 0x3AF045FD63Afc040aE9FD8C5b380d2DF2BB04cfc.
3.8 Improved claim() in WETHMultiPoolStrategy

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: WETHMultiPoolStrategy
- CWE subcategory: CWE-841 [8]

Description

The Vaults protocol contains another strategy WETHMultiPoolStrategy that takes the WETH assets and invests in three different reward pools, i.e., Golff.finance, ChickenSwap, Soda.finance. This strategy exports the functionality to claim current rewards. To elaborate, we show below the related code snippet of the claim() routine.

```solidity
/**
 * @dev See {IStrategy - claim}.
 */
function claim(address _vault) external override {
    require(valueVaultMaster.isVault(_vault), "not vault");
    require(poolPreferredIds.length > 0, "no pool");
    for (uint256 i = 0; i < poolPreferredIds.length; ++i) {
        uint256 _pid = poolPreferredIds[i];
        if (poolMap[_pid].vault == _vault) {
            _claim(_pid);
        }
    }
}

/**
 * @dev See {IStrategyV2 - claim}.
 */
function claim(uint256 _poolId) external override {
    require(poolMap[_poolId].vault == msg.sender, "sender not vault");
    _claim(_poolId);
}

function _claim(uint256 _poolId) internal {
    if (poolMap[_poolId].poolType == 0) {
        ISstakingRewards(poolMap[_poolId].targetPool).getReward();
    } else if (poolMap[_poolId].poolType == 1) {
        ISushiPool(poolMap[_poolId].targetPool).deposit(poolMap[_poolId].targetPoolId, 0);
    } else {
        ISodaPool(poolMap[_poolId].targetPool).claim(poolMap[_poolId].targetPoolId);
    }
}
```

Listing 3.9: WETHMultiPoolStrategy.sol
Two claim() routines are publicly accessible while they both rely on an internal helpful routine, _claim(). It comes to our attention that the first claim() routine (lines 287–299) essentially imposes no restriction on the caller, while the second claim() routine (lines 301–307) requires the caller to be a known vault. Apparently, there is an inconsistency in enforcing who will be allowed to invoke the claim() functionality. Note that the deposit() functionality in the same contract shares a similar issue.

**Recommendation**  
Be consistent in applying sanity checks on both claim() and deposit() functionalities.

### 3.9 Asset Consistency Check Between Bank And Strategy

**ID:** PVE-009  
**Severity:** Low  
**Likelihood:** Low  
**Impact:** Low  
**Target:** Multiple Contracts  
**Category:** Coding Practices [10]  
**CWE subcategory:** CWE-1099 [2]

**Description**

In the Vaults protocol, there is a one-to-one mapping between a vault and its strategy. To properly link a vault with its strategy, it is natural for the two to operate on the same underlying asset. For example, ValueVaultV2 allows for WETH-based deposits and withdraws. The associated strategy, i.e., a WETHMultiPoolStrategy-based instance, naturally has WETH as the underlying asset. If these two have different underlying assets, the link should not be successful.

```solidity
// Deposit tokens to Bank. If we have a strategy, then tokens will be moved there.
function deposit(uint256 _poolId, uint256 _amount, bool _farmMinorPool, address _referrer) public discountCHI {
    PoolInfo storage pool = poolMap[_poolId];
    require(now >= pool.startTime, "deposit: after startTime");
    require(_amount > 0, "!_amount");
    require(address(pool.vault) != address(0), "pool.vault = 0");
    require(pool.individualCap == 0, stakers[_poolId][msg.sender].stake.add(_amount) <= pool.individualCap, "Exceed pool.individualCap");
    require(pool.totalCap == 0, global[_poolId].total_stake.add(_amount) <= pool.totalCap, "Exceed pool.totalCap");

    pool.token.safeTransferFrom(msg.sender, address(pool.vault), _amount);
    pool.vault.mintByBank(pool.token, msg.sender, _amount);
    if (!_farmMinorPool && address(vaultMaster) != address(0)) {
        address minorPool = vaultMaster.minorPool();
        if (minorPool != address(0)) {
            IValueMinorPool(minorPool).depositOnBehalf(msg.sender, pool.minorPoolId, pool.vault.balanceOf(msg.sender), _referrer);
        }
    }
}
```
For elaboration, we show above the deposit() routine of the ValueVaultBank contract. It is important to ensure the deposited asset is consistent with the expected asset in receiving strategies for investment. In other words, it is suggested to impose another requirement, i.e., require(pool.vault.strategy.getLpToken() == pool.token). By doing so, we can ensure users may not mistakenly sent wrong assets for investment.

**Recommendation**  Ensure the consistency of the underlying asset between the bank and its associated strategies.
4 | Conclusion

In this audit, we have analyzed the design and implementation of Vaults in the Value DeFi protocol. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. Inspired from YFI, the Value DeFi protocol has been equipped with additional home-made strategies that work with different yield-generating pools. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that 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.
5  Appendix

5.1  Basic Coding Bugs

5.1.1  Constructor Mismatch

- **Description:** Whether the contract name and its constructor are not identical to each other.
- **Result:** Not found
- **Severity:** Critical

5.1.2  Ownership Takeover

- **Description:** Whether the set owner function is not protected.
- **Result:** Not found
- **Severity:** Critical

5.1.3  Redundant Fallback Function

- **Description:** Whether the contract has a redundant fallback function.
- **Result:** Not found
- **Severity:** Critical

5.1.4  Overflows & Underflows

- **Description:** Whether the contract has general overflow or underflow vulnerabilities [16, 17, 18, 19, 22].
- **Result:** Not found
- **Severity:** Critical
5.1.5 Reentrancy
• **Description**: Reentrancy [24] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
• **Result**: Not found
• **Severity**: Critical

5.1.6 Money-Giving Bug
• **Description**: Whether the contract returns funds to an arbitrary address.
• **Result**: Not found
• **Severity**: High

5.1.7 Blackhole
• **Description**: Whether the contract locks ETH indefinitely: merely in without out.
• **Result**: Not found
• **Severity**: High

5.1.8 Unauthorized Self-Destruct
• **Description**: Whether the contract can be killed by any arbitrary address.
• **Result**: Not found
• **Severity**: Medium

5.1.9 Revert DoS
• **Description**: Whether the contract is vulnerable to DoS attack because of unexpected revert.
• **Result**: Not found
• **Severity**: Medium
5.1.10 **Unchecked External Call**

- **Description:** Whether the contract has any external call without checking the return value.
- **Result:** Not found
- **Severity:** Medium

5.1.11 **Gasless Send**

- **Description:** Whether the contract is vulnerable to gasless send.
- **Result:** Not found
- **Severity:** Medium

5.1.12 **Send Instead Of Transfer**

- **Description:** Whether the contract uses send instead of transfer.
- **Result:** Not found
- **Severity:** Medium

5.1.13 **Costly Loop**

- **Description:** Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- **Result:** Not found
- **Severity:** Medium

5.1.14 **(Unsafe) Use Of Untrusted Libraries**

- **Description:** Whether the contract use any suspicious libraries.
- **Result:** Not found
- **Severity:** Medium
5.1.15 (Unsafe) Use Of Predictable Variables

- **Description:** Whether the contract contains any randomness variable, but its value can be predicated.
- **Result:** Not found
- **Severity:** Medium

5.1.16 Transaction Ordering Dependence

- **Description:** Whether the final state of the contract depends on the order of the transactions.
- **Result:** Not found
- **Severity:** Medium

5.1.17 Deprecated Uses

- **Description:** Whether the contract use the deprecated `tx.origin` to perform the authorization.
- **Result:** Not found
- **Severity:** Medium

5.2 Semantic Consistency Checks

- **Description:** Whether the semantic of the white paper is different from the implementation of the contract.
- **Result:** Not found
- **Severity:** Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- **Description:** Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- **Result:** Not found
- **Severity:** Low
5.3.2  Make Visibility Level Explicit

- **Description:** Assign explicit visibility specifiers for functions and state variables.
- **Result:** Not found
- **Severity:** Low

5.3.3  Make Type Inference Explicit

- **Description:** Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- **Result:** Not found
- **Severity:** Low

5.3.4  Adhere To Function Declaration Strictly

- **Description:** Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- **Result:** Not found
- **Severity:** Low
References


