SMART CONTRACT AUDIT REPORT

for

VALUE SET FINANCE

Prepared By: Shuxiao Wang

PeckShield

February 20, 2021

Document Properties

Client	Value Set Finance
Title	Smart Contract Audit Report
Target	Value Set Dollar (VSD)
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 20, 2021	Xuxian Jiang	Final Release
1.0-rc	February 19, 2021	Xuxian Jiang	Release Candidate #1
0.3	February 10, 2021	Xuxian Jiang	Additional Findings #2
0.2	February 5, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 2, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

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

1.1 About Value Set Dollar

Value Set Dollar (VSD) is a value-backed, self-stabilizing, and decentralized stablecoin with a basket of collateral backing and algorithmic incentive mechanism. Different from existing stablecoin solutions, it features an algorithmic and partially collateral approach to maintain price stability around a 1 USDC/DAI target, and relies on a tuned incentive mechanism (e.g., coupon premium and coupon extension). The VSD supply allows for dynamic expansion and contraction based on current market demands. The protocol merges the native support of DAO and multiple LP pools from the original ESD design. It is also partially backed by a basket of collaterals (USDC/DAI/USDT/ETH) and partially stabilized algorithmically.

The basic information of the VSD protocol is as follows:

Table 1.1: Basic Information of The VSD Protocol

Item	Description
Issuer	Value Set Finance
Website	https://www.valueset.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 20, 2021

In the following, we show the reviewed file and the `md5` checksum value used in this audit.

- audit.zip (md5: 5ca91271c67440459c5e68d2a475f760)

And this is the `md5` checksum value of the compressed file after all fixes for the issues found in the audit have been checked in:

- audit.zip (md5: 87c5434dff98dc83c53dd662278d970d)

1.2 About PeckShield

PeckShield Inc. [15] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the 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) [13], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.3: The Full List of Check Items

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

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

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

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	5	■■■■■
Informational	1	■
Total	8	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, and 5 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Value Set Dollar (VSD) Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Fee Consideration For Buy/Sell Amount Calculation	Business Logic	Confirmed
PVE-003	Informational	Redundant/Unused Code Removal	Time and State	Fixed
PVE-004	Medium	Sandwiched Advance With Influenced Reward/Debt Allocation	Time and State	Mitigated
PVE-005	Low	Improved Logic Of Setters::_sellAndDepositCollateral()	Business Logic	Confirmed
PVE-006	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-007	Low	Lack Of Sanity Checks For System/Functional Parameters	Coding Practices	Confirmed
PVE-008	Low	Race Condition Between approveCoupons() And transferCoupons()	Time and State	Confirmed

Besides recommending specific countermeasures to mitigate these issues, based on the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.8.0` instead of specifying a range, e.g., `pragma solidity ^0.8.0`.

In addition, 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 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Collateral`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [3]

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 `transfer()` routine and 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.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
```

```
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function transfer(address recipient, uint256 amount)external returns (bool)`. 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()`.

In the following, we show the `redeem()` routine in the `Collateral` contract. If the `USDT` token is supported in the protocol's `collateralAssetList`, the unsafe version of `IERC20(addr).transfer(msg.sender, actual.mul(IERC20(addr).balanceOf(address(this))).div(dollarTotalSupply)` (lines $51-54$) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
31    function redeem(uint256 value) external nonReentrant {
32        uint256 actual = value;
33        uint256 debt = totalDebt();
34        if (debt > value) {
35            // if there is debt, redeem at no cost
36            debt = value;
37        } else {
38            // redeem with cost
39            actual = value.sub((10000 - Constants.getRedemptionRate()).mul(value.sub(
                  debt)).div(10000));
40            uint256 fundReward = value.sub(actual);
41            uint256 devReward = fundReward.mul(Constants.getFundDevPct()).div(100);
42            uint256 treasuryReward = fundReward.sub(devReward);
43            dollar().transferFrom(msg.sender, Constants.getDevAddress(), devReward);
44            dollar().transferFrom(msg.sender, Constants.getTreasuryAddress(),
                  treasuryReward);
45        }

47        uint256 len = _state.collateralAssetList.length;
48        uint256 dollarTotalSupply = dollar().totalSupply();
49        for (uint256 i = 0; i < len; i++) {
50            address addr = _state.collateralAssetList[i];
51            IERC20(addr).transfer(
52                msg.sender,
53                actual.mul(IERC20(addr).balanceOf(address(this))).div(dollarTotalSupply)
```

```
            );
        }

        burnFromAccountForDebt(msg.sender, actual, debt);
    }
```

Listing 3.2: CoverFeeReceiver::buyBack()

Note a always-reverted `redeem()` operation essentially disables the collateral support in `VSD`, which is the reason why we rate this issue as high-severity.

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

Status The issue has been fixed by following the recommendation.

3.2 Fee Consideration For Buy/Sell Amount Calculation

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Getters`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `VSD` protocol implements a unique expansion and contraction mechanism in order to meet the target collateral ratio (even the prices of underlying collateral assets can be volatile) and maximize the benefits of active participants including bonded liquidity providers and value set share (VSS) holders. During the expansion epoch, additional `VSDs` are minted to sell in order to bring the price back to the normal range. During the contraction epoch, new `coupons` are allowed for purchase to reduce the total supply of `VSD`. The amount of `VSD` minted during expansion or the amount of `coupons` for purchase are facilitated by the following two routines, i.e., `_getSellAndReturnAmount()` and `_getBuyAmount()`.

To elaborate, we show below these two routines. As the names indicate, the first routine handles the expansion and the second one handles the contraction, with the same goal of adjusting the price back to its normal range ($[0.95, 1.05]$).

```
    function _getSellAndReturnAmount(
        uint256 price,
        uint256 targetPrice,
        uint256 reserve
    ) internal pure returns (uint256 sellAmount, uint256 returnAmount) {
        // price in resolution 1e18
```

```
        sellAmount = 0;
        returnAmount = 0;

        uint256 rootPoT = Babylonian.sqrt(price.mul(1e36).div(targetPrice));
        if (rootPoT > 1e18) { // res error
            sellAmount = (rootPoT - 1e18).mul(reserve).div(1e18);
        }

        uint256 rootPT = Babylonian.sqrt(price.mul(targetPrice));
        if (price > rootPT) { // res error
            returnAmount = (price - rootPT).mul(reserve).div(1e18);
        }
        if (sellAmount > returnAmount) { // res error
            sellAmount = returnAmount;
        }
    }

    function _getBuyAmount(uint256 price, uint256 targetPrice, uint256 reserve) internal
         pure returns (uint256 shouldBuy) {
        shouldBuy = 0;

        uint256 root = Babylonian.sqrt(price.mul(1e36).div(targetPrice));
        if (root < 1e18) { // res error
            shouldBuy = (1e18 - root).mul(reserve).div(1e18);
        }
    }
```

Listing 3.3: Getters::_getSellAndReturnAmount() and Getters::_getBuyAmount()

It comes to our attention that the above routines does not properly take the swap fee (in `UniswapV2`) into consideration. Note that each swap operation in `UniswapV2` will be charged by 0.3% fee after the trade. With that, current routines may not lead to the situation of resulting in an expected target price. Fortunately, from the perspective of target price range, the swap fee is non-essential and may not significantly undermine the purpose of the buy/sell amount calculation.

Recommendation Properly take the swap fee into consideration when computing the token amount to buy or sell.

Status The issue has been confirmed. However, considering that the fee inclusion will likely increase the gas cost for the amount calculation and the algorithmic approach to adjust the `VSD` supply has the flexible target range of $[0.95, 1.05]$, the team decides to leave it as is.

3.3 Redundant/Unused Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1099 [2]

Description

Value Set Dollar (VSD) makes use of a number of reference libraries and contracts, such as `SafeMath`, `ERC20`, and `Uniswap`, to facilitate the protocol implementation and organization. For instance, the `Implementation` smart contract interacts with at least five different 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 `Implementation::advance()` routine, it has a modifier `nonReentrant` to prevent it from being reentered. However, it also enforces the caller to be an externally-owned account (or EOA), which makes `nonReentrant` redundant. Therefore, this modifier in this specific routine can be safely removed.

```
contract Implementation is State, Bonding, Market, Regulator, Govern, Collateral {
    using SafeMath for uint256;

    event Advance(uint256 indexed epoch, uint256 block, uint256 timestamp);
    event Incentivization(address indexed account, uint256 amount);

    function initialize() initializer public {
    }

    function advance() nonReentrant external {
        require(msg.sender == tx.origin, "Must from user");
        incentivize(msg.sender, Constants.getAdvanceIncentive());

        Bonding.step();
        Regulator.step();
        Market.step();

        emit Advance(epoch(), block.number, block.timestamp);
    }

  ...
}
```

Listing 3.4: Implementation::advance()

In the same vein, we also observe another state, e.g., `COUPON_SUPPLY_CHANGE_LIMIT`, in `Constants` is not used either. The associated `getter`, i.e., `getCouponSupplyChangeLimit()` from the same contract can also be removed. For maintenance, their removals are recommended.

Recommendation Remove unnecessary imports of reference contracts and remove unused code.

Status The issue has been fixed by following the recommendation.

3.4 Sandwiched Advance With Influenced Reward/Debt Allocation

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Regulator`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

As mentioned in Section 3.2, the `VSD` protocol implements a unique expansion and contraction mechanism in order to meet the target collateral ratio (even the prices of underlying collateral assets can be volatile) and maximize the benefits of active participants including bonded liquidity providers and value set share (VSS) holders. Whether an epoch undergoes expansion and contraction is determined by current market price measured from specified pools (via `Oracle`).

To elaborate, we show below the `step()` routine from `Regulator`. This routine measures and determines current `VSD` price, and grows/shrinks the total supply if the price is above/below the specified upper/below threshold.

```
    function step() internal {
        Decimal.D256 memory price = oracleCapture();

        uint256 allReserve = _updateReserve();

        if (price.greaterThan(Decimal.D256({value: getSupplyIncreasePriceThreshold()})))
            {
            growSupply(price, allReserve);
            return;
        }

        if (price.lessThan(Decimal.D256({value: getSupplyDecreasePriceThreshold()}))) {
            shrinkSupply(price, allReserve);
            return;
        }
```

```
        emit SupplyNeutral(epoch());
    }
```

Listing 3.5: Regulator::step()

Specifically, if we focus on the expansion-handling logic, the helper routine `growSupply()` mints additional `VSD` tokens and sells them on specified `UniswapV2` pairs (e.g., `USDC/VSD`, `DAI/VSD`, and `USDT/VSD`).

```
    function growSupply(Decimal.D256 memory price, uint256 allReserve) private {
        uint256 lessDebt = resetDebt(Decimal.zero());

        (uint256 sellAmount, uint256 returnAmount) = _getSellAndReturnAmount(
            price.value,
            getSupplyIncreasePriceTarget(),
            allReserve
        );
        _sellAndDepositCollateral(sellAmount, allReserve);
        uint256 mintAmount = returnAmount.mul(10000).div(getCollateralRatio());
        (uint256 newRedeemable, uint256 newSupply, uint256 newReward) = increaseSupply(
            mintAmount.sub(sellAmount));
        emit SupplyIncrease(epoch(), price.value, sellAmount, newRedeemable, lessDebt,
            newSupply, newReward);
    }
```

Listing 3.6: Regulator::growSupply()

We notice the sell-off of minted `VSD` tokens is performed by sending the tokens to `UniswapV2` in order to swap one token to another. 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 return of collateral. Fortunately, this `step()` is restricted in a way that only EOA account is qualified to invoke, which significantly reduces the risks from possible flashloans. However, it should be emphasized that this does not eliminate this risk as powerful miners may still be able to launch sandwich-related attacks. A similar issue also exists in the `vote()` routine for proposing a new candidate with manipulated `bondedVotes` (by influencing the `balanceOfBondedDollar()` outcome).

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. 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.

Depending on the expansion/contraction, this issue may have implication to influence reward allocation and coupon allowance. The very same issue is also possible to bypass the proposal qualification restriction in governance.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of farming users.

Status The issue has been confirmed and largely mitigated by enforcing the EOA validation.

3.5 Improved Logic Of Setters::_sellAndDepositCollateral()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Setters
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The expansion and contraction mechanism (Section 3.3) is the cornerstone of the the VSD protocol. In this section, we further examine the business logic behind the expansion mechanism.

In particular, when it is determined to expand during an epoch, the protocol triggers the call to growSupply(), which takes two arguments. The first one is the current market price and the second one is current sum of VSD reserves of pooled pairs. With them, the protocol can compute the amount of new VSD tokens to mint to stabilize the VSD price.

```
    function growSupply(Decimal.D256 memory price, uint256 allReserve) private {
        uint256 lessDebt = resetDebt(Decimal.zero());

        (uint256 sellAmount, uint256 returnAmount) = _getSellAndReturnAmount(
            price.value,
            getSupplyIncreasePriceTarget(),
            allReserve
        );
        _sellAndDepositCollateral(sellAmount, allReserve);
        uint256 mintAmount = returnAmount.mul(10000).div(getCollateralRatio());
        (uint256 newRedeemable, uint256 newSupply, uint256 newReward) = increaseSupply(
            mintAmount.sub(sellAmount));
        emit SupplyIncrease(epoch(), price.value, sellAmount, newRedeemable, lessDebt,
            newSupply, newReward);
    }
```

Listing 3.7: Regulator::growSupply()

To elaborate, we show above the growSupply() routine and below the internal handler _sellAndDepositCollateral(). The bulk of expansion efforts is performed in the internal handler. It comes to our attention that the condition `if (reserveA == 0 || sellAmount == 0)` used in the internal `if-branch` (line 262) can be simplified as `if (reserveA == 0)`. The reason is that the check of `sellAmount == 0` is redundant. Also, the update to the `actualSold` state (line 260) can be better performed after the `if-branch` (line 262).

```
240     function _sellAndDepositCollateral(uint256 totalSellAmount, uint256 allReserve)
            internal {
241         if (totalSellAmount == 0  allReserve == 0) {
242             return;
243         }
244 
245         dollar().mint(address(this), totalSellAmount);
246         uint256 len = _state.poolList.length;
247         uint256 actualSold = 0;
248         // Sell to pools according to their reserves
249         for (uint256 i = 0; i < len; i++) {
250             address pool = _state.poolList[i];
251             address token0 = IUniswapV2Pair(pool).token0();
252             (uint256 reserve0, uint256 reserve1, ) = IUniswapV2Pair(pool).getReserves();
253 
254             uint256 reserveA = token0 == address(dollar()) ? reserve0 : reserve1;
255             uint256 reserveB = token0 == address(dollar()) ? reserve1 : reserve0;
256 
257             uint256 sellAmount = totalSellAmount
258                 .mul(reserveA)
259                 .div(allReserve);
260             actualSold = actualSold.add(sellAmount);
261 
262             if (reserveA == 0  sellAmount == 0) {
263                 // The pool is not ready yet or insufficient lp in pool.
264                 continue;
265             }
266 
267             uint256 assetAmount = UniswapV2Library.getAmountOut(
268                 sellAmount,
269                 reserveA,
270                 reserveB
271             );
272 
273             dollar().transfer(pool, sellAmount);
274 
275             // Non-Reentrancy?
276             IUniswapV2Pair(pool).swap(
277                 token0 == address(dollar()) ? 0 : assetAmount,
278                 token0 == address(dollar()) ? assetAmount : 0,
279                 address(this),
280                 new bytes(0)
281             );
282         }
283 
284         // Make sure we don't sell extra
285         assert(actualSold <= totalSellAmount);
```

```
    }
```

Listing 3.8: setters :: _sellAndDepositCollateral ()

Recommendation Simplify the `_sellAndDepositCollateral()` logic by removing unnecessary checks.

Status The issue has been confirmed.

3.6 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bonding`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `Bonding` as an example, the `provide()` 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 104) starts before effecting the update on internal states (lines 105−108), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `provide()` function.

```
99      function provide(address pool, address token, address another, uint256 amount)
            external {
100         preClaimDollar(pool);
101
102         unfreeze(pool, msg.sender);
103
104         uint256 bondedLP = _addLiquidity(pool, token, another, amount);
```

```
        incrementBalanceOfBonded(pool, msg.sender, bondedLP);

        emit Bond(pool, msg.sender, epoch().add(1), bondedLP);
        postClaimDollar(pool);
    }
```

Listing 3.9: Bonding::provide()

In the meantime, we should mention that the supported tokens in the protocol do 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.

Status The issue has been fixed by following the recommendation.

3.7 Lack Of Sanity Checks For System/Functional Parameters

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1099 [2]

Description

As mentioned in Section 3.1, DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Value Set Dollar (VSD) protocol is no exception. Specifically, if we examine the `Constants` contract, it has defined a number of constant system-wide risk parameters. Being constant, various risk parameters will have to rely on the governance to upgrade the protocol logic for their updates.

In the following, we show a specific routine `provide()` that is used to provide bonded liquidity.

```
    function provide(address pool, address token, address another, uint256 amount)
        external {
        preClaimDollar(pool);

        unfreeze(pool, msg.sender);

        uint256 bondedLP = _addLiquidity(pool, token, another, amount);
        incrementBalanceOfBonded(pool, msg.sender, bondedLP);

        emit Bond(pool, msg.sender, epoch().add(1), bondedLP);
        postClaimDollar(pool);
```

```
    }
```

Listing 3.10: Bonding::provide()

Note that this routine can be improved by applying a more rigorous validity check. Specifically, it has an argument `another`, which is unchecked and may lead to an undesirable consequence. For example, an user intends to add bonded liquidity by calling `provide()`, but however the user mistakenly provides an incorrect argument of `another`. If `another` does not belong to the second token of the pooled pair, it will result in loss of user funds.

Recommendation Validate the given `another` argument before adding the bonded liquidity.

Status The issue has been confirmed.

3.8 Race Condition Between approveCoupons() And transferCoupons()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Market`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [4]

Description

In the `VSD` protocol, there is a `Market` contract that implements the `coupon` support. Specifically, when the `VSD` price is less than 1.0, the protocol will move to the contraction stage and issue debt to attract users to reduce the `VSD` supply. The user can buy the debt as a coupon at a discounted price, and the coupon holder will have the right to redeem the coupon for future expansion reward.

The `coupon` support comes with a few handy routines. Two of them are `approveCoupons()` and `transferCoupons()`. The former allows the setting of a spender up to a specified allowance while the latter transfers a certain amount of `coupons` to another user. To elaborate, we show below related code snippet of these two routines.

```
    function approveCoupons(address spender, uint256 amount) external {
        Require.that(
            spender != address(0),
            FILE,
            "Coupon approve to 0x0"
        );

        updateAllowanceCoupons(msg.sender, spender, amount);
```

```
        emit CouponApproval(msg.sender, spender, amount);
    }

    function transferCoupons(address sender, address recipient, uint256 epoch, uint256 amount) external {
        Require.that(
            sender != address(0),
            FILE,
            "Coupon transfer from 0x0"
        );
        Require.that(
            recipient != address(0),
            FILE,
            "Coupon transfer to 0x0"
        );

        decrementBalanceOfCoupons(sender, epoch, amount);
        incrementBalanceOfCoupons(recipient, epoch, amount);

        if (msg.sender != sender && allowanceCoupons(sender, msg.sender) != uint256(-1)) {
            decrementAllowanceCoupons(sender, msg.sender, amount);
        }

        emit CouponTransfer(sender, recipient, epoch, amount);
    }
```

Listing 3.11: Market::approveCoupons() and Market::transferCoupons()

This pair of routines resembles the ERC20-specified `approve()` / `transferFrom()` pair and shares a similar known race condition issue [1]. Specifically, when a user intends to reduce the `couponAllowances` amount previously approved from, say, 10 `coupons` to 1 `coupon`. The user may race to transfer up to the previously approved `couponAllowances` (the 10 `coupons`) and then additionally transfer the new amount just approved (1 `coupon`). This breaks the user's intention of restricting the allowance to the new amount, **not** the sum of old amount and new amount.

In order to properly approve the `couponAllowances`, there also exists a known workaround: users can utilize the `increaseAllowanceCoupons()` and `decreaseAllowanceCoupons()` functions.

Recommendation Add the suggested workaround functions `increaseAllowanceCoupons()` and `decreaseAllowanceCoupons()`. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

Status This issue has been confirmed. Like in the `approval()`/`transferFrom()` pattern, there is no easy fix. The team plans to make sure builders and users are aware of this limitation.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Value Set Dollar` (`VSD`), which is value-backed, self-stabilizing, and decentralized stablecoin with a basket of collateral backing and algorithmic incentive mechanism. Based on the original solid design and implementation of `ESD`, `VSD` makes a number of innovations in allowing for collateral backing, multiple LP pools, and coupon extension. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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.

References

[1] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[17] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.