SMART CONTRACT AUDIT REPORT

for

VELO LABS TECHNOLOGY LTD.

Prepared By: Shuxiao Wang

Hangzhou, China

Aug. 15, 2020

Document Properties

Client	Velo Labs Technology Ltd.
Title	Smart Contract Audit Report
Target	DRSv2
Version	1.0-rc1
Author	Jeff Liu
Auditors	Chiachih Wu, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	Aug. 15, 2020	Jeff Liu	Release Candidate #1
0.2	Aug. 11, 2020	Chiachih Wu	Status Update
0.1	Aug. 10, 2020	Chiachih Wu	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 **DRSv2** smart contract source code, 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About DRSv2

The Velo Protocol is a blockchain financial protocol enabling digital credit issuance and borderless asset transfers for businesses using a smart contract system, and it enables Trusted Partners to issue digital credits via a smart contract layer. The Velo Protocol comprises a Digital Credit Issuance System and a Digital Reserve System, which are two sets of smart contracts that issue Velo tokens and manage the Velo Reserve Pool. As the name indicates, the smart contract being audited here, DRSv2, is part of the Digital Reserve System.

The basic information of DRSv2 is as follows:

Table 1.1: Basic Information of DRSv2

Item	Description
Issuer	Velo Labs Technology Ltd.
Website	https://velo.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Aug. 15, 2020

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

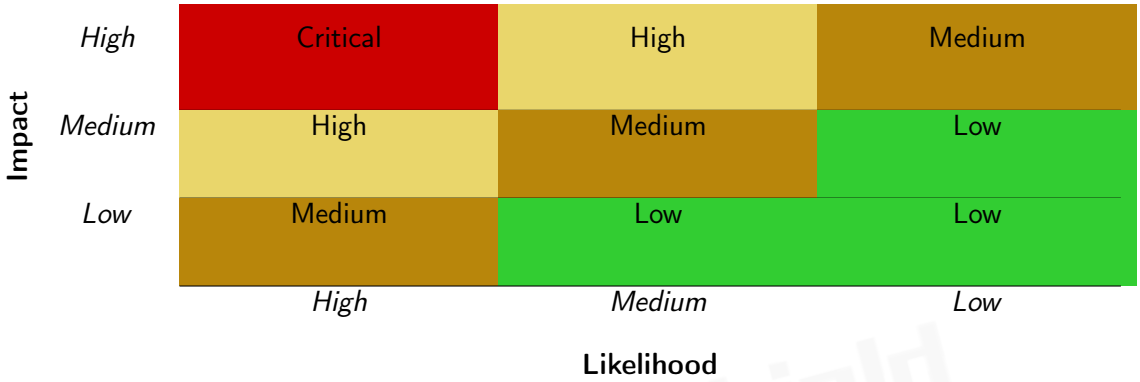
- https://github.com/velo-protocol/DRSv2.git (879a1fc)

- https://github.com/velo-protocol/DRSv2.git (2e10b19)
- https://github.com/velo-protocol/DRSv2.git (b774bcb)

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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

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

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

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

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], 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 an investment advice.

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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the DRSv2 implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	1	■
Medium	1	■
Low	2	■■
Informational	6	■■■■■■
Total	10	

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 vulnerabilities, 1 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 6 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Business Logic Error in releaseReserve()	Business Logics	Fixed
PVE-002	Medium	Privileged Interface to Move Out Assets from Reserve Manager	Business Logics	Fixed
PVE-003	Info.	Unnecessary Payable Functions	Business Logics	Fixed
PVE-004	Low	Incompatibility with Deflationary Tokens as Collateral	Business Logics	Fixed
PVE-005	Info.	Reducing SafeMath Gas Consumption	Business Logics	Fixed
PVE-006	Info.	Re-Entry Attack Risks in Reserve Manager	Security Features	Fixed
PVE-007	Info.	Removing Unused Variables	Coding Practices	Fixed
PVE-008	Info.	Improving Sanity Checks in Feeder	Coding Practices	Fixed
PVE-009	Low	Missing Collision Check in lockReserve()	Business Logics	Fixed
PVE-010	Info.	Incorrect Assertion Messages	Coding Practices	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Business Logic Error in releaseReserve()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `ReserveManager.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In `releaseReserve()`, the current block number, `block.number`, is checked against the block number set in the `lockReserve()`. This ensures the reserved assets are locked until the release time. However, the block number check in line 63 is not correctly implemented such that the reserved assets could be released right after the assets are locked (i.e., at the same block).

```
62      function releaseReserve(bytes32 lockedReserveId, bytes32 assetCode, uint256 amount)
            external {
63          require(block.number.sub(lockedReserves[lockedReserveId].blockNumber).mul(3) <
                heart.getReserveFreeze(assetCode), "release time not reach");
64          require(lockedReserves[lockedReserveId].owner == msg.sender, "only owner can
                release reserve");

66          heart.getCollateralAsset(assetCode).transfer(msg.sender, amount);

68          lockedReserves[lockedReserveId].amount = lockedReserves[lockedReserveId].amount.
                sub(amount);
69      }
```

Listing 3.1: ReserveManager.sol

In addition, the `reserveFeeze` time is set as a number of seconds as shown in the following code snippets.

```
100     function setReserveFreeze(bytes32 assetCode, uint32 newSeconds) external {
101         reserveFreeze[assetCode] = newSeconds;
```

```
102     }
```

Listing 3.2: Heart.sol

But `releaseReserve()` checks the offset between the lock block number and the release block number. When the offset times three exceeds that number of seconds (if implemented correctly), the locked assets are allowed to be released. It seems a mapping from block number to timestamp. However, in Solidity, we can simply use `block.timestamp` to checks the release time accurately.

Recommendation Fix the block number check as follows:

```
62      function releaseReserve(bytes32 lockedReserveId, bytes32 assetCode, uint256 amount)
            external {
63          require(block.number.sub(lockedReserves[lockedReserveId].blockNumber).mul(3) >
                heart.getReserveFreeze(assetCode), "release time not reach");
64          require(lockedReserves[lockedReserveId].owner == msg.sender, "only owner can
                release reserve");

66          heart.getCollateralAsset(assetCode).transfer(msg.sender, amount);

68          lockedReserves[lockedReserveId].amount = lockedReserves[lockedReserveId].amount.
                sub(amount);
69      }
```

Listing 3.3: ReserveManager.sol

In addition, we suggest to keep `block.timestamp` in `lockedReserves[]` to accurately check the release time.

3.2 Privileged Interface to Move Out Assets from Reserve Manager

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ReserveManager.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-287 [4]

Description

In the Reserve Manager, the `injectCollateral()` allows the DRS contract to transfer `amount` of a specific asset indexed by `assetCode` to the `to` address. The normal use case is moving assets from the Reserve Manager contract to the Stable Credit contract.

```
71      function injectCollateral(bytes32 assetCode, address to, uint256 amount) external
            onlyDRSSC {
```

```
72          heart.getCollateralAsset(assetCode).transfer(to, amount);

74          emit InjectCollateral(assetCode, amount);
75      }
```

Listing 3.4: ReserveManager.sol

As shown in the following code snippets. The governor address could set the DRS address, which means the owner of the governor address has full control of all assets in the Reserve Manager.

```
108     function setDrsAddress(address newDrsAddress) external onlyGovernor {
109         drsAddr = newDrsAddress;
110     }
```

Listing 3.5: Heart.sol

For example, if there's a malicious governor address owner, she could set the DRS address to an attacker contract and invoke `injectCollateral()` from the attacker contract for sending arbitrary `amount` of any asset to an arbitrary `to` address.

Recommendation Validate the `to` address to ensure that it is one of the StableCredit contract addresses. In addition, the `assetCode` should be the `collateralAssetCode` of the specific StableCredit contract. However, this can only prevents the governor from setting DRS to an non-DRS address and moving assets to non-StableCredit address. A malicious governor could simply deploy another DRS contract with StableCredit contracts to steal digital assets from the Reserve Manager. We suggest to remove the capability of setting a new DRS after the first-time setup.

3.3 Unnecessary Payable Functions

- ID: PVE-003
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `DigitalReserveSystem.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

While reviewing the DRS contract, we identified that there are three functions declared as payable functions. Typically, we expect `msg.value` is checked inside those functions since they could have the use cases of receiving ether. However, we do not see those use cases. For example, the `mintFromCollateralAmount()` collects the collateral assets from the `msg.sender` and `_mint()` Stable Credit tokens to the `msg.sender` with no ether involved, which means the `payable` modifier is not necessary here.

```
94      function mintFromCollateralAmount(
95          uint256 netCollateralAmount,
96          string calldata assetCode
97      ) external onlyTrustedPartner payable returns (bool) {
98          (IStableCredit stableCredit, ICollateralAsset collateralAsset, bytes32
                collateralAssetCode, bytes32 linkId) = _validateAssetCode(assetCode);

100         // validate stable credit belong to the message sender
101         require(msg.sender == stableCredit.creditOwner(), "DigitalReserveSystem.
                mintFromCollateralAmount: the stable credit does not belong to you");

103         (uint256 mintAmount, uint256 actualCollateralAmount, uint256
                reserveCollateralAmount, uint256 fee) = _calMintAmountFromCollateral(
104             netCollateralAmount,
105             heart.getPriceContract(linkId).get(),
106             heart.getCreditIssuanceFee(),
107             heart.getCollateralRatio(collateralAssetCode),
108             stableCredit.peggedValue(),
109             100000000000
110         );

112         _mint(collateralAsset, stableCredit, mintAmount, fee, actualCollateralAmount,
                reserveCollateralAmount);
```

Listing 3.6: DigitalReserveSystem.sol

Same logic applies to `mintFromStableCreditAmount()` and `rebalance()`.

```
127     function mintFromStableCreditAmount(
128         uint256 mintAmount,
129         string calldata assetCode
130     ) external onlyTrustedPartner payable returns (bool) {
```

Listing 3.7: DigitalReserveSystem.sol

```
127     function rebalance(
128         string calldata assetCode
129     ) external payable returns (bool) {
130         return _rebalance(assetCode);
131     }
```

Listing 3.8: DigitalReserveSystem.sol

Recommendation Remove the payable modifiers from the functions mentioned above.

3.4 Incompatibility with Deflationary Tokens as Collateral

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ReserveManager.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In `lockReserve()`, users provide collateral tokens to the Reserve Manager. However, if the collateral token is a deflationary token, the user may not able to `releaseReserve()` what she had locked. Specifically, in line 49, the collateral assets of `from` are collected by the Reserve Manager with the `transferFrom()` call. But the `amount` is not necessary equal to the amount received by the Reserve Manager if part of the `amount` is burned or paid to the token controller.

```
48      function lockReserve(bytes32 assetCode, address from, uint256 amount) external {
49          heart.getCollateralAsset(assetCode).transferFrom(from, address(this), amount);
50
51          bytes32 lockedReserveId = keccak256(abi.encodePacked(from, assetCode, amount,
                block.number));
52          lockedReserves[lockedReserveId] = LockedReserve(
53              from,
54              assetCode,
55              amount,
56              block.number
57          );
58
59          emit LockReserve(lockedReserveId);
60      }
```

Listing 3.9: ReserveManager.sol

Later on, when the user is about to unlock her assets by `releaseReserve()`, Reserve Manager transfers `amount` of collateral assets to her (line 66) with the book-keeping records updated in line 68. As we mentioned earlier, the Reserve Manager could have received less than `amount` in her `lockReserve()` call, which means this `releaseReserve()` essentially causes a deficit.

```
62      function releaseReserve(bytes32 lockedReserveId, bytes32 assetCode, uint256 amount)
            external {
63          require(block.number.sub(lockedReserves[lockedReserveId].blockNumber).mul(3) <
                heart.getReserveFreeze(assetCode), "release time not reach");
64          require(lockedReserves[lockedReserveId].owner == msg.sender, "only owner can
                release reserve");
65
66          heart.getCollateralAsset(assetCode).transfer(msg.sender, amount);
67
```

```
68          lockedReserves[lockedReserveId].amount = lockedReserves[lockedReserveId].amount.
                sub(amount);
69      }
```

Listing 3.10: ReserveManager.sol

Recommendation Check the balance before and after the `transferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in, which should be verified carefully.

3.5 Reducing SafeMath Gas Consumption

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CustomSafeMath.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `CustomSafeMath` library provides the unsigned integer arithmetic operations with overflow checks. However, we identified that the `div()` operation relies on Solidity exception handling to deal with the zero divisor case. This could be a waste of gas due to the fact that Solidity uses an invalid opcode to terminate the execution with all remaining gas burned.

```
10      function div(uint a, uint b) internal pure returns (uint) {
11          // assert(b > 0); // Solidity automatically throws when dividing by 0
12          uint c = a / b;
13          // assert(a == b * c + a % b); // There is no case in which this doesn't hold
14          return c;
15      }
```

Listing 3.11: CustomSafeMath.sol

In addition, we noticed that all the overflow cases are handled with the assertions. We suggest to replace them with `require()` to save some gas when error occurs.

Recommendation Revert the zero divisor case to keep the remaining gas untouched.

```
10      function div(uint a, uint b) internal pure returns (uint) {
11          require(b > 0, "CustomSafeMath: division by zero");
12          uint c = a / b;
13          // assert(a == b * c + a % b); // There is no case in which this doesn't hold
14          return c;
```

```
15      }
```

Listing 3.12: CustomSafeMath.sol

Moreover, we suggest to replace all `assert()` with `require()`.

3.6 Re-Entry Attack Risks in Reserve Manager

- ID: PVE-006
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `ReserveManager.sol`
- Category: Security Features [6]
- CWE subcategory: CWE-269 [3]

Description

In Reverse Manager, the `lockReserve()` and `releaseReserve()` functions are prone to re-entry attack similar to the Lendf.Me hack [16]. As shown in the code snippets, the collateral assets are `transferFrom()` from `from` to `address(this)` in line 49. After that, the `lockedReserves[lockedReserveId]` is set in line 51. This is pretty similar to Lendf.Me's `supply()`.

```
48      function lockReserve(bytes32 assetCode, address from, uint256 amount) external {
49          heart.getCollateralAsset(assetCode).transferFrom(from, address(this), amount);
50
51          bytes32 lockedReserveId = keccak256(abi.encodePacked(from, assetCode, amount,
                block.number));
52          lockedReserves[lockedReserveId] = LockedReserve(
53              from,
54              assetCode,
55              amount,
56              block.number
57          );
58
59          emit LockReserve(lockedReserveId);
60      }
```

Listing 3.13: ReserveManager.sol

Suppose the collateral asset is an ERC777 token, a bad actor could hijack the `transferFrom()` call and invoke the `releaseReserve()` function. Inside `releaseReserve()`, the collateral assets are `transfer()` to the `msg.sender` in line 66. Then, `lockedReserves[lockedReserveId].amount` is updated in line 68. When the code execute goes back to `lockReserve()`, the `lockedReserves[lockedReserveId].amount` is reset.

```
62      function releaseReserve(bytes32 lockedReserveId, bytes32 assetCode, uint256 amount)
            external {
```

```
63          require(block.number.sub(lockedReserves[lockedReserveId].blockNumber).mul(3) <
                heart.getReserveFreeze(assetCode), "release time not reach");
64          require(lockedReserves[lockedReserveId].owner == msg.sender, "only owner can
                release reserve");

66          heart.getCollateralAsset(assetCode).transfer(msg.sender, amount);

68          lockedReserves[lockedReserveId].amount = lockedReserves[lockedReserveId].amount.
                sub(amount);
69      }
```

Listing 3.14: ReserveManager.sol

Fortunately, the hash function to derive `lockedReserveId` includes `amount`. This prevents the bad actor to `lockReserve()` a small `amount` to reset the `lockedReserves[lockedReserveId].amount`. We set the severity of this issue to informational.

Recommendation Apply the Check-Effects-Interactions design pattern [2].

3.7 Removing Unused Variables

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Price.sol, Feeder.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-287 [4]

Description

The `lagAddr` in Price contract is replaced by `feederAddr` but the declaration in line 20 is not removed.

```
9   contract Price is Initializable, IPrice {

11      using SafeMath for uint256;

13      address public owner;
14      modifier onlyOwner {
15          require(msg.sender == owner, "Price.onlyOwner: The message sender is not found
                or does not have sufficient permission");
16          _;
17      }

19      uint256 public price=0;
20      address public lagAddr;
21      address  public  feederAddr;
```

Listing 3.15: ReserveManager.sol

Also, `lastOperationTime` (line 21), `operationCoolDown` (line 22), `valueTimestamp` (line 28), and `active` (line 29) in Feeder contract are not used as well.

```
6  contract Feeder is IFeeder {
7      using CustomSafeMath for uint;
8      struct FeedPrice {
9          uint priceInWei;
10         uint timeInSecond;
11         address source;
12     }
13     FeedPrice public firstPrice;
14     FeedPrice public secondPrice;
15     FeedPrice public lastPrice;
16     uint public priceTolInBP = 500;
17     uint constant BP_DENOMINATOR = 10000;
18     uint public priceFeedTimeTol = 1 minutes;
19     uint public priceUpdateCoolDown=1 minutes;
20     uint public numOfPrices = 0;
21     uint public lastOperationTime;
22     uint public operationCoolDown=0;
23     bool public started = false;
24     address public priceFeed1;
25     address public priceFeed2;
26     address public priceFeed3;
27     address public owner;
28     uint256 public valueTimestamp;
29     bool public active;
```

Listing 3.16: Feeder.sol

Recommendation Remove unused variables.

3.8 Improving Sanity Checks in Feeder

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Feeder.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-287 [4]

Description

In the Feeder contract, the `commitPrice()` function allows price feeders to commit a new price. When the delta between the `lastPrice` and the new price is greater than `priceTolInBP/BP_DENOMINATOR`, the oracle system needs a second price feeder. However, if the second price comes after `priceFeedTimeTol`, the `firstPrice` is accepted. While reviewing the `commitPrice()` function, we identified that the check

in line 114 may not happen due to the fact that `firstPrice.timeInSecond` should be always less or equal to `timeInSecond`.

```
112                 // if second price times out, use first one
113                 if (firstPrice.timeInSecond.add(priceFeedTimeTol) < timeInSecond
114                     firstPrice.timeInSecond.sub(priceFeedTimeTol) > timeInSecond) {
115                     acceptPrice(firstPrice.priceInWei, firstPrice.timeInSecond,
                            firstPrice.source);
```

Listing 3.17: Feeder.sol

Same theory applies to the third price case in line 139.

```
137                 // if third price times out, use first one
138                 if (firstPrice.timeInSecond.add(priceFeedTimeTol) < timeInSecond
139                     firstPrice.timeInSecond.sub(priceFeedTimeTol) > timeInSecond) {
140                     acceptedPriceInWei = firstPrice.priceInWei;
```

Listing 3.18: Feeder.sol

Besides, we also identified that the arithmetic operation in line 129 is not using safemath although it would not overflow as `priceUpdateCoolDown` should be seconds or minutes.

```
129             if (timeInSecond > firstPrice.timeInSecond + priceUpdateCoolDown) {
130                 if ((firstPrice.source == msg.sender  secondPrice.source == msg.sender))
131                     acceptPrice(priceInWei, timeInSecond, msg.sender);
132                 else
133                     acceptPrice(secondPrice.priceInWei, timeInSecond, secondPrice.source
                            );
```

Listing 3.19: Feeder.sol

Recommendation Remove unnecessary checks.

```
112                 // if second price times out, use first one
113                 if (firstPrice.timeInSecond.add(priceFeedTimeTol) < timeInSecond ) {
114                     acceptPrice(firstPrice.priceInWei, firstPrice.timeInSecond,
                            firstPrice.source);
```

Listing 3.20: Feeder.sol

```
137                 // if third price times out, use first one
138                 if (firstPrice.timeInSecond.add(priceFeedTimeTol) < timeInSecond ) {
139                     acceptedPriceInWei = firstPrice.priceInWei;
```

Listing 3.21: Feeder.sol

Besides, always use safemath in all arithmetic operations.

```
129             if (timeInSecond > firstPrice.timeInSecond.add(priceUpdateCoolDown)) {
130                 if ((firstPrice.source == msg.sender  secondPrice.source == msg.sender))
131                     acceptPrice(priceInWei, timeInSecond, msg.sender);
132                 else
```

```
133                     acceptPrice(secondPrice.priceInWei, timeInSecond, secondPrice.source
                            );
```

Listing 3.22: Feeder.sol

3.9 Missing Collision Check in lockReserve()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ReserveManager.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `lockReserve()` function uses the owner address, the `assetCode`, the `amount`, and the `block.number` to derive the `lockedReserveId`. Then, the `lockedReserveId` is used to index the `lockedReserves[]` array to book-keep the reserve fund. With the book-keeping record, `lockedReserves[lockedReserveId]`, the owner could release the locked assets through `releaseReserve()`. However, if an user accidently `lockReserve()` the same amount of the same collateral asset in the same block, the previously locked assets cannot be released anymore as the old record is simply overwritten by the new record.

```
48      function lockReserve(bytes32 assetCode, address from, uint256 amount) external {
49          heart.getCollateralAsset(assetCode).transferFrom(from, address(this), amount);
50
51          bytes32 lockedReserveId = keccak256(abi.encodePacked(from, assetCode, amount,
                block.number));
52          lockedReserves[lockedReserveId] = LockedReserve(
53              from,
54              assetCode,
55              amount,
56              block.number
57          );
58
59          emit LockReserve(lockedReserveId);
60      }
```

Listing 3.23: ReserveManager.sol

Recommendation Ensure `lockedReserves[lockedReserveId]` is not used.

```
48      function lockReserve(bytes32 assetCode, address from, uint256 amount) external {
49          heart.getCollateralAsset(assetCode).transferFrom(from, address(this), amount);
50
51          bytes32 lockedReserveId = keccak256(abi.encodePacked(from, assetCode, amount,
                block.number));
```

```
52          require( lockedReserves[lockedReserveId].owner == address(0) );
53          lockedReserves[lockedReserveId] = LockedReserve(
54              from,
55              assetCode,
56              amount,
57              block.number
58          );

60          emit LockReserve(lockedReserveId);
61      }
```

Listing 3.24: ReserveManager.sol

3.10 Incorrect Assertion Messages

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DigitalReserveSystem.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-287 [4]

Description

Some typos are identified in the assertion messages.

Case I `getStableCreditAmount()` line 241

```
238     function getStableCreditAmount(
239         string calldata assetCode
240     ) external view returns ( uint256) {
241         require(bytes(assetCode).length > 0 && bytes(assetCode).length <= 12, "
                DigitalReserveSystem.collateralHealthCheck: invalid assetCode format");

243         (IStableCredit stableCredit,,, ) = _validateAssetCode(assetCode);
244         return stableCredit.totalSupply();

246     }
```

Listing 3.25: DigitalReserveSystem.sol

Case II `_rebalance()` line 251

```
248     function _rebalance(
249         string memory assetCode
250     ) private returns (bool) {
251         require(bytes(assetCode).length > 0 && bytes(assetCode).length <= 12, "
                DigitalReserveSystem.rebalance: invalid assetCode format");
```

Listing 3.26: DigitalReserveSystem.sol

Recommendation Fix assertion messages.

```
238     function getStableCreditAmount(
239         string calldata assetCode
240     ) external view returns ( uint256) {
241         require(bytes(assetCode).length > 0 && bytes(assetCode).length <= 12, "
                DigitalReserveSystem.getStableCreditAmount: invalid assetCode format");

243         (IStableCredit stableCredit,,, ) = _validateAssetCode(assetCode);
244         return stableCredit.totalSupply();

246     }
```

Listing 3.27: DigitalReserveSystem.sol

```
248     function _rebalance(
249         string memory assetCode
250     ) private returns (bool) {
251         require(bytes(assetCode).length > 0 && bytes(assetCode).length <= 12, "
                DigitalReserveSystem._rebalance: invalid assetCode format");
```

Listing 3.28: DigitalReserveSystem.sol

3.11 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, we always suggest using fixed compiler version whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.0;` instead of `pragma solidity ^0.5.0;`.

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 the mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the DRSv2 implementation. 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.

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 [11, 12, 13, 14, 17].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [18] 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 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

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] Ethereum. Use the Checks-Effects-Interactions Pattern. https://solidity.readthedocs.io/en/v0.5.0/security-considerations.html#use-the-checks-effects-interactions-pattern.

[3] MITRE. CWE-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

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

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

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

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

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

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

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

[11] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[12] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[13] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[14] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://blog.peckshield.com/2020/04/19/erc777/.

[17] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[18] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.