SMART CONTRACT AUDIT REPORT

for

SHIELD

Prepared By: Shuxiao Wang

PeckShield
May 20, 2021

Document Properties

Client	Shield
Title	Smart Contract Audit Report
Target	Shield
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 20, 2021	Xuxian Jiang	Final Release
1.0-rc1	May 1, 2021	Xuxian Jiang	Release Candidate #1
0.4	April 28, 2021	Xuxian Jiang	Add More Findings #3
0.3	April 20, 2021	Xuxian Jiang	Add More Findings #2
0.2	April 19, 2021	Xuxian Jiang	Add More Findings #1
0.1	April 12, 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 **Shield** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Shield

`Shield` is a non-custodial decentralized derivative exchange that trades risk-free perpetual contracts. The risk-free perpetual contract is the solution from `Shield` to the existing limitations within the decentralized derivative ecosystem. It uses a combination of 0 position loss, a dual liquidity pool model, high leverages, a decentralized brokerage system, and external liquidators to counteract the existing limitations. This new perpetual product goes above and beyond the mentioned limitations in the current derivative products, aiming to get to be a more competitive space and bring DeFi the next generation of global decentralized derivative infrastructure.

The basic information of Shield is as follows:

Table 1.1: Basic Information of Shield

Item	Description
Issuer	Shield
Type	Ethereum and BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 20, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Shield assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/Jackluren/DDS-Contract-Test.git (6aecab7)

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

- https://github.com/Jackluren/DDS-Contract-Test.git (8c49a63)
- https://github.com/Jackluren/DDS-Contract-Test.git (51f24d4)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [13]:

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

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

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

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

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	6	■■■■■■
Informational	1	■
Total	9	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Shield Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Re-Architecture Of Common Parameters And Configurations	Coding Practices	Confirmed
PVE-002	Low	Possible Duplicate AirDrop User In setAirdropUsers()	Business Logic	Fixed
PVE-003	Low	Business Logic Error In claimRewardsForBroker()	Business Logic	Fixed
PVE-004	Medium	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-005	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-006	Low	Redundant Code Removal	Coding Practices	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Low	Potential LP2 Front-running For Reduced Loss	Business Logic	Confirmed
PVE-009	Low	Proper Liquidation Reward Attribution in migrationContract()	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Re-Architecture Of Common Parameters And Configurations

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

The Shield protocol has a number of essential contracts for different functionalities and duties: formular, riskFundAddr, repayFudAddr, DDSBrokerAddr, and DDSLiquidorAddr. Our analysis with these essential contracts shows that they share a number of configurations, parameters, and functions. These shared states and functions are better relocated to a dedicated registry or address provider.

To elaborate, we use DDSBroker and DDSLiquidor contracts as an example. Both contracts have defined the states DAIErc, USDTErc, USDCErc, and contractAddressesForLP1, as well as their setter routines as shown below.

```
    function setStableContractAddress(address _daiAddr, address _usdtAddr, address _usdcAddr) public onlyOwner{
        DAIErc = IERC20(_daiAddr);
        USDTErc = IERC20(_usdtAddr);
        USDCErc = IERC20(_usdcAddr);
    }

    //
    function enableContractAddressesForLP1(address[] memory _addresses) public onlyOwner {
        for(uint256 i = 0; i < _addresses.length; i++){
            contractAddressesForLP1[_addresses[i]] = true;
        }
    }
```

```
    //
    function disableContractAddressesForLP1(address[] memory _addresses) public
        onlyOwner{
        for(uint256 i = 0; i < _addresses.length; i++){
            contractAddressesForLP1[_addresses[i]] = false;
        }
    }
```

Listing 3.1: A number of setters in DDSBroker

```
    function setStableContractAddress(address _daiAddr, address _usdtAddr, address
        _usdcAddr) public onlyOwner{
        DAIErc = IERC20(_daiAddr);
        USDTErc = IERC20(_usdtAddr);
        USDCErc = IERC20(_usdcAddr);
    }

    function setPriceProvider(address _daiAddr, address _usdtAddr, address _usdcAddr,
        address _gasPriceAddr) public onlyOwner{
        priceProviderByDAIETH = AggregatorV3Interface(_daiAddr);
        priceProviderByUSDTETH = AggregatorV3Interface(_usdtAddr);
        priceProviderByUSDCETH = AggregatorV3Interface(_usdcAddr);
        priceProviderByGASPRICE = AggregatorV3Interface(_gasPriceAddr);
    }

    //
    function enableContractAddressesForLP1(address[] memory _addresses) public onlyOwner
        {
        for(uint256 i = 0; i < _addresses.length; i++){
            contractAddressesForLP1[_addresses[i]] = true;
        }
    }

    //
    function disableContractAddressesForLP1(address[] memory _addresses) public
        onlyOwner{
        for(uint256 i = 0; i < _addresses.length; i++){
            contractAddressesForLP1[_addresses[i]] = false;
        }
    }
```

Listing 3.2: A number of setters in DDSLiquidor

Apparently, the scattered duplicates of these states and their setters bring additional operation overhead and may cause inconsistency. From the maintenance and protocol consistency perspective, it is strongly suggested to relocate these common states and functions to a dedicated registry-style contract.

Recommendation Re-architect current design to have a registry contract with common states and routines.

Status This issue has been confirmed.

3.2 Possible Duplicate AirDrop User In setAirdropUsers()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DDSAirdrop
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In order to engage the community and broaden the adoption, the `Shield` protocol has developed a `DDSAirdrop` contract to provide airdrops to protocol users. Within this contract, there is a function `setAirdropUsers()` to add airdrop users and count the total number of airdrop users. It comes to our attention that the provided airdrop user may be a duplicate and the current approach of calculating the total number of airdrop users does not exclude duplicate users.

```
    function setAirdropUsers(address[] memory _users) public onlyOwner {
        for(uint i = 0; i < _users.length; i++) {
            airdropUsers[_users[i]]=true;
            userNumber = userNumber.add(1);
        }
    }
```

Listing 3.3: DDSAirdrop::setAirdropUsers()

As a result, the counted total number of airdrop users may be unexpectedly larger and the `claim()` function may give out less tokens for airdrop. Fortunately, the permissioned `setAirdropUsers()` function ensures that only owner may be able to provide airdrop users, which greatly alleviate this concern.

```
    function claim() public {
        require(userNumber > 0, "no users could claim");
        require(airdropUsers[msg.sender], "msg.sender could not claim");
        airdropUsers[msg.sender] = false;
        uint256 amount = getSupply().div(userNumber);
        userNumber = userNumber.sub(1);
        require(amount > 0, "no tokens to claim");
        _safeTransfer(address(_token), msg.sender, amount);
    }
```

Listing 3.4: DDSAirdrop::claim()

Recommendation Revise the `setAirdropUsers()` logic to not count duplicate airdrop users, if any.

Status This issue has been fixed in this commit: `51f24d4`.

3.3 Business Logic Error In claimRewardsForBroker()

- ID: PVE-003
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `DDSRewards`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Shield` protocol, there is a `DDSRewards` contract that maintains reward states about current `brokers`. The reward comes from 40% of the exchange fee from opening accounts. In the following, we examine the logic reward accounting and distribution in `DDSRewards`.

To elaborate, we show below the `claimRewardsForBroker()` routine. As the name indicates, this routine allows for the broker to claim current rewards in three different token types: `DAI`, `USDT`, and `USDC`. It implements a rather straightforward logic and distributes the rewards for each token type. However, we notice that the `USDC`-related reward logic improperly records the `brokersClaimedRewards` state (line 157). The correct state after claim should be `brokersRewards[msg.sender][USDC].brokersClaimedRewards.add(usdcRewards)`, not current `brokersRewards[msg.sender][USDC].brokersClaimedRewards.add(usdtRewards)`.

```
132     function claimRewardsForBroker() public {
133         // dai rewards
134         uint256 daiRewards = brokersRewards[msg.sender][DAI].brokersClaimRewards;
135         uint256 usdtRewards = brokersRewards[msg.sender][USDT].brokersClaimRewards;
136         uint256 usdcRewards = brokersRewards[msg.sender][USDC].brokersClaimRewards;
137         require(daiRewards != 0  usdtRewards != 0  usdcRewards != 0, "no stable coins to
                claim");
138
139         if (daiRewards > 0) {
140             brokersRewards[msg.sender][DAI].brokersClaimRewards = 0;
141             brokersRewards[msg.sender][DAI].brokersClaimedRewards = brokersRewards[msg.
                    sender][DAI].brokersClaimedRewards.add(daiRewards);
142             _safeTransfer(address(DAIErc), msg.sender, daiRewards);
143         }
144
145         // usdt rewards
146         uint256 actualUSDTRewards = usdtRewards.div(compensatoryDecimal);
147         if (actualUSDTRewards > 0) {
148             brokersRewards[msg.sender][USDT].brokersClaimRewards = 0;
149             brokersRewards[msg.sender][USDT].brokersClaimedRewards = brokersRewards[msg.
                    sender][USDT].brokersClaimedRewards.add(usdtRewards);
150             _safeTransfer(address(USDTErc), msg.sender, actualUSDTRewards);
```

```
        }

        // usdc rewards
        uint256 actualUSDCRewards = usdcRewards.div(compensatoryDecimal);
        if (actualUSDCRewards > 0) {
            brokersRewards[msg.sender][USDC].brokersClaimRewards = 0;
            brokersRewards[msg.sender][USDC].brokersClaimedRewards = brokersRewards[msg.
                sender][USDC].brokersClaimedRewards.add(usdtRewards);
            _safeTransfer(address(USDCErc), msg.sender, actualUSDCRewards);
        }
    }
```

Listing 3.5: DDSRewards::claimRewardsForBroker()

Recommendation Properly record the `brokersClaimedRewards` state about the broker.

Status This issue has been fixed in this commit: `51f24d4`.

3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [10]
- 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` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there are a number of occasions where the `checks-effects-interactions` principle is violated. Using the `DDSDAIContract` as an example, the `deposit()` 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 94) starts before effecting the update on internal states (lines 96 – 97), 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 same entry function.

```
    function deposit(uint256 amount) public {
        require(amount >=minAmount,'less than minAmount');
        _safeTransferFrom(tokenAddr, msg.sender,address(this), amount);
        AccountInfo  storage userAcc = userAccount[msg.sender];
        userAcc.depositAmount = userAcc.depositAmount.add(amount);
        userAcc.availableAmount = userAcc.availableAmount.add(amount);

        emit DDSDeposit(msg.sender, address(this), amount);
    }
```

Listing 3.6: DDSDAIContract::deposit()

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`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`. Note similar issues exist in other functions, including `deposit()`, `closecontract()`, and `migrationContract()` in the following contracts: `DDSDAIContract`, `DDSUSDTContract`, and `DDSUSDCContract`. The adherence of `checks-effects-interactions` best practice in these routines is strongly recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in this commit: `cb1198e`.

3.5 Improved Precision By Multiplication And Division Reordering

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `DDSBuyback`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `DDSBuyBack::calcDDSPrice()` as an example. This routine is used to calculate the DDS price.

```
79     function calcDDSPrice() public view returns(uint256) {
80         (uint256 ddsAmount, uint256 daiAmount, uint256 usdtAmount, uint256 usdcAmount) =
             getBuyBackInfo();
81         uint256 stableAmount = daiAmount.add(usdtAmount.mul(compensatoryDecimal)).add(
            usdcAmount.mul(compensatoryDecimal));
82         uint256 ddsPrice = stableAmount.mul(priceDecimal).div(ddsAmount.mul(
            ddsPriceForBuyBackNumerator).div(ddsPriceForBuyBackDenominator));
83         return ddsPrice;
84     }
```

Listing 3.7: DDSBuyBack::calcDDSPrice()

We notice the calculation of `ddsPrice` (line 82) involves mixed multiplication and devision. To avoid unnecessary precision loss, it is better to compute it with the following equation: `_ddsPrice = stableAmount.mul(priceDecimal).mul(ddsPriceForBuyBackDenominator).div(ddsAmount).mul(ddsPriceFor BuyBackNumerator))`. It is important to that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Note the `calcAndSendRewardsForLP2()` in `DDSRewards` as well as `sendDAI()/sendUSDT()/sendUSDC()` in `DDSLiquidor` can benefit from the same optimization.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in this commit: `51f24d4`.

3.6 Redundant Code Removal

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The `Shield` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the `DDSDAIContract` smart contract has so far imported at least five 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 `checkOrderIsAtRisk` routine, it contains repeated inter-contract calls to obtain the current margin amount (lines $512-513$ and lines $516-517$). With that, it is suggested to cache and then re-use the return value.

```
502  function checkOrderIsAtRisk(uint256 orderID) public view returns(bool){
503          Order memory order = orders[orderID];
504          require(order.state == State.ACTIVE,"order state wrong");
505          uint256 puborPriPool = checkOrder(orderID);
506          require(puborPriPool >0,"not matched");
507          //
508          uint256 currentPrice = getCurrPriceByEx(order.exchangeType); //real
509          uint256 calProfit = payProfit(currentPrice,orderID);
510          uint256 marginAmount;
511          if (puborPriPool ==2) {//lp2RiskControl
512              (marginAmount,)  = privPool.geMarginAmount(orderID);
513               if(calProfit > getLpMarginAmount(orderID))
514                   return true;
515          }else{
516              (marginAmount,) = pubPool.geMarginAmount(orderID);
517              if(calProfit > getLpMarginAmount(orderID).mul(50).div(100))
518                return true;
519          }
520  }
```

Listing 3.8: DDSDAIContract::checkOrderIsAtRisk()

In addition, the `closecontract()` routine has an internal variable `repayPeriondFee`, which is always calculated to be 0. Therefore, we suggest to simplify the routine by avoiding the use of this variable.

Moreover, the states `airdropUsers` in `DDSBuyBack` are defined, but not used. Also, the state `accIndex` is defined in `DDSDAIPools2`, but not used either.

Recommendation Consider the removal of the redundant code in `checkOrderIsAtRisk()` and `closecontract()`.

Status This issue has been fixed in this commit: `51f24d4`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Shield` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components, i.e., `riskFund`, `DDSBuyBack`, and `DDSReward`.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Shield` protocol.

```
    function setDDSReward(address _ddsReward) public onlyOwner{
        ddsReward = DDSRewardsInterface(_ddsReward);
    }

    function setRiskFundAddr(address _riskFundAddr) public onlyOwner{
        riskFundAddr = _riskFundAddr;
    }
```

Listing 3.9: Various Setters in DDSLiquidor

```
    function setOtherUnlockAmount(uint256 _otherUnlockAmount) public onlyOwner{
        otherUnlockAmount = _otherUnlockAmount;
    }

    function setDDSBuyBackParamaters(uint256 _ddsPriceForBuyBackNumerator, uint256
        _ddsPriceForBuyBackDenominator) public onlyOwner{
        ddsPriceForBuyBackNumerator = _ddsPriceForBuyBackNumerator;
        ddsPriceForBuyBackDenominator = _ddsPriceForBuyBackDenominator;
    }
```

Listing 3.10: Various Setters in DDSBuyBack

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the `owner` will be managed by a multi-sig account.

We point out that a compromised `owner` account is capable of modifying current protocol configuaration with adverse consequences, including permanent lock-down of user funds.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract.

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance/controller privileges.

3.8 Potential LP2 Front-running For Reduced Loss

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Shield` protocol, there is an innovative feature, i.e., the `dual liquidity pool model`. When a new order is created, the private pool (`LP2`) is first checked for a possible match before evaluating with the public pool (`LP1`). Naturally, when there is a need to liquidate a private pool-matched order, the respective provider in `LP2` will be deducted for possible deficit, if any.

To elaborate, we show below the `DDSDAIPools2::close()` routine. As the name indicates, this routine is designed to close an order. It implements a rather complex logic to distribute the user profit and adjust the `LP2` liquidity. It comes to our attention that when the margin amount and fee is sufficient to cover the user profit, there is a need to deduct the deficit from the available balance of the `LP2` provider.

```
    function close(uint256 id,uint256 profit,uint256 holdFee) public onlyKeeeper returns (uint256 userProfit){
        //move hold Fee to lps user
        uint256 lpId = matchIds[id]-1;
        address makerAddr = lockedLiquidity[lpId].makerAddr;
        lockedLiquidity[lpId].locked = false;
        uint256 marginAmount = lockedLiquidity[lpId].marginAmount;
        uint256 marginFee = lockedLiquidity[lpId].marginFee;
        LP2Account storage lpqAccount = lpAccount[makerAddr];
        lpqAccount.amount = lpqAccount.amount.add(holdFee);
        lpqAccount.availableAmount = lpqAccount.availableAmount.add(holdFee);
        //
        if (profit > 0){
            if (marginAmount >= profit){
                userProfit = profit;
                lpqAccount.availableAmount = lpqAccount.availableAmount.add(marginFee).
                    add(marginAmount.sub(profit));
                lpqAccount.amount = lpqAccount.amount.sub(profit);
```

```
                // lpqAccount.lockedAmount = lpqAccount.lockedAmount.sub(marginAmount.add(marginFee));
            }else {
                //
                if(marginAmount.add(marginFee) >= profit){//
                    //
                    userProfit = profit;
                    uint256 mvRiskFund = marginAmount.add(marginFee).sub(profit);
                    //update account
                    lpqAccount.amount = lpqAccount.amount.sub(marginAmount.add(marginFee));
                    //update account
                    _safeTransfer(tokenAddress, riskFundAddr, mvRiskFund);
                }else{//
                        uint256 fixAmount = profit.sub(marginAmount.add(marginFee));
                        if(lpqAccount.availableAmount >= fixAmount){
                            userProfit = profit;
                            //update account
                            lpqAccount.amount = lpqAccount.amount.sub(marginAmount.add(marginFee)).sub(fixAmount);
                            lpqAccount.availableAmount = lpqAccount.availableAmount.sub(fixAmount);
                        }else{
                            uint256 newFixAmount = fixAmount.sub(lpqAccount.availableAmount);
                            uint256 riskFund = getRiskFundAmount();
                            if(riskFund >= newFixAmount){
                                userProfit = profit;
                                 //
                                 _safeTransferFrom(tokenAddress, riskFundAddr, address(this), newFixAmount);
                             }else{//
                                userProfit = marginAmount.add(marginFee).add(lpqAccount.availableAmount).add(riskFund);
                                 //
                                 _safeTransferFrom(tokenAddress, riskFundAddr, address(this), riskFund);
                            }
                            //update account
                            lpqAccount.amount = lpqAccount.amount.sub(marginAmount.add(marginFee).add(lpqAccount.availableAmount));
                            lpqAccount.availableAmount = 0;
                        }
                }
            }
        }else{
                //update account
                lpqAccount.availableAmount = lpqAccount.availableAmount.add(marginAmount.add(marginFee));
                //lpqAccount.lockedAmount = lpqAccount.lockedAmount.sub(marginAmount.add(marginFee));
        }
```

```
        //update account freeze info
        lpqAccount.lockedAmount = lpqAccount.lockedAmount.sub(marginAmount.add(marginFee
            ));

        //
        _safeTransferFrom(tokenAddress, msg.sender,address(this), holdFee);
        _safeTransfer(tokenAddress,msg.sender,userProfit);
    }
```

Listing 3.11: DDSDAIPools2::close()

Meanwhile, we notice that when the above deficit situation occurs, the `LP2` may be able to front-run the close operation by explicitly calling `withdraw()`. As shown in the following code snippet, by effectively transferring out all available amount in the `LP2` account (line 74), the `LP2` can avoid any deficit from being charged. In other words, the cost will be shifted from the `LP2` account to the risk fund. Certainly, after the close operation, the `LP2` may choose add back the removed liquidity.

```
66      function withdraw(uint256 amount) public {//
67          require(lastProvideTm[msg.sender].add(lockupPeriod) <= block.timestamp,"Withdraw
                is locked up");
68          require(amount > 0, "Pool: Amount is too small");
69          require(amount <=  lpAccount[msg.sender].availableAmount,"Pool:Please lower the
              amount.");
70          _safeTransfer(tokenAddress,msg.sender,amount);
71          LP2Account memory lp2Acccount = lpAccount[msg.sender];
72          //update user account info
73          lp2Acccount.amount = lp2Acccount.amount.sub(amount);
74          lp2Acccount.availableAmount = lp2Acccount.availableAmount.sub(amount);
75          lpAccount[msg.sender] = lp2Acccount;
76
77          emit Withdraw(msg.sender, amount);
78      }
```

Listing 3.12: DDSDAIPools2::withdraw()

Note that this is a common issue among current private pool implementations, including `DDSDAIPools2`, `DDSUSDTPools2`, and `DDSUSDCPools2`.

Recommendation This is in essence a sandwich-based attack. While it is an inherently challenging issue, it is suggested to revisit the lock up mechanism in the private pool to mitigate this issue.

Status This issue has been confirmed.

3.9 Proper Liquidation Reward Attribution in migrationContract()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Shield` protocol, there is a migration feature that periodically validates the states of the given orders. This feature allows for timely liquidation of underwater orders. To incentivize external liquidators, the protocol provides the refunds in the amount of 150% of used gas via `DDSLiquidor::calcLiquidorAmount()`. In the following, we examine the gas refund logic.

To elaborate, we show below the `DDSUSDCContract::close()` routine. This routine implements a rather straightforward logic in iterating the given order set and for each one perform the intended `migration()` operation. Note that this routine properly measures the gas consumption for the routine and refunds the cost via `calcLiquidorAmount()` (line 246).

```
225   function migrationContract(uint256[] memory orderIDs) public{
226      uint256 beforeGas = gasleft();
227       for(uint256 i =0;i <orderIDs.length;i++ ){
228           uint256 orderID = orderIDs[i];
229           Order memory order = orders[orderID];
230           if(order.state != State.ACTIVE) continue;
231           //get curr update
232           uint256 currDate = DateTime.getCurrentDay();
233          //get curr migration Time(1:0:0)
234           uint256 currMigrationTm = DateTime.getMigrationTime(migrationHour);
235           if(migrationInfo[orderID][currDate] == 0){
236                if(block.timestamp >= currMigrationTm){
237                    migrationInfo[orderID][currDate] = block.timestamp;
238                    //calculate days
239                    uint256 intervalDay = calculatIntervalDay(migrationTime[orderID],
          currMigrationTm);
240                   migrationTime[orderID] = currMigrationTm;
241                   migration(orderID,intervalDay);
242               }
243           }
244       }
245       uint256 afterGas = beforeGas - gasleft();
246      _calcLiquidorAmount(DDSLiquidorAddr,msg.sender, 1, afterGas);
247
248   }
```

Listing 3.13: DDSUSDCContract::migrationContract()

It comes to our attention that the call to `calcLiquidorAmount()` (line 246) is given the token type of 1, which represents $DAI = 1$. Apparently, the token type in `DDSUSDCContract` should be $USDC = 3$. Similarly, the token type in `DDSUSDTContract` should be $USDT = 2$, which is currently still 1.

Recommendation Properly correct the token type for gas funds in `DDSUSDCContract` and `DDSUSDTContract`.

Status This issue has been fixed in this commit: `cb1198e`.

4 Conclusion

In this audit, we have analyzed the Shield design and implementation. The system presents a unique, robust offering as a decentralized perpetual contracts for crypto derivatives trading. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

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

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