SMART CONTRACT AUDIT REPORT

for

DERI PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

February 03, 2021

Document Properties

Client	Deri protocol
Title	Smart Contract Audit Report
Target	Deri protocol
Version	1.0
Author	Ruiyi Zhang
Auditors	Ruiyi Zhang, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 03, 2021	Ruiyi Zhang	Final Release
1.0-rc1	February 03, 2021	Ruiyi Zhang	Release Candidate #1
0.2	January 25, 2021	Ruiyi Zhang	Additional Findings
0.1	January 20, 2021	Ruiyi Zhang	Initial Draft

Contact

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

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

Contents

1 Introduction 4
- 1.1 About Deri protocol . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Timely _updateCumuFundingRate During Pool Liquidity Changes . . . 11
- 3.2 Sandwiched Liquidation To Bypass isQualifiedLiquidator() Check . . . 13
- 3.3 Potential Replay Of Signed Prices . . . 14
- 3.4 Potential Front-Running For setPool() . . . 15
- 3.5 Lack Of Sanity Checks For System Parameters . . . 16
- 3.6 Incompatibility with Deflationary/Rebasing Tokens . . . 18

4 Conclusion 20

References 21

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Deri` 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 Deri protocol

The `Deri` protocol is a decentralized protocol for users to exchange risk exposures precisely and capital-efficiently. In other words, it is the DeFi way to trade derivatives. This is achieved by liquidity pools playing the roles of counter-parties for users. With `Deri` protocol, risk exposures are tokenized as non-fungible tokens so that they can be imported into other DeFi projects for their own financial purposes.

The basic information of the `Deri` protocol is as follows:

Table 1.1: Basic Information of The `Deri` Protocol

Item	Description
Issuer	Deri protocol
Website	https://deri.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 03, 2021

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

- https://github.com/dfactory-tech/deriprotocol (cae2151)

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

- https://github.com/dfactory-tech/deriprotocol (c3041af)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ 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 [8]:

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

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

To evaluate the risk, we go through a 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

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 1.4: 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 Deri protocol 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	1	■
Low	2	■ ■
Informational	3	■ ■ ■
Total	6	

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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and and 3 informational recommendations.

Table 2.1: Key Deri protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely _updateCumuFundingRate During Pool Liquidity Changes	Business Logic	Resolved
PVE-002	Low	Sandwiched Liquidation To Bypass isQualifiedLiquidator() Check	Time and State	Resolved
PVE-003	Informational	Potential Replay Of Signed Prices	Business Logic	Resolved
PVE-004	Low	Potential Front-Running For setPool()	Time and State	Resolved
PVE-005	Informational	Lack Of Sanity Checks For System Parameters	Coding Practices	Resolved
PVE-006	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Timely _updateCumuFundingRate During Pool Liquidity Changes

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `PerpetualPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Similar to other `perpetual` protocols, the `Deri` protocol has a funding payment mechanism. Traders with open long or short positions will pay each other a funding payment, depending on market conditions. This incentivizes more traders to take the unpopular side of the trade. Specifically, four routines, i.e., `_addLiquidity()`, `_removeLiquidity()`, `_withdrawMargin()`, and `trade()` update the cumulative funding rate. In the `_withdrawMargin()` and `trade()` routines, the funding fee of the trader will be calculated. Conversely, `_addLiquidity()` and `_removeLiquidity()` routines only call `_updateCumuFundingRate()` after updating `_liquidity`. We notice an issue that attackers may make profits by sending multiple transactions in a block to manipulate funding rate and trade.

In particular, we show the related code snippet below. On its entry of `_updateCumuFundingRate()`, there is a check, i.e., `if (block.number > _cumuFundingRateBlock)`. It ensures that `_cumuFundingRate` can only be modified once in a block. Furthermore, the result of `rate` is influenced by `_tradersNetVolume`, `price`, and `_liquidity` (line 682). In the `_addLiquidity()` routine, `_liquidity` is updated before `_updateCumuFundingRate()` (lines 575-577). Therefore, a malicious liquidity provider could intentionally manipulate the funding rate, and the other `trade()` transactions in the block can make profits with that funding rate.

```
678     function _updateCumuFundingRate(uint256 price) private {
679         if (block.number > _cumuFundingRateBlock) {
680             int256 rate;
```

```
            if (_liquidity != 0) {
                rate = _tradersNetVolume.mul(price).mul(_multiplier).mul(_fundingRateCoefficient).div(_liquidity);
            } else {
                rate = 0;
            }
            int256 delta = rate * (int256(block.number.sub(_cumuFundingRateBlock)));
            _cumuFundingRate += delta; // overflow is intended
            _cumuFundingRateBlock = block.number;
        }
    }
```

Listing 3.1: PerpetualPool::_updateCumuFundingRate()

```
    function _addLiquidity(uint256 bAmount) internal _lock_ {
        require(bAmount >= _minAddLiquidity, "PerpetualPool: add liquidity less than minimum requirement");
        require(bAmount.reformat(_bDecimals) == bAmount, "PerpetualPool: _addLiquidity bAmount not valid");

        _bToken.safeTransferFrom(msg.sender, address(this), bAmount.rescale(_bDecimals));

        uint256 poolDynamicEquity = _liquidity.add(_tradersNetCost.sub(_tradersNetVolume.mul(_price).mul(_multiplier)));
        uint256 totalSupply = _lToken.totalSupply();
        uint256 lShares;
        if (totalSupply == 0) {
            lShares = bAmount;
        } else {
            lShares = bAmount.mul(totalSupply).div(poolDynamicEquity);
        }

        _lToken.mint(msg.sender, lShares);
        _liquidity = _liquidity.add(bAmount);

        _updateCumuFundingRate(_price);

        emit AddLiquidity(msg.sender, lShares, bAmount);
    }
```

Listing 3.2: PerpetualPool::_addLiquidity()

Recommendation Update `_CumuFundingRate` timely in `_addLiquidity()` and `_removeLiquidity()`.

Status The issue has been fixed by this commit: `c3041af`

3.2 Sandwiched Liquidation To Bypass isQualifiedLiquidator() Check

- ID: PVE-002
- Severity: Low
- Likelihood: medium
- Impact: Low
- Target: `LiquidatorQualifier`
- Category: Time and State [4]
- CWE subcategory: CWE-682 [2]

Description

In the `Deri` protocol, if a trader's margin ratio falls below the maintenance margin ratio, the trader will get liquidated by liquidators. In this section, we examine the `isQualifiedLiquidator()` routine. The function check if the parameter, `liquidator`, is qualified as a liquidator. In other words, it ensures the `liquidator`'s deposited stake token is above or equal to the average amount of deposits. However, this check can be easily bypassed by sandwiching. In particular, we show the related code snippet below. A liquidator can send three internal transactions in the following order:

- The first one makes a deposit so that the balance is not less than the average.
- The second one liquidates under-water positions to gain liquidation rewards.
- The last one withdraws the balance.

```
    function deposit(uint256 amount) public {
        require(amount != 0, "LiquidatorQualifier: deposit 0 stake tokens");
        IERC20(stakeTokenAddress).safeTransferFrom(msg.sender, address(this), amount);

        totalStakedTokens = totalStakedTokens.add(amount);
        if (stakes[msg.sender] == 0) {
            totalStakers = totalStakers.add(1);
        }
        stakes[msg.sender] = stakes[msg.sender].add(amount);
    }
```

Listing 3.3: LiquidatorQualifier :: deposit()

```
    function withdraw(uint256 amount) public {
        require(amount != 0, "LiquidatorQualifier: withdraw 0 stake tokens");
        require(amount <= stakes[msg.sender], "LiquidatorQualifier: withdraw amount
            exceeds staked");

        totalStakedTokens = totalStakedTokens.sub(amount);
        if (stakes[msg.sender] == amount) {
            totalStakers = totalStakers.sub(1);
```

```
        }
        stakes[msg.sender] = stakes[msg.sender].sub(amount);

        IERC20(stakeTokenAddress).safeTransfer(msg.sender, amount);
    }
```

Listing 3.4: LiquidatorQualifier ::withdraw()

```
    function isQualifiedLiquidator(address liquidator) public view override returns (
        bool) {
        if (totalStakers == 0) {
            return false;
        }
        return stakes[liquidator] >= totalStakedTokens / totalStakers;
    }
```

Listing 3.5: LiquidatorQualifier :: isQualifiedLiquidator ()

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of other liquidators.

Status This issue has been confirmed. The team further clarifies an interesting point that the liquidation is generally encouraged in the `Deri` protocol and any liquidation transaction is strengthening the protocol's functioning and security. Being said, this issue could be still in favor of the protocol's liquidity providers and traders and therefore is considered welcome!

3.3 Potential Replay Of Signed Prices

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PerpetualPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the current implementation of the Deri protocol, the `_price` is provided by a trusted oracle. If the oracle is an on-chain contract, `PerpetualPool` can get a price by the oracle's interface. Conversely, if the oracle is an EOA account, `PerpetualPool` should check the price signature first to verify if the price is authorized. We notice that the price signature is signed with a `timestamp`. However, if the signature is not signed in the currently active chain, e.g., a testnet. It could lead to a replay attack. In particular, we show the related code snippet below.

```
function _updatePriceWithSignature(
    uint256 timestamp, uint256 price, uint8 v, bytes32 r, bytes32 s
) internal
{
    if (block.number != _lastPriceBlockNumber) {
        require(timestamp >= _lastPriceTimestamp, "PerpetualPool: price is not the
            newest");
        require(block.timestamp - timestamp <= _priceDelayAllowance, "PerpetualPool:
            price is older than allowance");

        _checkPriceSignature(timestamp, price, v, r, s);

        _price = price;
        _lastPriceTimestamp = timestamp;
        _lastPriceBlockNumber = block.number;
    }
}
```

Listing 3.6: PerpetualPool::_updatePriceWithSignature()

Recommendation Add new fields into signature calculation, i.e., `chainID` and the address of `PerpetualPool`.

Status After detailed discussions, the team has informed us that this is part of design and the above replay is expected and accepted. Therefore, we agree with the team there is no need to address it.

3.4 Potential Front-Running For setPool()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LToken`
- Category: Time and State [4]
- CWE subcategory: CWE-682 [2]

Description

In the Deri protocol, there are two token contracts named `LToken` and `PToken`. `LToken` is a liquidity provider token implementation and `PToken` is a non-fungible position token implementation. Each of these two token contracts implements the same routine, `setPool()`. As the name indicates, it set the address of `PerpetualPool` for token contracts. When the `PerpetualPool` migrates to a new address, the `LToken` and `PToken` will update the `_pool`. However, there is a front-running issue in the `setPool()` routine. Specifically, after deploying the two token contracts, anyone can call the `setPool()` function to set an arbitrary pool address to the token. In the following, we show the related code snippet as below.

```
    /**
     * @dev Initializes the contract by setting a `name` and a `symbol` to the token
     */
    constructor(string memory name_, string memory symbol_) ERC20(name_, symbol_) {}

    /**
     * @dev See {ILToken}.{setPool}
     */
    function setPool(address newPool) public override {
        require(newPool != address(0), "LToken: setPool to 0 address");
        require(
            _pool == address(0)   msg.sender == _pool,
            "LToken: setPool caller is not current pool"
        );
        _pool = newPool;
    }
```

Listing 3.7: LToken::setPool()

Recommendation Setting a pool address to the token in initialization.

```
     constructor(string memory name_, string memory symbol_, address pool_) ERC20(name_,
         symbol_) {
         require(pool_ != address(0), "LToken: construct with 0 address pool");
         _pool = pool_;
     }
```

Listing 3.8: LToken::constructor()

Status The issue has been fixed by this commit: `c3041af`

3.5 Lack Of Sanity Checks For System Parameters

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PerpetualPool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Deri protocol is no exception. Specifically, if we examine the `PerpetualPool` contract, it has defined a number of system-wide risk parameters, e.g., `_redemptionFeeRatio`, `_minMaintenanceMarginRatio`, and `_minLiquidationReward`. In the following, we show the `initialize()` routine that sets up these parameters.

```
    function initialize(
        string memory symbol_,
        address[5] calldata addresses_,
        uint256[12] calldata parameters_
    ) public override {
        require(bytes(_symbol).length == 0 && _controller == address(0), "PerpetualPool: 
            already initialized");

        _controller = msg.sender;
        _symbol = symbol_;

        _bToken = IERC20(addresses_[0]);
        _bDecimals = _bToken.decimals();
        _pToken = IPToken(addresses_[1]);
        _lToken = ILToken(addresses_[2]);
        _oracle = IOracle(addresses_[3]);
        _isContractOracle = _isContract(address(_oracle));
        _liquidatorQualifier = ILiquidatorQualifier(addresses_[4]);

        _multiplier = parameters_[0];
        _feeRatio = parameters_[1];
        _minPoolMarginRatio = parameters_[2];
        _minInitialMarginRatio = parameters_[3];
        _minMaintenanceMarginRatio = parameters_[4];
        _minAddLiquidity = parameters_[5];
        _redemptionFeeRatio = parameters_[6];
        _fundingRateCoefficient = parameters_[7];
        _minLiquidationReward = parameters_[8];
        _maxLiquidationReward = parameters_[9];
        _liquidationCutRatio = parameters_[10];
        _priceDelayAllowance = parameters_[11];
    }
```

Listing 3.9: PerpetualPool::initialize()

This parameter defines an important aspect of the protocol operation and needs to exercise extra care when configuring or updating it. Our analysis shows the configuration logic on it can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to undesirable consequences. For example, an unlikely misconfiguration of `_minMaintenanceMarginRatio` may prevent liquidation. (line 618).

```
613     function _liquidate(address owner, uint256 timestamp, uint256 price) internal _lock_
            {
614         (int256 volume, int256 cost, , uint256 margin, uint256 lastUpdateTimestamp) =
                _pToken.getPosition(owner);
615         require(timestamp > lastUpdateTimestamp, "PerpetualPool: liquidate price is
                before position timestamp");
```

```
        int256 pnl = volume.mul(price).mul(_multiplier).sub(cost);
        require(pnl.add(margin) <= 0   _calculateMarginRatio(volume, cost, price, margin)
            < _minMaintenanceMarginRatio, "PerpetualPool: cannot liquidate");

        _liquidity = _liquidity.add(margin);
        _tradersNetVolume = _tradersNetVolume.sub(volume);
        _tradersNetCost = _tradersNetCost.sub(cost);
        _pToken.update(owner, 0, 0, 0, 0, 0);
```

Listing 3.10: PerpetualPool::_liquidate()

Recommendation Validate any changes regarding the system-wide parameters to ensure the changes fall in an appropriate range.

Status The issue has been confirmed.

3.6 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PerpetualPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

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

```
    function _depositMargin(uint256 bAmount) internal _lock_ {
        require(bAmount != 0, "PerpetualPool: deposit zero margin");
        require(bAmount.reformat(_bDecimals) == bAmount, "PerpetualPool: _depositMargin
            bAmount not valid");

        _bToken.safeTransferFrom(msg.sender, address(this), bAmount.rescale(_bDecimals))
            ;
        if (!_pToken.exists(msg.sender)) {
            _pToken.mint(msg.sender, bAmount);
        } else {
            (int256 volume, int256 cost, int256 lastCumuFundingRate, uint256 margin,) =
                _pToken.getPosition(msg.sender);
```

```
            margin = margin.add(bAmount);
            _pToken.update(msg.sender, volume, cost, lastCumuFundingRate, margin, block.
                timestamp);
        }
        emit DepositMargin(msg.sender, bAmount);
    }
```

Listing 3.11: PerpetualPool::_depositMargin()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

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

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

Status The issue has been fixed by this commit: c3041af

4 Conclusion

In this audit, we have analyzed the `Deri` protocol design and implementation. The `Deri` protocol is a decentralized protocol for users to exchange risk exposures precisely and capital-efficiently. 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] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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