SMART CONTRACT AUDIT REPORT

for

Deri Protocol V2

Prepared By: Shuxiao Wang

PeckShield

May 24, 2021

Document Properties

Client	Deri Protocol V2
Title	Smart Contract Audit Report
Target	Deri-V2
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 24, 2021	Xuxian Jiang	Final Release
1.0-rc1	May 25, 2021	Xuxian Jiang	Release Candidate #1
0.2	May 24, 2021	Xuxian Jiang	Additional Findings
0.1	May 14, 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 **4**

1.1 About Deri-V2 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 6

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Replays Against SymbolOracleOffChain::updatePrice() 11

3.2 Invariant Enforcement in removeLiquidity()/trade() 12

3.3 Proper Protocol Fee Accounting in removeMargin() 14

3.4 Potential DoS Against setPool() in LToken/PToken 15

3.5 Incomplete Migration Support In executePoolMigration() 16

3.6 Trust Issue of Admin Keys 17

3.7 Potential Front-Running/MEV With Reduced Return 18

3.8 Potential Manipulation of BToken Prices 20

3.9 Improved Sanity Checks For System Parameters 21

4 Conclusion **24**

References **25**

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-V2

`Deri` 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`, risk exposures are tokenized as non-fungible tokens so that they can be imported into other DeFi projects for their own financial purposes. The audited `Deri-V2` protocol inherits all the features of V1 and further supports several key new features, such as dynamic mixed margin and liquidity-providing. These new features support multiple tokens as base tokens for liquidity providers to provide as liquidity and for traders to deposit as margin. By doing so, the derivative trading can achieve an optimal capital efficiency, which is potentially higher than that of centralized exchanges.

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

Table 1.1: Basic Information of The `Deri` Protocol

Item	Description
Name	Deri Protocol V2
Website	https://deri.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 24, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Deri-V2 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/dfactory-tech/deriprotocol-v2.git (78dc021c)

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-v2.git (9486ace)

1.2 About PeckShield

PeckShield Inc. [12] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

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

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

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

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

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.3: The Full List of Check Items

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

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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Deri-V2 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	4	■■■■
Low	4	■■■■
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 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 4 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Deri-V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Replays Against SymbolOracleOffChain::updatePrice()	Coding Practices	Resolved
PVE-002	Low	Invariant Enforcement in removeLiquidity()/trade()	Business Logic	Resolved
PVE-003	Medium	Proper Protocol Fee Accounting in removeMargin()	Business Logic	Fixed
PVE-004	Low	Potential DoS Against setPool() in LToken/PToken	Time and State	Resolved
PVE-005	Low	Incomplete Migration Support In executePoolMigration()	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Medium	Potential Front-Running/MEV With Reduced Return	Time and State	Fixed
PVE-008	Medium	Potential Manipulation of BToken Prices	Business Logic	Confirmed
PVE-009	Low	Improved Sanity Checks For System Parameters	Coding Practices	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 Replays Against SymbolOracleOffChain::updatePrice()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SymbolOracleOffChain`
- Category: Coding Practices [7]
- CWE subcategory: CWE-841 [4]

Description

The `Deri-V2` protocol allows the authorized `signatory` to update oracle price using off-chain signed price. Our analysis shows that the related `updatePrice()` routine generates the message hash using required fields, i.e., `symbol`, `timestamp_`, and `price_`. However, this calculation leads to the same generated signature for different blockchains. The absence of a `EIP-712 domain separator` with the `EIP-155 chainID` in current calculation makes signature validation susceptible to possible replays across different chains.

```
    // update oracle price using off chain signed price
    // the signature must be verified in order for the price to be updated
    function updatePrice(uint256 timestamp_, uint256 price_, uint8 v_, bytes32 r_,
        bytes32 s_) public override {
        uint256 lastTimestamp = timestamp;
        if (timestamp_ > lastTimestamp) {
            if (v_ == 27  v_ == 28) {
                bytes32 message = keccak256(abi.encodePacked(symbol, timestamp_, price_)
                    );
                bytes32 hash = keccak256(abi.encodePacked('\x19Ethereum Signed Message:\
                    n32', message));
                address signer = ecrecover(hash, v_, r_, s_);
                if (signer == signatory) {
                    timestamp = timestamp_;
                    price = price_;
                }
            }
        }
```

```
42      }
```

Listing 3.1: SymbolOracleOffChain::updatePrice()

Recommendation Add the `EIP-712 domain separator` with the `chainID` into calculation.

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.

3.2 Invariant Enforcement in removeLiquidity()/trade()

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `PerpetualPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

In the `Deri-V2` protocol, there are a number of protocol-wide invariants, e.g., `_minBToken0Ratio` and `_minMaintenanceMarginRatio`. For example, if a trader's margin ratio falls below the maintenance margin ratio (`_minMaintenanceMarginRatio`), the trader will get liquidated by liquidators. Also, for any liquidity changes, there is a need to ensure the `BToken0` ratio is no less than `_minBToken0Ratio`. In this section, we examine the `_minBToken0Ratio` invariant enforcement.

```
538     function _getBToken0Ratio(int256 totalDynamicEquity, int256[] memory dynamicEquities
            ) internal pure returns (int256) {
539         return totalDynamicEquity == 0 ? type(int256).max : dynamicEquities[0] * ONE /
                totalDynamicEquity;
540     }
```

Listing 3.2: PerpetualPool::_getBToken0Ratio()

There are a number of liquidity-changing functions: `addliquidity()`, `removeLiquidity()`, `addMargin()`, `removeMargin()`, `trade()`, and `liquidate()`. Currently, the `_minBToken0Ratio` invariant is validated only in the `addliquidity()` routine (line 248). There is also a need to enforce the invariant in all other liquidity-changing routines.

```
218     function addLiquidity(address owner, uint256 bTokenId, uint256 bAmount, uint256
            blength, uint256 slength) public override _router_ _lock_ {
219         ILToken lToken = ILToken(_lTokenAddress);
220         if(!lToken.exists(owner)) lToken.mint(owner);

222         _updateBTokenPrice(bTokenId);
223         _updatePricesAndDistributePnl(blength, slength);
```

```
        BTokenInfo storage b = _bTokens[bTokenId];
        bAmount = _deflationCompatibleSafeTransferFrom(b.bTokenAddress, b.decimals, owner, address(this), bAmount);

        int256 cumulativePnl = b.cumulativePnl;
        ILToken.Asset memory asset = lToken.getAsset(owner, bTokenId);

        int256 delta; // owner's liquidity change amount for bTokenId
        int256 pnl = (cumulativePnl - asset.lastCumulativePnl) * asset.liquidity / ONE; // owner's pnl as LP since last settlement
        if (bTokenId == 0) {
            delta = bAmount.utoi() + pnl.reformat(_decimals0);
            b.pnl -= pnl; // this pnl comes from b.pnl, thus should be deducted from b.pnl
            _protocolFeeAccrued += pnl - pnl.reformat(_decimals0); // deal with accuracy tail
        } else {
            delta = bAmount.utoi();
            asset.pnl += pnl;
        }
        asset.liquidity += delta;
        asset.lastCumulativePnl = cumulativePnl;
        b.liquidity += delta;

        lToken.updateAsset(owner, bTokenId, asset);

        (int256 totalDynamicEquity, int256[] memory dynamicEquities) = _getBTokenDynamicEquities(blength);
        require(_getBToken0Ratio(totalDynamicEquity, dynamicEquities) >= _minBToken0Ratio, "insuf't b0");

        emit AddLiquidity(owner, bTokenId, bAmount);
    }
```

Listing 3.3: PerpetualPool::addLiquidity()

Recommendation Enforce the `_minBToken0Ratio` invariant in all liquidity-changing routines.

Status This issue has been confirmed. And the team clarifies that, by design, this `_minBToken0Ratio` invariant does not need to strictly enforced in `removeLiquidity()` and `trade()`.

3.3 Proper Protocol Fee Accounting in removeMargin()

- ID: PVE-003
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `PerpetualPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The `Deri-V2` protocol is designed to collect necessary protocol fee, which is accumulated in a storage variable `_protocolFeeAccrued`. The accrued protocol fee is denominated at the `BToken0` and will be collected in a number of routines `addLiquidity()`, `removeLiquidity()`, `trade()`, and `removeMargin()`.

However, our analysis of the protocol fee shows the `removeMargin()` routine has an inconsistent logic in accruing the protocol fee. To elaborate, we show below its implementation.

```
    function removeMargin(address owner, uint256 bTokenId, uint256 bAmount, uint256 blength, uint256 slength) public override _router_ _lock_ {
        _updatePricesAndDistributePnl(blength, slength);
        _settleTraderFundingFee(owner, slength);
        _coverTraderDebt(owner, blength);

        IPToken pToken = IPToken(_pTokenAddress);
        BTokenInfo storage b = _bTokens[bTokenId];
        uint256 decimals = b.decimals;
        bAmount = bAmount.reformat(decimals);

        int256 amount = bAmount.utoi();
        int256 margin = pToken.getMargin(owner, bTokenId);

        if (amount >= margin) {
            bAmount = margin.itou();
            _protocolFeeAccrued += margin - margin.reformat(_decimals0); // deal with accuracy tail
            margin = 0;
        } else {
            margin -= amount;
        }
        pToken.updateMargin(owner, bTokenId, margin);

        require(_getTraderMarginRatio(owner, blength, slength) >= _minInitialMarginRatio, "insuf't margin");

        IERC20(b.bTokenAddress).safeTransfer(owner, bAmount.rescale(18, decimals));
        emit RemoveMargin(owner, bTokenId, bAmount);
    }
```

Listing 3.4: PerpetualPool::removeMargin()

The inconsistency comes form the margin removal of a non-`BToken0` asset. Notice the intended margin for removal may not be the protocol-wide `BToken0`. However, the protocol fee accrued at line 339 is denomiated at the intended margin for removal, hence introducing the inconsistency: `_protocolFeeAccrued += margin - margin.reformat(_decimals0)`.

Recommendation Be consistent in calculating the protocol fee in all affected routines.

Status This issue has been fixed in the following commit: `9486ace`.

3.4 Potential DoS Against setPool() in LToken/PToken

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LToken`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

In the `Deri-V2` protocol, there are two token contracts named `LToken` and `PToken`. The `LToken` contract is a liquidity provider token implementation and the `PToken` contract is a non-fungible position token implementation. Each of these two token contracts implements a common routine `setPool()`. As the name indicates, it sets the address of `PerpetualPool` for token contracts. When the `PerpetualPool` migrates to a new address, the `LToken` and `PToken` contract 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.

```
function setPool(address newPool) public override {
    require(_pool == address(0)  _pool == msg.sender, 'LToken.setPool: not allowed');
    _pool = newPool;
}
```

Listing 3.5: LToken::setPool()

Recommendation Authenticate the caller when the pool address is being initialized.

Status This issue has been confirmed and the team plans to take a guarded launch to properly initialize the above contracts.

3.5 Incomplete Migration Support In executePoolMigration()

- ID: PVE-005
- Severity: Low
- Likelihood: N/A
- Impact: N/A
- Target: PerpetualPool
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The Deri-V2 protocol has the support of migrating the pool to a new one. And the migration logic is mainly implemented in two functions, i.e., approveMigration() and executeMigration(). The first function is designed to approve the new pool to transfer the pool assets while the second function actually executes the migration operation. While examining these two functions in PerpetualPool, we notice the function body of the second function is currently commented out and has an incomplete implementation.

```
    // during a migration, this function is intended to be called in the target pool
    function executePoolMigration(address sourcePool) public override _router_ {
        // (uint256 blength, uint256 slength) = IPerpetualPool(sourcePool).getLengths();
        // for (uint256 i = 0; i < blength; i++) {
        //     BTokenInfo memory b = IPerpetualPool(sourcePool).getBToken(i);
        //     IERC20(b.bTokenAddress).safeTransferFrom(sourcePool, address(this),
            IERC20(b.bTokenAddress).balanceOf(sourcePool));
        //     _bTokens.push(b);
        // }
        // for (uint256 i = 0; i < slength; i++) {
        //     _symbols.push(IPerpetualPool(sourcePool).getSymbol(i));
        // }
        // _protocolFeeAccrued = IPerpetualPool(sourcePool).getProtocolFeeAccrued();
    }
```

Listing 3.6: PerpetualPool::executePoolMigration()

Recommendation Complete the proper implementation of executePoolMigration().

Status After detailed discussions, the team has informed us that this is intended. The reason is that the deployed pool is a new one, which does not need to migrate any assets from an old pool.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PerpetualPoolRouter`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Deri-V2` protocol, there is a special administrative account, i.e., `controller`. This `controller` account plays a critical role in governing and regulating the system-wide operations (e.g., authorizing other roles, setting various parameters, and adjusting external oracles). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `controller` account is indeed privileged. In the following, we show representative privileged operations in the `Deri-V2` protocol.

```
    function setBTokenParameters(
        uint256 bTokenId,
        address swapperAddress,
        address oracleAddress,
        uint256 discount
    )
        public override _controller_
    {
        IPerpetualPool(_pool).setBTokenParameters(bTokenId, swapperAddress,
            oracleAddress, discount);
    }

    function setSymbolParameters(
        uint256 symbolId,
        address oracleAddress,
        uint256 feeRatio,
        uint256 fundingRateCoefficient
    )
        public override _controller_
    {
        IPerpetualPool(_pool).setSymbolParameters(symbolId, oracleAddress, feeRatio,
            fundingRateCoefficient);
    }
```

Listing 3.7: Various Setters in PerpetualPoolRouter

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the `controller` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that it is currently managed by a multi-sig account. We point out that a compromised controller account would allow the attacker to undermine necessary assumptions behind the protocol and subvert various protocol operations.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and partially mitigated with a multi-sig controller account.

3.7 Potential Front-Running/MEV With Reduced Return

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: LendingStrategyUSDT
- Category: Time and State [9]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the Deri-V2 protocol supports multiple tokens as base tokens for liquidity providers to provide as liquidity and for traders to deposit as margin. Because of that, there is a constant need of swapping one asset to another. With that, the protocol has provided two helper routines to facilitate the asset conversion: _swapExactTokensForTokens() and _swapTokensForExactTokens()

```
    // low-level swap function
    function _swapExactTokensForTokens(address a, address b, address to) internal
        override {
        address[] memory path = new address[](2);
        path[0] = a;
        path[1] = b;

        IUniswapV2Router02(router).swapExactTokensForTokens(
            IERC20(a).balanceOf(address(this)),
            0,
            path,
            to,
            block.timestamp + 3600
        );
    }

    // low-level swap function
    function _swapTokensForExactTokens(address a, address b, uint256 amount, address to)
         internal override {
```

```
        address[] memory path = new address[](2);
        path[0] = a;
        path[1] = b;

        IUniswapV2Router02(router).swapTokensForExactTokens(
            amount,
            IERC20(a).balanceOf(address(this)),
            path,
            to,
            block.timestamp + 3600
        );
    }
```

Listing 3.8: BTokenSwapper1::_swapExactTokensForTokens()/_swapTokensForExactTokens()

To elaborate, we show above these two helper routines. We notice the conversion is routed to `UniswapV2` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commit: `9486ace`.

3.8 Potential Manipulation of BToken Prices

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BTokenOracle1`, `BTokenOracle2`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

In the `Deri-V2` protocol, each base token has a price oracle that is based on the widely used `UniswapV2` `time-weighted average price (TWAP)`. The `TWAP` is constructed by reading the cumulative price from a `UniswapV2` pair at the beginning and at the end of the desired interval. The difference in this cumulative price can then be divided by the length of the interval to create a `TWAP` for that period.

To elaborate, we show below the `getPrice()` implementation. It comes to our attention that the interval used to compute the `TWAP` is not restricted (line 58). As a result, it leaves the room or possibility for undesired price manipulation. To mitigate, it is helpful to ensure a minimum interval for the `TWAP`-based price calculation.

```
39      function getPrice() public override returns (uint256) {
40          IUniswapV2Pair p = IUniswapV2Pair(pair);
41          uint256 reserveQ;
42          uint256 reserveB;
43          uint256 timestamp;

45          if (isQuoteToken0) {(reserveQ, reserveB, timestamp) = p.getReserves();
46          } else { (reserveB, reserveQ, timestamp) = p.getReserves();}

48          if (timestamp != timestampLast2) {
49              priceCumulativeLast1 = priceCumulativeLast2;
50              timestampLast1 = timestampLast2;
51              priceCumulativeLast2 = isQuoteToken0 ? p.price0CumulativeLast() : p.
                    price1CumulativeLast();
52              timestampLast2 = timestamp;
53          }

55          uint256 price;
56          if (timestampLast1 != 0) {
57              // TWAP
58              price = (priceCumulativeLast2 - priceCumulativeLast1) / (timestampLast2 -
                    timestampLast1) * 10**(18 + qDecimals - bDecimals) / Q112;
59          } else {
60              // Spot
61              // this price will only be used when BToken is newly added to pool
62              // since the liquidity for newly added BToken is always zero,
63              // there will be no manipulation consequences for this price
64              price = reserveB * 10**(18 + qDecimals - bDecimals) / reserveQ;
```

```
        }

        return price;
    }
```

Listing 3.9: BTokenOracle1::getPrice()

Recommendation Develop an effective mitigation to avoid the price oracle from being manipulated.

Status This issue has been confirmed. The discussion with the team shows the cost of manipulating inter-block `TWAP` is significantly higher than internal-transaction-level flashloans. And even such a high-cost manipulation could only affect the weights of the `BTokens` for the `PnL` distribution during the manipulated period, which is an insignificant consequence. Such manipulations may not benefit the manipulator.

3.9 Improved Sanity Checks For System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PerpetualPoolRouter`
- Category: Coding Practices [7]
- 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-V2` protocol is no exception. Specifically, if we examine the `PerpetualPool` contract, it has defined a number of system-wide risk parameters, e.g., `_minBToken0Ratio`, `_minPoolMarginRatio`, and `_liquidationCutRatio`. In the following, we show the representative `setter` routines that allow for their update.

```
    function addSymbol(SymbolInfo memory info) public override _router_ {
        _symbols.push(info);
        IPToken(_pTokenAddress).setNumSymbols(_symbols.length);
    }

    function setBTokenParameters(uint256 bTokenId, address swapperAddress, address
        oracleAddress, uint256 discount) public override _router_ {
        BTokenInfo storage b = _bTokens[bTokenId];
        b.swapperAddress = swapperAddress;
        if (bTokenId != 0) {
            IERC20(_bTokens[0].bTokenAddress).safeApprove(swapperAddress, type(uint256).
                max);
```

```
            IERC20(_bTokens[bTokenId].bTokenAddress).safeApprove(swapperAddress, type(uint256).max);
        }
        b.oracleAddress = oracleAddress;
        b.discount = int256(discount);
    }

    function setSymbolParameters(uint256 symbolId, address oracleAddress, uint256 feeRatio, uint256 fundingRateCoefficient) public override _router_ {
        SymbolInfo storage s = _symbols[symbolId];
        s.oracleAddress = oracleAddress;
        s.feeRatio = int256(feeRatio);
        s.fundingRateCoefficient = int256(fundingRateCoefficient);
    }
```

Listing 3.10: Example Setters in PerpetualPool

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 mis-configuration of `fundingRateCoefficient` may prevent the symbol price from being updated (line 525).

```
516     function _updateSymbolPrices(int256 totalDynamicEquity, uint256 slength) internal returns (int256) {
517         if (totalDynamicEquity <= 0) return 0;
518         int256 undistributedPnl;
519         for (uint256 i = 0; i < slength; i++) {
520             SymbolInfo storage s = _symbols[i];
521             int256 price = IOracle(s.oracleAddress).getPrice().utoi();
522             int256 tradersNetVolume = s.tradersNetVolume;
523             if (tradersNetVolume != 0) {
524                 int256 multiplier = s.multiplier;
525                 int256 ratePerBlock = tradersNetVolume * price / ONE * price / ONE * multiplier / ONE * multiplier / ONE * s.fundingRateCoefficient / totalDynamicEquity;
526                 int256 delta = ratePerBlock * int256(block.number - _lastUpdateBlock);

528                 undistributedPnl += tradersNetVolume * delta / ONE;
529                 undistributedPnl -= tradersNetVolume * (price - s.price) / ONE * multiplier / ONE;

531                 unchecked { s.cumulativeFundingRate += delta; }
532             }
533             s.price = price;
534         }
535         return undistributedPnl;
536     }
```

Listing 3.11: PerpetualPool::_updateSymbolPrices()

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

Status The issue has been confirmed. And the team plans to exercise extra caution when configuring these risk parameters.

4 Conclusion

In this audit, we have analyzed the `Deri-V2` 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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