SMART CONTRACT AUDIT REPORT

for

Deri Protocol V2 - EverlastingOption

Prepared By: Yiqun Chen

PeckShield

August 9, 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	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 9, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 31, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

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. The new EverlastingOption supports multiple symbols. Note that the V2 version allows for multiple base tokens so that liquidity providers can provide as liquidity and traders can 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	August 9, 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 (a7a03c0)

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 (71fbc98)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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) [5], 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 implementation of the new `EverlastingOption`. 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	1	■
Informational	1	■
Total	3	

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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Deri-V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Suggested Event Generation in Controller Changes	Coding Practices	Resolved
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Suggested Event Generation in Controller Changes

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Ownable`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `Ownable` contract as an example. This contract is designed to manage the ownership surrounding the current `controller`. While examining the events that reflect the `controller` dynamics, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `controller` is being changed, there is no respective event being emitted to reflect the transfer of `controller` (line 23).

```
1461    function setNewController(address newController) public override _controller_ {
1462        _newController = newController;
1463    }
```

Listing 3.1: `Ownable::setNewController()`

Recommendation Properly emit the `NewController` event when the controller is being transferred.

Status The issue has been fixed by this commit: `71fbc98`.

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BTokenSwapper
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

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.

```
    // swap exact `amountB0` amount of tokenB0 for tokenBX
    function swapExactB0ForBX(uint256 amountB0, uint256 referencePrice) external
        override returns (uint256 resultB0, uint256 resultBX) {
        address caller = msg.sender;

        IERC20 tokenB0 = IERC20(addressB0);
        IERC20 tokenBX = IERC20(addressBX);

        uint256 bx1 = tokenBX.balanceOf(caller);
        amountB0 = amountB0.rescale(18, decimalsB0);

        if (amountB0 == 0) {
            return (0, 0);
        }

        tokenB0.safeTransferFrom(caller, address(this), amountB0);
        _swapExactTokensForTokens(addressB0, addressBX, caller);
        uint256 bx2 = tokenBX.balanceOf(caller);

        resultB0 = amountB0.rescale(decimalsB0, 18);
        resultBX = (bx2 - bx1).rescale(decimalsBX, 18);

        require(
            resultBX >= resultB0 * (UONE - maxSlippageRatio) / referencePrice,
            'BTokenSwapper.swapExactB0ForBX: slippage exceeds allowance'
        );
    }
```

Listing 3.2: BTokenSwapper::swapExactB0ForBX()

In particular, if we examine the `BTokenSwapper::swapExactB0ForBX()` routine, this routine is designed to swap a given amount of `tokenB0` for `tokenBX`. We notice the slippage control enforcement is achieved by satisfying `require(resultBX >= resultB0 * (UONE - maxSlippageRatio)/ referencePrice)` (lines $56 - 59$). The enforcement is performed with mixed multiplication and devision. For improved precision, it is better to calculate the equation without involving the division, i.e., `require(resultBX * referencePrice >= resultB0 * (UONE - maxSlippageRatio))`. Note 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.

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

Status The issue has been fixed by this commit: `71fbc98`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- 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 a representative privileged operation in the `Deri-V2` protocol.

```
    function executeMigration(address source) external override _controller_ {
        uint256 migrationTimestamp_ = IEverlastingOption(source).migrationTimestamp();
        address migrationDestination_ = IEverlastingOption(source).migrationDestination
            ();
        require(migrationTimestamp_ != 0 && block.timestamp >= migrationTimestamp_, '
            time inv');
        require(migrationDestination_ == address(this), 'not dest');

        // transfer bToken to this address
        IERC20(_bTokenAddress).safeTransferFrom(source, address(this), IERC20(
            _bTokenAddress).balanceOf(source));
```

```
        // transfer symbol infos
        uint256[] memory symbolIds = IPTokenOption(_pTokenAddress).getActiveSymbolIds();
        for (uint256 i = 0; i < symbolIds.length; i++) {
            _symbols[symbolIds[i]] = IEverlastingOption(source).getSymbol(symbolIds[i]);
        }

        // transfer state values
        _liquidity = IEverlastingOption(source).getLiquidity();
        _lastTimestamp = IEverlastingOption(source).getLastTimestamp();
        _protocolFeeAccrued = IEverlastingOption(source).getProtocolFeeAccrued();

        emit ExecuteMigration(migrationTimestamp_, source, migrationDestination_);
    }
```

Listing 3.3: EverlastingOption::executeMigration()

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 with necessary timelock enforcement. 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 time-locked, multi-sig `controller` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of `EverlastingOption` in the `Deri-V2` protocol. 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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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