



SMART CONTRACT AUDIT REPORT

for

Deri Protocol V3



Prepared By: Yiqun Chen

PeckShield

December 26, 2021

Document Properties

Client	Deri Protocol V3
Title	Smart Contract Audit Report
Target	Deri-V3
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	December 26, 2021	Xuxian Jiang	Final Release

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	4
1.1	About Deri-V3	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	Trust Issue of Admin Keys	11
3.2	Removal of Unused State/Code	13
3.3	Meaningful Events For Important States Change	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Deri-V3` 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-V3

`Deri` is a decentralized protocol for users to exchange risk exposures precisely and capital-efficiently. The audited `Deri-V3` protocol is upgraded from V2 with the consistent and scalable support of perpetual futures and everlasting options. The `Deri-V3` protocol also introduces a core feature [external custody](#), so that the user capital is held in an external money market protocol instead of internal liquidity pools. This new mechanism supports multiple base tokens with substantially higher scalability and capital efficiency.

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

Table 1.1: Basic Information of The `Deri` Protocol

Item	Description
Name	Deri Protocol V3
Website	https://deri.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 26, 2021

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

in this audit. Note that Deri-V3 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-v3.git> (b4458d5)

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-v3.git> (a12346b)

1.2 About PeckShield

PeckShield Inc. [8] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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) [6], 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-v3` 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	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-V3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-002	Informational	Removal of Unused State/Code	Coding Practices	Resolved
PVE-003	Low	Meaningful Events For Important States Change	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 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Deri-V3 protocol, there is a special administrative account, i.e., `admin`, which 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 `admin` account is indeed privileged. In the following, we show representative privileged operations in the Deri-V3 protocol.

```
124     function addMarket(address market) external _onlyAdmin_ {
125         address underlying = IVToken(market).underlying();
126         require(
127             IERC20(underlying).decimals() == 18,
128             'PoolImplementation.addMarket: only token of decimals 18'
129         );
130         require(
131             IVToken(market).isVToken(),
132             'PoolImplementation.addMarket: invalid vToken'
133         );
134         require(
135             IVToken(market).comptroller() == IVault(vaultImplementation).comptroller(),
136             'PoolImplementation.addMarket: wrong comptroller'
137         );
138         require(
139             swapper.isSupportedToken(underlying),
```

```

140         'PoolImplementation.addMarket: no swapper support'
141     );
142     require(
143         markets[underlying] == address(0),
144         'PoolImplementation.addMarket: replace not allowed'
145     );

147     markets[underlying] = market;
148     approveSwapper(underlying);
149 }

151 function approveSwapper(address underlying) public _onlyAdmin_ {
152     uint256 allowance = IERC20(underlying).allowance(address(this), address(swapper)
153     );
154     if (allowance != type(uint256).max) {
155         if (allowance != 0) {
156             IERC20(underlying).safeApprove(address(swapper), 0);
157         }
158         IERC20(underlying).safeApprove(address(swapper), type(uint256).max);
159     }
160 }

161 function collectProtocolFee() external _onlyAdmin_ {
162     require(protocolFeeCollector != address(0), 'PoolImplementation.
163         collectProtocolFee: collector not set');
164     uint256 amount = protocolFeeAccrued.itou();
165     protocolFeeAccrued = 0;
166     IERC20(tokenB0).safeTransfer(protocolFeeCollector, amount);
167     emit CollectProtocolFee(protocolFeeCollector, amount);
168 }

```

Listing 3.1: Various Setters in PoolImplementation

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it would be worrisome if the `admin` is not governed by a DAO-like structure. The discussion with the team has confirmed that it is currently managed by a timelock contract with mandatory minimum 2 days cool down period. We point out that a compromised `admin` account would allow the attacker to undermine necessary assumptions behind the protocol and subvert various protocol operations.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 `timelock` contract. Any admin operations will be executed through the `timelock` contract which imposes a mandatory minimum 2 days cool down period between submitting a transaction and executing it. The community can use this period to check the transaction and its possible consequence.

3.2 Removal of Unused State/Code

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PoolImplementation`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

The `Deri-V3` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, and `SafeMath`, to facilitate its code implementation and organization. For example, the smart contract `Swapper` 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 `PoolImplementation` contract, it defines a number of states: `tokenB0`, `tokenWETH`, `vTokenB0`, `vTokenETH`, and `oracleManager`. However, it turns out a specific state `tokenWETH` can be actually avoided. Specifically, it is so far only used in the constructor. The unused state can be safely removed.

```

57     address public immutable vaultTemplate;
58
59     address public immutable vaultImplementation;
60
61     address public immutable tokenB0;
62
63     address public immutable tokenWETH;
64
65     address public immutable vTokenB0;
66
67     address public immutable vTokenETH;
68
69     IDToken public immutable lToken;
70
71     IDToken public immutable pToken;

```

Listing 3.2: Example States Defined in `PoolImplementation`

Recommendation Consider the removal of the unused state in the above `PoolImplementation` contract.

Status This issue has been resolved. Particularly, the state `tokenWETH` is defined as an `immutable` parameter which doesn't consume any storage, and it is useful to directly query this address from the `Pool` contract. With that, the team considers it convenient by making a redundant parameter `tokenWETH`.

3.3 Meaningful Events For Important States Change

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PoolImplementation`
- Category: Coding Practices [5]
- 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 `PoolImplementation` contract as an example. This contract has public functions that are used to create new markets into the protocol. While examining the events that reflect the market changes, we notice certain events are not properly emitted. To elaborate, we show below the `addMarket()` function.

```

124     function addMarket(address market) external _onlyAdmin_ {
125         address underlying = IVToken(market).underlying();
126         require(
127             IERC20(underlying).decimals() == 18,
128             'PoolImplementation.addMarket: only token of decimals 18'
129         );
130         require(
131             IVToken(market).isVToken(),
132             'PoolImplementation.addMarket: invalid vToken'
133         );
134         require(
135             IVToken(market).comptroller() == IVault(vaultImplementation).comptroller(),
136             'PoolImplementation.addMarket: wrong comptroller'
137         );
138         require(
139             swapper.isSupportedToken(underlying),
140             'PoolImplementation.addMarket: no swapper support'
141         );
142         require(

```

```
143     markets[underlying] == address(0),  
144     'PoolImplementation.addMarket: replace not allowed'  
145 );  
  
147     markets[underlying] = market;  
148     approveSwapper(underlying);  
149 }
```

Listing 3.3: PoolImplementation::addMarket()

Recommendation Properly emit the AddMarket event when a new market is being added into the protocol. This is very helpful for external off-chain analytics and monitoring tools.

Status This issue has been fixed by emitting the proper market-adding event.



4 | Conclusion

In this audit, we have analyzed the `Deri-v3` 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 addressed.

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-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [8] PeckShield. PeckShield Inc. <https://www.peckshield.com>.