SMART CONTRACT AUDIT REPORT

for

PikaPerpV3

Prepared By: Xiaomi Huang

PeckShield

May 10, 2023

Document Properties

Client	Pika Protocol
Title	Smart Contract Audit Report
Target	PikaPerpV3
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc	April 16, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
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 PikaV3 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Pika

Pika protocol is a decentralized perpetual swap exchange on Ethereum layer 2 with a number of features, including high leverage, deep liquidity, numerous assets for trade, limit orders, as well as user-friendly composability with other DeFi systems. The protocol token PIKA is designed to facilitate and incentivize the decentralized governance of the protocol. PIKA holders can lock PIKA for different periods to get vePIKA. A portion of the protocol fees are distributed to vePIKA holders as reward. The protocol fees come from the liquidation reward and interest fees. esPIKA is a token that can be vested to PIKA via a vesting contract, and it might be distributed as rewards to protocol contributors such as vault stakers, vePIKA holders or maybe traders, etc. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Pika

Item	Description
Name	Pika Protocol
Website	https://www.pikaprotocol.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 10, 2023

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

- https://github.com/PikaProtocol/PikaPerpV2/tree/v3Audit (11186cb)
- https://github.com/PikaProtocol/PikaPerpV2/tree/v3Audit2 (f290a6d)

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

- https://github.com/PikaProtocol/PikaPerpV2/tree/v3Audit2 (27dc167)

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

1.3 Methodology

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

Table 1.3: The Full Audit Checklist

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

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

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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `PikaV3` smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
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 need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 4 low-severity vulnerabilities

Table 2.1: Key PikaPerpV3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect Order Management Logic in OrderBook	Business Logic	Resolved
PVE-002	Low	Consistent Order Management in OrderBook	Coding Practices	Resolved
PVE-003	Low	Inconsistent Reentrancy Enforcement in PikaPerpV3	Coding Practices	Resolved
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-005	Low	Improved Event Generation in PositionManager/OrderBook	Coding Practices	Resolved
PVE-006	Medium	Revisited Logic in PikaPerpV3::modifyMargin()	Business Logic	Resolved

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

3 | Detailed Results

3.1 Incorrect Order Management Logic in OrderBook

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OrderBook
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The PikaV3 protocol has an OrderBook contract to faciliate the interaction with the main PikaPerpV3 contract. While examining the current helper routines, we notice a specific one can be improved.

Specifically, this affected routine _createOpenOrder() is designed to execute an operation to create a new user position. It comes to our attention that it locates the openOrdersIndex with the caller (msg.sender), instead of the given _account. As a result, the created order may overwrite unintended ones. Note the same issue is also applicable to another routine, i.e., cancelCloseOrder().

```
    function _createOpenOrder(
        address _account,
        uint256 _productId,
        uint256 _margin,
        uint256 _tradeFee,
        uint256 _leverage,
        bool _isLong,
        uint256 _triggerPrice,
        bool _triggerAboveThreshold,
        uint256 _executionFee
    ) private {
        uint256 _orderIndex = openOrdersIndex[msg.sender];
        OpenOrder memory order = OpenOrder(
            _account,
            _productId,
            _margin,
            _leverage,
            _tradeFee,
```

```
            _isLong,
            _triggerPrice,
            _triggerAboveThreshold,
            _executionFee,
            block.timestamp
        );
        openOrdersIndex[_account] = _orderIndex.add(1);
        openOrders[_account][_orderIndex] = order;
        emit CreateOpenOrder(
            _account,
            _orderIndex,
            _productId,
            _margin,
            _leverage,
            _tradeFee,
            _isLong,
            _triggerPrice,
            _triggerAboveThreshold,
            _executionFee,
            block.timestamp
        );
    }
```

Listing 3.1: `OrderBook::_createOpenOrder()`

Recommendation Revise the above affected routines to properly provide the user account, instead of `msg.sender`.

Status This issue has been resolved by following the above the suggestions.

3.2 Consistent Order Management in OrderBook

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OrderBook`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, the `OrderBook` contract is designed to maintain an order book and faciliate the interaction with the main `PikaPerp` contract. While reviewing the current order management, we observe an inconsistency that can be better resolved.

To elaborate, we show below the two related routines: `createOpenOrder()` and `createCloseOrder()`. As the names indicate, the former is used to create an open order while the latter is designed to create a close order. It comes to our attention that the created open order does not validate the caller with `require(msg.sender == _account || _validateManager(_account)`, while the created close order does. It is important to validate the caller for authorization need.

```
    function createOpenOrder(
        address _account,
        uint256 _productId,
        uint256 _margin,
        uint256 _leverage,
        bool _isLong,
        uint256 _triggerPrice,
        bool _triggerAboveThreshold,
        uint256 _executionFee
    ) external payable nonReentrant {
        require(_executionFee >= minExecutionFee, "OrderBook: insufficient execution fee");

        uint256 tradeFee = _getTradeFeeRate(_productId, _account) * _margin * _leverage / (FEE_BASE * BASE);
        if (IERC20(collateralToken).isETH()) {
            IERC20(collateralToken).uniTransferFromSenderToThis((_executionFee + _margin + tradeFee) * tokenBase / BASE);
        } else {
            require(msg.value == _executionFee * 1e18 / BASE, "OrderBook: incorrect execution fee transferred");
            IERC20(collateralToken).uniTransferFromSenderToThis((_margin + tradeFee) * tokenBase / BASE);
        }
        ...
    }
```

Listing 3.2: `OrderBook::createOpenOrder()`

```
    function createCloseOrder(
        address _account,
        uint256 _productId,
        uint256 _size,
        bool _isLong,
        uint256 _triggerPrice,
        bool _triggerAboveThreshold
    ) external payable nonReentrant {
        require(msg.value >= minExecutionFee * 1e18 / BASE, "OrderBook: insufficient execution fee");
        require(msg.sender == _account  _validateManager(_account), "PositionManager: no permission for account");
        _createCloseOrder(
            _account,
            _productId,
            _size,
            _isLong,
            _triggerPrice,
            _triggerAboveThreshold
```

```
        );
    }
```

Listing 3.3: `OrderBook::createCloseOrder()`

Recommendation Revise the above inconsistency by enforcing the caller validation.

Status This issue has been fixed in the following commit: `3dc49e9`.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `PikaPerpV3` as an example, the `liquidatePositions()` function (see the code snippet below) is provided to externally call a contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 522) start before effecting the update on internal states (`vault.balance`), hence violating the principle.

```
434    function liquidatePositions(uint256[] calldata positionIds) external {
435        require(liquidators[msg.sender]  allowPublicLiquidator, "!liquidator");
436
437        uint256 totalLiquidatorReward;
438        for (uint256 i = 0; i < positionIds.length; i++) {
439            uint256 positionId = positionIds[i];
440            uint256 liquidatorReward = liquidatePosition(positionId);
441            totalLiquidatorReward = totalLiquidatorReward + liquidatorReward;
```

```
        }
        if (totalLiquidatorReward > 0) {
            IERC20(token).uniTransfer(msg.sender, totalLiquidatorReward * tokenBase /
                BASE);
        }
    }
```

Listing 3.4: `PikaPerpV3::liquidatePositions()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been resolved as the team rules out the possibility of `re-entrancy`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `PikaPerpV3` protocol, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

In particular, the privileged functions in the `Pika` contract allows for the the `MINTER_ROLE` to mint new `PIKA/esPika` tokens, and for the `ADMIN_ROLE` to toggle whether the contract allows token transfer, etc.

```
    /// @dev Mints tokens to a recipient.
    ///
    /// This function reverts if the caller does not have the minter role.
    function mint(address _recipient, uint256 _amount) external onlyMinter {
        _mint(_recipient, _amount);
    }

    /// @dev Toggles transfer allowed flag.
    ///
    /// This function reverts if the caller does not have the admin role.
    function setTransfersAllowed(bool _transfersAllowed) external onlyAdmin {
        transfersAllowed = _transfersAllowed;
        emit TransfersAllowed(transfersAllowed);
    }
```

Listing 3.5: Privileged Operations in `Pika`

In addition, the privileged functions in the `PositionManager` contract allow for the `admin` to configure various protocol parameters.

```
    function setFeeCalculator(address _feeCalculator) external onlyAdmin {
        feeCalculator = _feeCalculator;
    }

    function setOracle(address _oracle) external onlyAdmin {
        oracle = _oracle;
    }

    function setPositionKeeper(address _account, bool _isActive) external onlyAdmin {
        isPositionKeeper[_account] = _isActive;
        emit SetPositionKeeper(_account, _isActive);
    }

    function setMinExecutionFee(uint256 _minExecutionFee) external onlyAdmin {
        minExecutionFee = _minExecutionFee;
        emit SetMinExecutionFee(_minExecutionFee);
    }

    function setIsUserExecuteEnabled(bool _isUserExecuteEnabled) external onlyAdmin {
        isUserExecuteEnabled = _isUserExecuteEnabled;
        emit SetIsUserExecuteEnabled(_isUserExecuteEnabled);
    }

    function setIsUserCancelEnabled(bool _isUserCancelEnabled) external onlyAdmin {
        isUserCancelEnabled = _isUserCancelEnabled;
        emit SetIsUserCancelEnabled(_isUserCancelEnabled);
    }
```

Listing 3.6: Privileged Operations in `PositionManager`

There are also some other privileged functions not listed above. And We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to these privileged accounts explicit to `Pika` protocol users.

Status This issue has been confirmed by the team with use of a multi-sig account for admin management.

3.5 Generation of Meaningful Events For Important State Changes

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SharerV4`, `CommonHealthCheck`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

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 `OrderBook` contract as an example. This contract has public functions that are used to execute the close order. While examining the events that reflect the `order` changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `referralStorage` not configured, the event of `ExecuteCloseOrder` is not emitted. Note the same issue is also applicable to another routine `executeOpenPosition()` in both `OrderBook` and `PositionManager` contracts.

```
    function executeCloseOrder(address _address, uint256 _orderIndex, address payable
        _feeReceiver) public nonReentrant {
        CloseOrder memory order = closeOrders[_address][_orderIndex];
        require(order.account != address(0), "OrderBook: non-existent order");
        require(msg.sender == address(this), "OrderBook: not calling from this contract"
            );
        (,uint256 leverage,,,,,,) = IPikaPerp(pikaPerp).getPosition(_address, order.
            productId, order.isLong);
        (uint256 currentPrice, ) = validatePositionOrderPrice(
            !order.isLong,
            order.triggerAboveThreshold,
            order.triggerPrice,
            order.productId
        );

        delete closeOrders[_address][_orderIndex];
        IPikaPerp(pikaPerp).closePosition(_address, order.productId, order.size * BASE /
             leverage , order.isLong, currentPrice);

        // pay executor
        _feeReceiver.sendValue(order.executionFee * 1e18 / BASE);
```

```
        if (referralStorage == address(0)) {
            return;
        }
        (bytes32 referralCode, address referrer) = IReferralStorage(referralStorage).
            getTraderReferralInfo(order.account);

        emit ExecuteCloseOrder(
            order.account,
            _orderIndex,
            order.productId,
            order.size,
            order.isLong,
            order.triggerPrice,
            order.triggerAboveThreshold,
            order.executionFee,
            currentPrice,
            order.orderTimestamp,
            referralCode,
            referrer
        );
    }
```

Listing 3.7: `OrderBook::executeCloseOrder()`

Recommendation Properly emit respective events when current orders are updated or executed

Status This issue has been fixed in the following commit: `602f409`.

3.6 Revisited Logic in PikaPerpV3::modifyMargin()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PikaPerpV3`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `PikaPerpV3` protocol has the main `PikaPerpV3` contract that manages the user positions. While examining a key routine to update a given position's margin, we notice the current implementation can be improved.

Specifically, we show below the related implementation. While it properly achieves the tasked logic, we notice it does not ensure the position after the margin modification is healthy. This needs to be fixed so that the position after margin decrease should not be liquidatable!

```
    function modifyMargin(uint256 positionId, uint256 margin, bool shouldIncrease)
        external payable nonReentrant {

        // Check position
        Position storage position = positions[positionId];
        require(msg.sender == position.owner  _validateManager(position.owner), "!allow"
            );
        uint256 newMargin;
        if (shouldIncrease) {
            IERC20(token).uniTransferFromSenderToThis(margin * tokenBase / BASE);
            newMargin = uint256(position.margin) + margin;
        } else {
            newMargin = uint256(position.margin) - margin;
            IERC20(token).uniTransfer(msg.sender, margin * tokenBase / BASE);
        }

        // New position params
        uint256 newLeverage = uint256(position.leverage) * uint256(position.margin) /
            newMargin;
        require(newLeverage >= 1 * BASE, "!low-lev");

        position.margin = uint128(newMargin);
        position.leverage = uint64(newLeverage);

        emit ModifyMargin(
            positionId,
            msg.sender,
            position.owner,
            margin,
            newMargin,
            newLeverage,
            shouldIncrease
        );

    }
```

Listing 3.8: `PikaPerpV3::modifyMargin()`

Recommendation Revise the above affected routine to properly ensure the position is healthy after margin adjustment.

Status This issue has been fixed in the following commit: `c6edfb4`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the PikaV3 protocol, which is a decentralized perpetual swap exchange on Ethereum layer 2 with a number of features, including high leverage, deep liquidity, numerous assets for trade, limit orders, as well as user-friendly composability with other DeFi systems. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

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

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