# SMART CONTRACT AUDIT REPORT

for

# Pika Protocol

Prepared By: Xiaomi Huang

PeckShield

July 22, 2022

## Document Properties

| | |
|---|---|
| Client | Pika Protocol |
| Title | Smart Contract Audit Report |
| Target | PikaPerpV3 |
| Version | 1.1 |
| Author | Luck Hu |
| Auditors | Luck Hu, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.1 | July 22, 2022 | Luck Hu | Final Release |
| 1.0 | May 31, 2022 | Luck Hu | Release #1 |
| 1.0-rc | April 28, 2022 | Luck Hu | 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 Pika 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 has 3 tokens, i.e., PIKA, vePIKA and esPIKA. PIKA is designed to facilitate and incentivize the decentralized governance of the protocol. PIKA holders can lock PIKA for different periods to get vePIKA. The longer the lock period, the more vePIKA the holder gets. 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 | July 22, 2022 |

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

- https://github.com/PikaProtocol/PikaPerpV2/tree/v3 (8139be4)

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/v3 (1eac910)

## 1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

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

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

To evaluate the risk, we go through a 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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 `Pika` 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 | 3 | ■ ■ ■ |
| Low | 1 | ■ |
| Informational | 0 | |
| Total | 4 | |

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 3 medium-severity vulnerabilities and 1 low-severity vulnerability

Table 2.1: Key PikaPerpV3 Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Improved claimable Calculation in claimable() | Business Logic | Fixed |
| PVE-002 | Low | Improper Validation of Function Arguments | Business Logic | Fixed |
| PVE-003 | Medium | Trust Issue Of Admin Keys | Security Features | Confirmed |
| PVE-004 | Medium | Incorrect ETH tokenBase Used in OrderBook | Coding Practices | Fixed |

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 Improved claimable Calculation in claimable()

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

### Description

In the `Pika` protocol, the `Vester` contract is implemented to support the vesting of `esPIKA` token to `PIKA` token. The vesting is carried out by depositing certain amount of `esPIKA` tokens into the contract with the vesting period (set by the `owner`). The stakers can claim the same amount of `PIKA` tokens from the contract in the whole vesting period.

To elaborate, we show below the code snippets of the `claimable()` and `setVestingPeriod()` routines. As the name indicate, the `setVestingPeriod()` is designed for the `owner` to update the `vestingPeriod` (vesting period), and the `claimable()` routine is designed to calculate the amount of `PIKA` tokens that are claimable for the given `_account` and `_depositId`. The claimable amount is calculated in proportional to the vested time in the vesting period. While examining the logic to calculate the claimable amount, it comes to our attention that, if the `_depositId` is created before the `vestingPeriod` is updated, the `claimable()` routine may return an unexpected amount for the given `_depositId`. Because the claimable amount shall be calculated per the dedicated `vestingPeriod` which is used to create the `_depositId`.

```
    function claimable(address _account, uint256 _depositId) public view returns(uint256
        ) {
        UserInfo memory user = userInfo[_account][_depositId];
        if (user.vestingLastUpdate > user.vestedUntil   user.claimedAmount >= user.
            depositAmount) {
            return 0;
        }
        if (block.timestamp < user.vestedUntil) {
```

```
            return user.depositAmount * (block.timestamp - user.vestingLastUpdate) /
                vestingPeriod;
        }
        uint256 claimableAmount = user.depositAmount * (user.vestedUntil - user.
            vestingLastUpdate) / vestingPeriod;
        return claimableAmount + user.claimedAmount > user.depositAmount ? user.
            depositAmount - user.claimedAmount : claimableAmount;
    }
```

Listing 3.1: Vester::claimable()

```
    function setVestingPeriod(uint256 _vestingPeriod) external onlyOwner {
        vestingPeriod = _vestingPeriod;
    }
```

Listing 3.2: Vester::setVestingPeriod()

Based on this, it is suggested to record the `vestingPeriod` used to create the `_depositId` and calculate the claimable amount per the recorded `vestingPeriod`.

**Recommendation** Properly revise the above `claimable()` routine to calculate the claimable amount of `PIKA` tokens with the dedicated `vestingPeriod` used to create the deposit.

**Status** This issue has been fixed in the following commit: `16f6013`.

## 3.2 Improper Validation of Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `PikaPriceFeed`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the `PikaPriceFeed` contract, it provides the `setPrices()` routine for the `keeper` to set prices for the supported tokens manually. The manually set prices are valid only in a dedicated duration which is configured by the contract owner.

To elaborate, we show below the code snippet of the `setPriceDuration()` routine. As the name indicates, this routine is designed for the `owner` to update the `priceDuration` (price duration). The max value of the `priceDuration` is `MAX_PRICE_DURATION` (30 minutes). While examining the price duration update logic in the `setPriceDuration()` routine, we notice that it doesn't validate the input parameter `_priceDuration`. Instead, it validates the state variable `priceDuration` (line 106). As a result, if the input `_priceDuration` is bigger than `MAX_PRICE_DURATION`, the `priceDuration` could also be updated successfully. Once this happens, the `priceDuration` can never be updated any more. As a result, the manually set prices for tokens could keep valid for an unexpected duration.

```
    function setPriceDuration(uint256 _priceDuration) external onlyOwner {
        require(priceDuration <= MAX_PRICE_DURATION, "!priceDuration");
        priceDuration = _priceDuration;
        emit PriceDurationSet(priceDuration);
    }
```

Listing 3.3: `PikaPriceFeed::setPriceDuration()`

**Recommendation** Revise the above `setPriceDuration()` routine to validate the input parameter `_priceDuration`.

**Status** This issue has been fixed in the following commit: `16f6013`.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In the `Pika` 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.

Firstly, 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 3.4: `Pika.sol`

Secondly, the privileged function in the `Vester` contract allows for the `owner` to change the vesting period.

```
    function setVestingPeriod(uint256 _vestingPeriod) external onlyOwner {
        vestingPeriod = _vestingPeriod;
    }
```

Listing 3.5: `Vester::setVestingPeriod()`

Lastly, the privileged functions in the `PikaPriceFeed` contract allow for the `keeper` to set tokens prices, set the price duration, etc.

```
    function setPrices(address[] memory tokens, uint256[] memory prices) external
        onlyKeeper {
        for (uint256 i = 0; i < tokens.length; i++) {
            address token = tokens[i];
            priceMap[token] = prices[i];
            emit PriceSet(token, prices[i], block.timestamp);
        }
        lastUpdatedTime = block.timestamp;
    }

    function setPriceDuration(uint256 _priceDuration) external onlyOwner {
        require(priceDuration <= MAX_PRICE_DURATION, "!priceDuration");
        priceDuration = _priceDuration;
        emit PriceDurationSet(priceDuration);
    }
```

Listing 3.6: `PikaPriceFeed.sol`

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 the `owner/minter`, etc. accounts explicit to `Pika` protocol users.

**Status** This issue has been confirmed by the team.

## 3.4 Incorrect ETH tokenBase Used in OrderBook

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OrderBook`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

### Description

In the `Pika` protocol, the `OrderBook` contract is implemented to facilitate users trading with the protocol. It provides routines for traders to create/cancel orders which can be executed to open/close trades. To create an open order, the trader needs to have both the underlying token and `ETH`, where the underlying token is used as the margin and the `ETH` is used as the execution fee. Both the underlying token and `ETH` have their own `tokenBase` which is the denomination of the token.

To elaborate, we show below the code snippets of the `createOpenOrder()` and the `cancelOpenOrder()` routines. As the name indicate, the `createOpenOrder()` routine is designed for users to create orders, and the `cancelOpenOrder()` routine is designed for users to cancel the created orders. While examining the token base used for `ETH` in these two routines, we notice the existence of possible incorrect token base used for `ETH` in the `cancelOpenOrder()` routine. Namely, the `createOpenOrder()` routine uses the $1e18$ as the token base for `ETH` (line 289), while the `cancelOpenOrder()` routine makes use of the `tokenBase` as the token base for `ETH` (line 377). By design, the `tokenBase` is the token base of the `collateralToken`. As a result, if the `collateralToken` is not equal to `ETH`, the `cancelOpenOrder()` routine may use the wrong token base for `ETH`.

```
274     function createOpenOrder(
275         uint256 _productId,
276         uint256 _margin,
277         uint256 _leverage,
278         bool _isLong,
279         uint256 _triggerPrice,
280         bool _triggerAboveThreshold,
281         uint256 _executionFee
282     ) external payable nonReentrant {
283         require(_executionFee >= minExecutionFee, "OrderBook: insufficient execution fee
            ");
284         (,uint256 maxLeverage,,,,,,,,,) = IPikaPerp(pikaPerp).getProduct(_productId);
285         require(_leverage <= maxLeverage, "leverage too high");
286         if (IERC20(collateralToken).isETH()) {
287             IERC20(collateralToken).uniTransferFromSenderToThis((_executionFee + _margin
                 * _leverage / BASE) * tokenBase / BASE);
288         } else {
289             require(msg.value == _executionFee * 1e18 / BASE, "OrderBook: incorrect
                execution fee transferred");
```

```
            IERC20(collateralToken).uniTransferFromSenderToThis((_margin * _leverage /
                BASE) * tokenBase / BASE);
        }

        _createOpenOrder(
            msg.sender,
            _productId,
            _margin,
            _leverage,
            _isLong,
            _triggerPrice,
            _triggerAboveThreshold,
            _executionFee
        );
    }
```

Listing 3.7: `OrderBook::createOpenOrder()`

```
    function cancelOpenOrder(uint256 _orderIndex) public nonReentrant {
        OpenOrder memory order = openOrders[msg.sender][_orderIndex];
        require(order.account != address(0), "OrderBook: non-existent order");

        delete openOrders[msg.sender][_orderIndex];

        if (IERC20(collateralToken).isETH()) {
            IERC20(collateralToken).uniTransfer(msg.sender, (order.executionFee + order.
                margin * order.leverage / BASE) * tokenBase / BASE);
        } else {
            IERC20(collateralToken).uniTransfer(msg.sender, (order.margin * order.
                leverage / BASE) * tokenBase / BASE);
            payable(msg.sender).sendValue(order.executionFee.mul(tokenBase).div(BASE));
        }

        emit CancelOpenOrder(
            order.account,
            _orderIndex,
            order.productId,
            order.margin,
            order.leverage,
            order.isLong,
            order.triggerPrice,
            order.triggerAboveThreshold,
            order.executionFee
        );
    }
```

Listing 3.8: `OrderBook::cancelOpenOrder()`

Note the same issue also exists in the `executeOpenOrder()/createCloseOrder()/_createCloseOrder()/cancelCloseOrder()` routines.

**Recommendation** Revise the above mentioned routines to use the consistent $1e18$ as the token base for ETH.

**Status** This issue has been fixed in the following commit: 16f6013.

# 4 Conclusion

In this audit, we have analyzed the Pika protocol design and implementation. 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 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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