SMART CONTRACT AUDIT REPORT

for

Hybra

Prepared By: Xiaomi Huang

PeckShield

July 25, 2025

Document Properties

Client	Hybra
Title	Smart Contract Audit Report
Target	Hybra
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 25, 2025	Xuxian Jiang	Final Release
1.0-rc	July 25, 2025	Xuxian Jiang	Release Candidate #1

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 source code of the `Hybra` 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 Hybra

`Hybra` is a community-owned public liquidity layer on `Hyperliquid`, designed to deliver deep, capital-efficient liquidity with low fees and minimal price impact for traders, while tightly aligning rewards with long-term `LPs`. Liquidity is supplied through `UniswapV3`–compatible concentrated-liquidity pools and a `ve(3,3)`-style gauge/emissions system. `LPs` earn swap fees, `HYBR` emissions and, locking in deterministic upside for early participants. Governance is driven by `veHYBR`, which directs emissions and fee flows to pools. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Hybra

Item	Description
Name	Hybra
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 25, 2025

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

- https://github.com/hybra-finance/hybra-v3-cl.git (ca361e8)

1.2   About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the secu-
rity, 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

	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact (vertical axis: High, Medium, Low)   **Likelihood** (horizontal axis: High, Medium, 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 accordingly classified
into four categories, 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

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

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:

- <u>Basic Coding Bugs</u>: 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.

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

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

- <u>Additional Recommendations</u>: 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. 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 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 1.4:   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 `Hybra` protocol. 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	0	
Low	1	◼
Informational	1	◼
Total	2	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 low-severity vulnerability and 1 informational issue.

Table 2.1: Key Hybra Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Flashloan Protocol Fee Distribution Logic	Business Logic	Confirmed
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1   Revisited Flashloan Protocol Fee Distribution Logic

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `UniswapV3Pool`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, the `Hybra` protocol is in essence a `DEX` engine that facilitates the swaps between tokens. It also supports the flashloan feature that allows users to borrow assets without having to provide collateral or a credit score. This type of loan has to be paid back within the same blockchain transaction block. While reviewing the flashloan logic, we notice the way to distribute flashloan fee may need to be revisited.

To elaborate, we show below the related `flash()` routine. It has a rather straightforward logic in making the liquidity available to flashloaners and collecting the flashloan fee accordingly. Note the flashloan funds are pooled together from all liquidity providers. However, the flashloan fee is only credited to in-range liquidity providers, not all liquidity providers. This design may need to be revisited.

```
791    function flash(
792        address recipient,
793        uint256 amount0,
794        uint256 amount1,
795        bytes calldata data
796    ) external override lock noDelegateCall {
797        uint128 _liquidity = liquidity;
798        require(_liquidity > 0, 'L');
799
800        uint256 fee0 = FullMath.mulDivRoundingUp(amount0, fee, 1e6);
801        uint256 fee1 = FullMath.mulDivRoundingUp(amount1, fee, 1e6);
802        uint256 balance0Before = balance0();
```

```
803        uint256 balance1Before = balance1();
804
805        if (amount0 > 0) TransferHelper.safeTransfer(token0, recipient, amount0);
806        if (amount1 > 0) TransferHelper.safeTransfer(token1, recipient, amount1);
807
808        IUniswapV3FlashCallback(msg.sender).uniswapV3FlashCallback(fee0, fee1, data);
809
810        uint256 balance0After = balance0();
811        uint256 balance1After = balance1();
812
813        require(balance0Before.add(fee0) <= balance0After, 'F0');
814        require(balance1Before.add(fee1) <= balance1After, 'F1');
815
816        // sub is safe because we know balanceAfter is gt balanceBefore by at least fee
817        uint256 paid0 = balance0After - balance0Before;
818        uint256 paid1 = balance1After - balance1Before;
819
820        if (paid0 > 0) {
821            uint8 feeProtocol0 = slot0.feeProtocol % 16;
822            uint256 fees0 = feeProtocol0 == 0 ? 0 : paid0 / feeProtocol0;
823            if (uint128(fees0) > 0) protocolFees.token0 += uint128(fees0);
824            feeGrowthGlobal0X128 += FullMath.mulDiv(paid0 - fees0, FixedPoint128.Q128,
                 _liquidity);
825        }
826        if (paid1 > 0) {
827            uint8 feeProtocol1 = slot0.feeProtocol >> 4;
828            uint256 fees1 = feeProtocol1 == 0 ? 0 : paid1 / feeProtocol1;
829            if (uint128(fees1) > 0) protocolFees.token1 += uint128(fees1);
830            feeGrowthGlobal1X128 += FullMath.mulDiv(paid1 - fees1, FixedPoint128.Q128,
                 _liquidity);
831        }
832
833        emit Flash(msg.sender, recipient, amount0, amount1, paid0, paid1);
834    }
```

Listing 3.1: `UniswapV3Pool::flash()`

Recommendation   Revisit the above routine to properly credit the flashloan fee to all liquidity providers.

Status   This issue has been confirmed as the team clarifies the need of maintaining the code consistency with the original `UniswapV3` codebase.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Hybra` protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., configure the fee-related parameters and collect protocol fee). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```solidity
54      function setOwner(address _owner) external override {
55          require(msg.sender == owner);
56          emit OwnerChanged(owner, _owner);
57          owner = _owner;
58      }
59
60      /// @inheritdoc IUniswapV3Factory
61      function enableFeeAmount(uint24 fee, int24 tickSpacing) public override {
62          require(msg.sender == owner);
63          require(fee < 1000000);
64          // tick spacing is capped at 16384 to prevent the situation where tickSpacing is
                    so large that
65          // TickBitmap#nextInitializedTickWithinOneWord overflows int24 container from a
                    valid tick
66          // 16384 ticks represents a >5x price change with ticks of 1 bips
67          require(tickSpacing > 0 && tickSpacing < 16384);
68          require(feeAmountTickSpacing[fee] == 0);
69
70          feeAmountTickSpacing[fee] = tickSpacing;
71          emit FeeAmountEnabled(fee, tickSpacing);
72      }
```

Listing 3.2: Example Privileged Functions in `UniswapV3Factory`

```solidity
837     function setFeeProtocol(uint8 feeProtocol0, uint8 feeProtocol1) external override
            lock onlyFactoryOwner {
838         require(
839             (feeProtocol0 == 0  (feeProtocol0 >= 4 && feeProtocol0 <= 10)) &&
840                 (feeProtocol1 == 0  (feeProtocol1 >= 4 && feeProtocol1 <= 10))
841         );
842         uint8 feeProtocolOld = slot0.feeProtocol;
843         slot0.feeProtocol = feeProtocol0 + (feeProtocol1 << 4);
844         emit SetFeeProtocol(feeProtocolOld % 16, feeProtocolOld >> 4, feeProtocol0,
                feeProtocol1);
```

```
845        }
846
847        /// @inheritdoc IUniswapV3PoolOwnerActions
848        function collectProtocol(
849            address recipient,
850            uint128 amount0Requested,
851            uint128 amount1Requested
852        ) external override lock onlyFactoryOwner returns (uint128 amount0, uint128 amount1)
              {
853            amount0 = amount0Requested > protocolFees.token0 ? protocolFees.token0 :
                  amount0Requested;
854            amount1 = amount1Requested > protocolFees.token1 ? protocolFees.token1 :
                  amount1Requested;
855
856            if (amount0 > 0) {
857                if (amount0 == protocolFees.token0) amount0--; // ensure that the slot is
                      not cleared, for gas savings
858                protocolFees.token0 -= amount0;
859                TransferHelper.safeTransfer(token0, recipient, amount0);
860            }
861            if (amount1 > 0) {
862                if (amount1 == protocolFees.token1) amount1--; // ensure that the slot is
                      not cleared, for gas savings
863                protocolFees.token1 -= amount1;
864                TransferHelper.safeTransfer(token1, recipient, amount1);
865            }
866
867            emit CollectProtocol(msg.sender, recipient, amount0, amount1);
868        }
```

Listing 3.3: Example Privileged Functions in `UniswapV3Pool`

Note that this privileged account only affects protocol fees, and does not pose any security risks to user funds. From another perspective, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 mitigated as the team makes use of a multisig to act as the privileged owner.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Hybra` protocol, which is a community-owned public liquidity layer on `Hyperliquid`, designed to deliver deep, capital-efficient liquidity with low fees and minimal price impact for traders, while tightly aligning rewards with long-term `LPs`. Liquidity is supplied through `UniswapV3`–compatible concentrated-liquidity pools and a `ve(3,3)`-style gauge/emissions system. `LPs` earn swap fees, `HYBR` emissions and, locking in deterministic upside for early participants. Governance is driven by `veHYBR`, which directs emissions and fee flows to pools. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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.
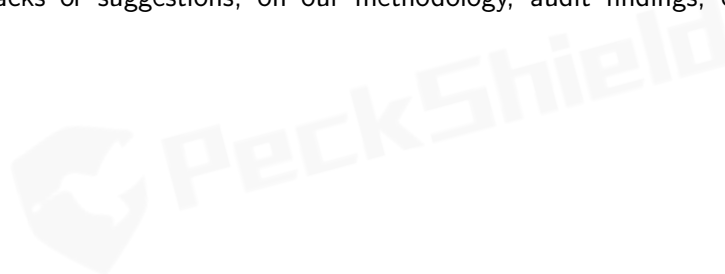
References

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

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

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

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.