SMART CONTRACT AUDIT REPORT

for

Hybra Finance

Prepared By: Xiaomi Huang

PeckShield

October 22, 2025

Document Properties

Client	Hybra Finance
Title	Smart Contract Audit Report
Target	Hybra Finance
Version	1.0.1
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
1.0.1	October 22, 2025	Xuxian Jiang	Post-Final Release #1
1.0	September 29, 2025	Xuxian Jiang	Final Release
1.0-rc	September 28, 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 Finance` is the public liquidity layer on `HyperliquidX`, fully community-owned with no insider allocations. It features an upgraded `ve(3,3)` flywheel with in-house, highly competitive dynamic-fee `CL pools` and introduces `G33`, a `PVP`-style mechanism that rewards loyal `LPs` with higher yields while reducing rewards for short-term capital — reinforcing long-term alignment and making liquidity incentives more sustainable The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Hybra Finance

Item	Description
Name	Hybra Finance
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 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-finance.git (8496c32)

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

- https://github.com/hybra-finance/hybra-finance.git (e056e64, a394e75)

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 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 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

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

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 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 Finance` 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	2	■■
Low	6	■■■■■■
Informational	1	■
Total	9	

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 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational issue.

Table 2.1: Key Hybra Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper transferFrom() Logic in RewardHYBR	Business Logic	Resolved
PVE-002	Low	Revisited withdraw() Logic in RewardHYBR	Business Logic	Resolved
PVE-003	Informational	Possible ERC7702 Incompatibility in Contract Check	Business Logic	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Resolved
PVE-005	Low	Simplified removeRole() Logic in PermissionsRegistry	Coding Practices	Resolved
PVE-006	Low	Voting Delegate Denial-of-Service With Dust Delegates	Numeric Errors	Mitigated
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Low	Improved Dynamic Fee Calculation in DynamicSwapFeeModule	Business Logic	Resolved
PVE-009	Low	Improper estimateAmount0/1() Logic in SugarHelper	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 contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper transferFrom() Logic in RewardHYBR

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

Description

The `Hybra` protocol has a core `RewardHYBR` contract that implements a dynamic conversion rate mechanism to encourage long-term locking. This contract is implemented as an `ERC20`-compliant `rHYBR` token and our analysis shows its current transfer logic should be revisited.

To elaborate, we show below the related `transferFrom()` routine. It has a rather straightforward logic in transferring tokens from the given sender to the intended recipient. However, it comes to our attention that it does not check the sender's allowance to the calling user. As a result, any use may steal tokens from others. Fortunately, throughout the entire lifecycle of `rHYBR`, no normal users will ever directly hold `rHYBR` tokens.

```
    function transferFrom(address from, address to, uint256 amount) external returns (
        bool) {
        _transfer(from, to, amount);
        return true;
    }

    function _transfer(address from, address to, uint256 amount) internal {
        if (amount == 0) revert ZeroAmount();
        if (balanceOf[from] < amount) revert InsufficientBalance();

        // Check transfer permissions
        uint8 allowed = 0;
        if (_isExempted(from, to)) {
            allowed = 1;
```

```
        } else if (gaugeManager != address(0) && IGaugeManager(gaugeManager).isGauge(
            from)) {
            exempt.add(from);
            allowed = 1;
        }

        if (allowed != 1) revert TransferNotAllowed();

        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        emit Transfer(from, to, amount);
    }
```

Listing 3.1: `RewardHYBR::transferFrom()`

Recommendation Revisit the above routine to properly validate the sender allowance. If necessary, simply disable the transfer function.

Status This issue has been fixed in the following commit: `e056e64`.

3.2 Revisited withdraw() Logic in RewardHYBR

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

Description

In Section 3.1, we have examined the `RewardHYBR` contract and reported an issue in its transfer functionality. In this section, we report another issue that mistakenly supports a public `withdraw()` function.

To elaborate, we show below the implementation of this `withdraw()` function. It has a rather straightforward logic in simply burning the given amount from the calling user. This function should not be present since `rHYBR` is only minted via `depositEmissionToken()` and burned via `redeemFor()`.

```
    function withdraw(uint256 amount) external {
        _burn(msg.sender, amount);
    }
```

Listing 3.2: `RewardHYBR::withdraw()`

Recommendation Remove the above `withdraw()` function.

Status This issue has been fixed in the following commit: `e056e64`.

3.3 Possible ERC7702 Incompatibility in Contract Check

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VotingEscrow`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The voting power in `Hybra` is represented by `ERC721`-compliant tokens implemented in `VotingEscrow`. As part of `ERC721` compliance, the `NFT` logic supports `safeTransferFrom()` to transfer the ownership of an `NFT` from one address to another. As part of the transfer logic, it allows the callback on the recipient if the recipient address is a contract. Our analysis shows that the logic to check whether the given recipient is a contract can be improved.

In the following, we show the implementation of this specific function `_isContract()`. It has a rather straightforward logic in detecting whether the destination is a contract based on `extcodesize (account)>0`, which does not take into consideration of the `ERC7702` specification. In particular, the `ERC7702` specification allows any `EOA` to set its code based on any existing smart contract. To accommodate the `ERC7702` specification, there is a need to improve the logic to reliably detect whether a given address is a true contract (excluding `ERC7702`-enabled `EOAs`).

```
    function _isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.
        uint size;
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }
```

Listing 3.3: `VotingEscrow::_isContract()`

Recommendation Revise the above `_isContract()` function to properly detect whether a given address is a contract. An example implementation can be found as follows:

```
    function _isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.
        uint256 csize = address(who).code.length;

        // EOA
```

```
        if (csize == 0) return false;

        // Delegated EOA (EIP-7702)
        if (csize == 23) {
            bytes32 word;
            assembly {
                let ptr := mload(0x40)
                extcodecopy(who, ptr, 0, 3) // copy first 3 bytes
                word := mload(ptr) // load the 32-byte word
            }
            return bytes3(word) != 0xef0100;
        }

        return true;
    }
```

Listing 3.4: Revised `VotingEscrow::_isContract()`

Status This issue has been resolved since the current logic does not introduce any business impact, as failed executions will revert as expected.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasisStrategy`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
```

```
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.5: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.6: `SafeERC20::safeApprove()`

In current implementation, if we examine the `GovernanceHYBR::executeSwap()` routine that is designed to perform token swaps. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (lines 429 and 435).

```
    function executeSwap(ISwapper.SwapParams calldata _params) external nonReentrant
        onlyOperator {
        require(address(swapper) != address(0), "Swapper not set");

        // Get token balance before swap
        uint256 tokenBalance = IERC20(_params.tokenIn).balanceOf(address(this));
        require(tokenBalance >= _params.amountIn, "Insufficient token balance");

        // Approve swapper to spend tokens
        IERC20(_params.tokenIn).approve(address(swapper), _params.amountIn);

        // Execute swap through swapper module
        uint256 hybrReceived = swapper.swapToHYBR(_params);

        // Reset approval for safety
        IERC20(_params.tokenIn).approve(address(swapper), 0);

        // HYBR is now in this contract, ready for compounding
    }
```

Listing 3.7: `GovernanceHYBR::executeSwap()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`. Note this issue affects a number of contracts, including `RewardHYBR`, `RewardsDistributor`, and `GovernanceHYBR`.

Status This issue has been fixed in the following commit: `e056e64`.

3.5 Simplified removeRole() Logic in PermissionsRegistry

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PermissionsRegistry`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate role management, `Hybra` has a dedicated `PermissionsRegistry` contract. While reviewing the logic to revoke roles, we notice an issue that does not cleanly reset related storage states.

To elaborate, we show below the implementation of the related `removeRole()` routine. This routine has a rather simple logic in removing a given role. While current implementation does properly remove the given role from the `_roles` array and the `_checkRole` mapping, it does not reset the `_roleToAddresses` array. In addition, the adjustment on the `addressToRoles` array can perform early exit once the removed role is identified and removed (line 114).

```
    function removeRole(string memory role) external onlyHybraMultisig {
        bytes memory _role = bytes(role);
        require(_checkRole[_role], 'not a role');

        for(uint i = 0; i < _roles.length; i++){
            if(keccak256(_roles[i]) == keccak256(_role)){
                _roles[i] = _roles[_roles.length -1];
                _roles.pop();
                _checkRole[_role] = false;
                emit RoleRemoved(_role);
                break;
            }
        }

        address[] memory rta = _roleToAddresses[bytes(role)];
        for(uint i = 0; i < rta.length; i++){
            hasRole[bytes(role)][rta[i]] = false;
            bytes[] memory __roles = _addressToRoles[rta[i]];
            for(uint k = 0; k < __roles.length; k++){
                if(keccak256(__roles[k]) == keccak256(bytes(role))){
                    _addressToRoles[rta[i]][k] = _roles[_roles.length -1];
                    _addressToRoles[rta[i]].pop();
                }
            }
        }

    }
```

Listing 3.8: `PermissionsRegistry::removeRole()`

Similarly, the early exit optimization can be applied to another `removeRoleFrom()`. routine as well.

Recommendation Revise the above-mentioned routines to cleanly reset outdated states.

Status This issue has been fixed in the following commit: `e056e64`.

3.6 Voting Delegate Denial-of-Service With Dust Delegates

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VotingDelegationLib`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.3, the `VotingEscrow` contract in `Hybra` escrows the governance tokens in the form of an `ERC-721 NFT`, which can be delegated from one user to another. While analyzing the delegation logic, we notice a possible denial-of-service issue.

In the following, we show the implementation of the related `_moveTokenDelegates()` routine. This routine is designed to move the delegated `NFT` from one user to another. We notice the recipient-side requirement upon the `NFT` delegation, i.e., `dstRepOld.length + 1 <= MAX_DELEGATES` (line 97). In other words, there is a limit on the maximum number of received `NFTs`. Further, our analysis shows it is possible to create dust `NFT` and delegate them to one victim user. As a result, the victim user may not be able to recieve legitimate delegation once the number of total delegates reaches the threshold, i.e., `MAX_DELEGATES`.

```
86         if (dstRep != address(0)) {
87             uint32 dstRepNum = self.numCheckpoints[dstRep];
88             uint[] storage dstRepOld = dstRepNum > 0
89                 ? self.checkpoints[dstRep][dstRepNum - 1].tokenIds
90                 : self.checkpoints[dstRep][0].tokenIds;
91             uint32 nextDstRepNum = findCheckpointToWrite(self, dstRep, block.timestamp);
92             bool _isCheckpointInNewBlock = (dstRepNum > 0) ? (nextDstRepNum != dstRepNum
                - 1) : true;
93             Checkpoint storage cpDstRep = self.checkpoints[dstRep][nextDstRepNum];
94             uint[] storage dstRepNew = cpDstRep.tokenIds;
95             cpDstRep.timestamp = block.timestamp;
96             require(
97                 dstRepOld.length + 1 <= MAX_DELEGATES,
98                 "tokens>1"
99             );
100            if(_isCheckpointInNewBlock) {
101                for (uint i = 0; i < dstRepOld.length; i++) {
102                    uint tId = dstRepOld[i];
103                    dstRepNew.push(tId);
104                }
105            }
106            dstRepNew.push(_tokenId);
107            self.numCheckpoints[dstRep] = nextDstRepNum + 1;
108        }
```

Listing 3.9: `VotingDelegationLib::_moveTokenDelegates()`

Recommendation Improve the above logic by restricting possible dust delegation. Note another routine `_moveAllDelegates()` shares the same issue.

Status This issue has been mitigated as it is part of the intended design.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

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 parameters, assign roles, manage gauges, whitelist tokens, and execute privileged operations). 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.

```
    function addRole(string memory role) external onlyHybraMultisig {
        bytes memory _role = bytes(role);
        require(!_checkRole[_role], 'is a role');
        _checkRole[_role] = true;
        _roles.push(_role);
        emit RoleAdded(_role);
    }

    /// @notice Remove a role
    /// @dev    set last one to i_th position then .pop()
    function removeRole(string memory role) external onlyHybraMultisig {
        ...
    }

    /// @notice Set a role for an address
    function setRoleFor(address c, string memory role) external onlyHybraMultisig {
        ...
    }


    /// @notice remove a role from an address
    function removeRoleFrom(address c, string memory role) external onlyHybraMultisig {
        ...
    }
```

Listing 3.10: Example Privileged Functions in PermissionsRegistry

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.

3.8 Improved Dynamic Fee Calculation in DynamicSwapFeeModule

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DynamicSwapFeeModule
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Hybra protocol allows to customize the swap fee for each specific pool. In the process of reviewing the swap fee logic, we notice it makes use of the built-in geometric mean price oracle to compute the time-weighted average price tick. Our analysis shows the use of average price tick may be improved.

In the following, we show the implementation of the related `_getDynamicFee()` function. The time-weighted average price tick is computed as `twAvgTick = int24((tickCumulatives[1] - tickCumulatives[0])/ _secondsAgo)` (line 189), which may be optimized to round to negative infinity if `tickCumulatives[1] < tickCumulatives[0]`. In fact, an example use can be found in the `periphery/libraries/OracleLibrary` contract.

```
177     function _getDynamicFee(address _pool, uint256 _scalingFactor) internal view returns (uint256) {
178         (, int24 currentTick,, uint16 observationCardinality,,) = ICLPool(_pool).slot0();
179         uint32 _secondsAgo = secondsAgo;
180 
181         if (observationCardinality < _secondsAgo / MIN_SECONDS_AGO) return 0;
182 
183         uint32[] memory sa = new uint32[](2);
184         sa[0] = _secondsAgo;
185         // sa[1] = 0; default is 0
186 
187         int24 twAvgTick;
188         try ICLPool(_pool).observe(sa) returns (int56[] memory tickCumulatives, uint160[] memory) {
```

```
            twAvgTick = int24((tickCumulatives[1] - tickCumulatives[0]) / _secondsAgo);
        } catch {
            return 0;
        }

        int24 tickDelta = currentTick - twAvgTick;
        uint24 absTickDelta = tickDelta < 0 ? uint24(-tickDelta) : uint24(tickDelta);
        return absTickDelta * _scalingFactor / SCALING_PRECISION;
    }
```

Listing 3.11: `DynamicSwapFeeModule::_getDynamicFee()`

Recommendation Improve the above `_getDynamicFee()` function to better make use of the time-weighted average price tick.

Status This issue has been resolved as it follows the original design of `Aerodrome`.

3.9 Improper estimateAmount0/1() Logic in SugarHelper

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SugarHelper`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Hybra` protocol has a `SugarHelper` contract that exposes on-chain helpers for liquidity-related mathematics formulas. Our analysis shows that two reported functions `estimateAmount0()` and `estimateAmount1()` should be improved.

To elaborate, we use `estimateAmount0()` as an example and show below its implementation. This function is used to compute the amount of `token0` for a given amount of `token1` and the intended price range for liquidity addition. It comes to our attention that the built-in sanity check of `sqrtRatioX96 <= sqrtRatioAX96 && sqrtRatioX96 >= sqrtRatioBX96` (line 68) should be revised as `sqrtRatioX96 <= sqrtRatioAX96 || sqrtRatioX96 >= sqrtRatioBX96`. And the final amount is currently computed as `amount0 = SqrtPriceMath.getAmount0Delta(sqrtRatioX96, sqrtRatioBX96, liquidity, false)`, which is better adjusted as `amount0 = SqrtPriceMath.getAmount0Delta(sqrtRatioX96, sqrtRatioBX96, liquidity, true)` for better prevision. Note the `estimateAmount1()` function shares the same issue.

```
57    function estimateAmount0(uint256 amount1, address pool, uint160 sqrtRatioX96, int24
          tickLow, int24 tickHigh)
58        external
59        view
```

```
        override
        returns (uint256 amount0)
    {
        uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(tickLow);
        uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(tickHigh);

        if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) = (
            sqrtRatioBX96, sqrtRatioAX96);

        if (sqrtRatioX96 <= sqrtRatioAX96 && sqrtRatioX96 >= sqrtRatioBX96) {
            return 0;
        }

        // @dev If a pool is provided, fetch updated `sqrtPriceX96`
        if (pool != address(0)) {
            (sqrtRatioX96,,,,,) = ICLPool(pool).slot0();
        }
        uint128 liquidity = LiquidityAmounts.getLiquidityForAmount1(sqrtRatioAX96,
            sqrtRatioX96, amount1);
        amount0 = SqrtPriceMath.getAmount0Delta(sqrtRatioX96, sqrtRatioBX96, liquidity,
            false);
    }
```

Listing 3.12: `SugarHelper::estimateAmount0()`

Recommendation Revise the above-mentioned routines to properly compute the intended token amounts for liquidity addition.

Status This issue has been resolved as this contract is currently unused.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Hybra Finance protocol, which is the public liquidity layer on HyperliquidX, fully community-owned with no insider allocations. It features an upgraded ve(3,3) flywheel with in-house, highly competitive dynamic-fee CL pools and introduces G33, a PVP-style mechanism that rewards loyal LPs with higher yields while reducing rewards for short-term capital — reinforcing long-term alignment and making liquidity incentives more sustainable. 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-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-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.