SMART CONTRACT AUDIT REPORT

for

HyperFlash

Prepared By: Xiaomi Huang

PeckShield

February 25, 2025

Document Properties

Client	HyperFlash
Title	Smart Contract Audit Report
Target	HyperFlash
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	February 25, 2025	Xuxian Jiang	Final Release
1.0-rc1	February 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 **4**
1.1 About HyperFlash . . . 4
1.2 About PeckShield . . . 5
1.3 Methodology . . . 5
1.4 Disclaimer . . . 7

2 Findings **9**
2.1 Summary . . . 9
2.2 Key Findings . . . 10

3 Detailed Results **11**
3.1 Improved Constructor Logic in Hyperflash . . . 11
3.2 Improved Ether Transfer With Necessary Reentrancy Guard . . . 12
3.3 Trust Issue of Admin Keys . . . 13

4 Conclusion **15**

References **16**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `HyperFlash` 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 HyperFlash

The `HyperFlash` protocol is a next-generation staking protocol on `HyperEVM` that combines `Liquid Staking Tokens (LSTs)` with `Maximum Extractable Value (MEV)` strategies. Its core purpose is to let `HYPE` token holders stake their tokens and receive a liquid staking token (called `flashHYPE`) while capturing additional yield from `MEV` opportunities. The basic information of HyperFlash is as follows:

Table 1.1: Basic Information of HyperFlash

Item	Description
Name	HyperFlash
Type	Ethereum Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 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/nreddy-eterna/flashhype-audit.git (752a481)

1.2 About PeckShield

PeckShield Inc. [7] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 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:

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

- 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) [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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `HyperFlash` 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 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	1	🟦
Low	2	🟦 🟦
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key HyperFlash Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor Logic in Hyperflash	Coding Practices	Confirmed
PVE-002	Low	Improved Ether Transfer With Necessary Reentrancy Guard	Coding Practices	Confirmed
PVE-003	Medium	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Constructor Logic in Hyperflash

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Hyperflash
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the Hyperflash contract is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we show its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., _disableInitializers ();. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the initialize() function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function initialize(address _feeAddress, address _delegator) public initializer {
        __ERC20_init("flashHYPE", "fHYPE");
        __Ownable_init();
        require(_feeAddress != address(0), "Fee address cannot be zero");
        require(_delegator != address(0), "Delegator address cannot be zero");
        feeAddress = _feeAddress;
        delegator = _delegator;
    }
```

Listing 3.1: Hyperflash::initialize()

Recommendation Improve the above-mentioned constructor routine in the Hyperflash contract.

Status This issue has been confirmed.

3.2 Improved Ether Transfer With Necessary Reentrancy Guard

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Hyperflash`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

Description

The `HyperFlash` protocol allows `HYPE` token holders to stake their tokens and receive a liquid staking token (called `flashHYPE`) while capturing additional yield from `MEV` opportunities. In the process of examining the staking/unstaking mechanism, we notice a possible issue that may arise from the native coin transfer.

To elaborate, we show below the code snippet of the related `unstake()` routine, which allows to unstake to withdraw native coin after fee deduction (e.g., `Ether`). We notice that this routine directly calls the native `transfer()` routine (lines 70 – 71) to transfer `Ether`. However, the `transfer()` is not recommended to use any more since the `EIP-1884` may increase the gas cost and the 2300 gas limit may be exceeded. There is a helpful blog stop-using-soliditys-transfer-now that explains why the `transfer()` is not recommended any more.

```
54      function unstake(uint256 flashHYPEAmount) public {
55          require(flashHYPEAmount > 0, "Cannot unstake 0 flashHYPE");
56          require(balanceOf(msg.sender) >= flashHYPEAmount, "Insufficient flashHYPE
                balance");
57          require(feeAddress != address(0), "Fee address not set");

59          uint256 totalFlashHYPE = totalSupply();
60          uint256 hypeToReturn = (flashHYPEAmount * totalHYPE) / totalFlashHYPE;
61          uint256 fee = (hypeToReturn * 10) / 10000; // 0.10%
62          uint256 netHYPEToUser = hypeToReturn - fee;

64          _burn(msg.sender, flashHYPEAmount);
65          require(totalSupply() + flashHYPEAmount == totalFlashHYPE, "Burn failed");
66          totalHYPE -= hypeToReturn;

68          if (address(this).balance >= hypeToReturn) {
69              // There's enough flashHYPE in the contract to pay out the user
70              payable(msg.sender).transfer(netHYPEToUser);
71              payable(feeAddress).transfer(fee);

73              emit Unstaked(msg.sender, flashHYPEAmount, netHYPEToUser, fee);
74          } else {
```

```
            // Not enough flashHYPE in the contract to pay out the user
            debt[msg.sender] += netHYPEToUser;
            unresolvedDebt += netHYPEToUser;
            emit DebtRecorded(msg.sender, netHYPEToUser);
        }
    }
```

Listing 3.2: `Hyperflash::unstake()`

Recommendation Revisit the above-mentioned routine to transfer `ETH` using `call()`. Note it also affects other two routines, i.e., `delegate()` and `resolve()`.

Status This issue has been confirmed.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Hyperflash`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Hyperflash` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., delegate funds, add yield, and upgrade proxy). It also has the privilege to control or govern the flow of assets managed by this protocol. 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.

```
    // Add yield to the contract
    function addYield() external payable onlyOwner {
        require(msg.value > 0, "Cannot add 0 HYPE as yield");
        totalHYPE += msg.value;
    }

    // Delegate HYPE to the delegator address (a multi-sig that handles the delegation
        on the native chain)
    function delegate(uint256 amount) external onlyOwner {
        require(amount <= address(this).balance, "Not enough HYPE");
        payable(delegator).transfer(amount);
    }
```

Listing 3.3: Example Privileged Functions in `Hyperflash`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, 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.

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

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as a multisig account will be used to hold the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `HyperFlash` protocol, which is a next-generation staking protocol on `HyperEVM` that combines `Liquid Staking Tokens (LSTs)` with `Maximum Extractable Value (MEV)` strategies. Its core purpose is to let `HYPE` token holders stake their tokens and receive a liquid staking token (called `flashHYPE`) while capturing additional yield from `MEV` opportunities. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.