Smart Contract
Security Audit Report

Prepared for Hyperpie

Prepared by Supremacy

March 11, 2025

Contents

1 Introduction 3
1.1 About Client 4
1.2 Audit Scope 4
1.3 Changelogs 4
1.4 About Us 4
1.5 Terminology 5
2 Findings 6
2.1 Low 7
2.2 Informational 9
3 Disclaimer 12

1 Introduction

Given the opportunity to review the design document and related codebase of the Hyperpie, we outline in the report our systematic approach to evaluate potential security issues in the smart contract(s) implementation, 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 Client

Magpie XYZ is an ecosystem of DeFi protocols that provide yield and veTokenomics boosting services across multiple blockchain networks.

Hyperpie is an advanced liquid staking solution designed for Hyperliquid users. It enables HYPE token staking while maintaining asset flexibility.

Item	Description
Client	Magpiexyz
Project	Hyperpie
Type	Smart Contract
Languages	Solidity
Platform	EVM-compatible

1.2 Audit Scope

In the following, we show the Git repository of reviewed file and the commit hash used in this security audit:

Version	Repository	Commit Hash
1	Hyperpie	`ba87f502f42df10a9275435a36ae78e6da61c2cf`
2	Hyperpie	`392ba80072507b9c51e56ea876da4d09903ce540`
3	Hyperpie	`d9d3ffc3a96cc0d4a865cf70fd819b9f00d0fe98`

1.3 Changelogs

Version	Date	Description
0.1	March 02, 2025	Initial Draft
1.0	March 11, 2025	Final Release

1.4 About Us

Supremacy is a leading blockchain security firm, composed of industry hackers and academic researchers, provide top-notch security solutions through our technology precipitation and innovative research.

We are reachable at X (https://x.com/SupremacyHQ), or Email (contact@supremacy.email).

1.5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- Likelihood represents the likelihood of a finding to be triggered or exploited in practice
- Impact specifies the technical and business-related consequences of a finding
- Severity is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Severity

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

Likelihood

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

2 Findings

The table below summarizes the findings of the audit, including status and severity details.

ID	Severity	Description	Status
1	Low	Insufficient Approval Check	Confirmed
2	Low	Withdrawal Buffer Management	Confirmed
3	Informational	Potential Exchange Rate Manipulation	Acknowledged
4	Informational	Lack of Minimum Deposit Check	Fixed
5	Informational	Follow Check-Effects-Interactions Pattern	Acknowledged

2.1 Low

1. Insufficient Approval Check [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Confirmed`

Description

The `queueWithdrawal()` function uses `safeTransferFrom` to move `mHYPE` but doesn't explicitly check if the user has approved the contract to spend the required amount. If the user hasn't approved enough `mHYPE`, the transaction will revert unnecessarily, wasting gas and degrading user experience.

```
154     /*//////////////////////////////////////////////////////////////
155                                 EXTERNAL FUNCTIONS
156     //////////////////////////////////////////////////////////////*/
157     function queueWithdrawal(uint256 mHYPEAmount) external whenNotPaused
    nonReentrant {
158         if (mHYPEAmount == 0) revert InvalidAmount();
159 
160         address mHYPE =
    hyperpieConfig.getAddress(HyperpieConstants.MHYPE_TOKEN);
161         IERC20 mHYPEToken = IERC20(mHYPE);
162         if (mHYPEToken.balanceOf(msg.sender) < mHYPEAmount) {
163             revert InsufficientBalance();
164         }
165 
166         // Transfer mHYPE tokens first
167         mHYPEToken.safeTransferFrom(msg.sender, address(this), mHYPEAmount);
168 
169         uint256 hypeAmount = (mHYPEAmount *
    ImHYPE(mHYPE).exchangeRateToUnderlying()) / 1 ether;
170         uint256 availableToWithdraw = getAvailableToWithdraw();
171 
172         // Increment the withdrawRequestId
173         withdrawRequestId++;
174 
175         WithdrawalRequest memory request = WithdrawalRequest({
176             owner: msg.sender,
177             requestId: withdrawRequestId,
178             mHYPEAmount: mHYPEAmount,
179             hypeAmount: hypeAmount,
180             requestTime: block.timestamp
181         });
182 
183         bool queued = false;
184         if (hypeAmount > availableToWithdraw) {
185             // Queue the withdrawal
186             claimReserve += availableToWithdraw;
187             withdrawQueue.queuedWithdrawFilled += availableToWithdraw;
188             withdrawQueue.queuedWithdrawToFill += hypeAmount;
189 
190             bytes32 withdrawHash = keccak256(abi.encode(request, msg.sender));
191             withdrawQueued[withdrawHash].queued = true;
192             withdrawQueued[withdrawHash].fillAt =
    withdrawQueue.queuedWithdrawToFill;
193             queued = true;
```

```
194         } else {
195             // For instant withdrawals from buffer
196             claimReserve += hypeAmount;
197         }
198 
199         withdrawRequests[msg.sender].push(request);
200 
201         emit WithdrawalQueued(msg.sender, withdrawRequestId, mHYPEAmount,
    hypeAmount, queued, availableToWithdraw);
202     }
```

HyperpieWithdrawManager.sol

Recommendation

Consider add an explicit approval check.

```
166         // Transfer mHYPE tokens first
167         if (mHYPEToken.allowance(msg.sender, address(this)) < mHYPEAmount) {
168           revert InsufficientAllowance();
169         }
170 
171         mHYPEToken.safeTransferFrom(msg.sender, address(this), mHYPEAmount);
```

HyperpieWithdrawManager.sol

2. Withdrawal Buffer Management [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Confirmed`

Description

In `stakeHype()`, the contract transfers `HYPE` to the `withdrawManager` to cover a deficit without verifying the transfer's success beyond a basic success check. There's no assurance that the `withdrawManager` can handle or process the funds correctly. If the `withdrawManager` is misconfigured, compromised, or fails to receive funds (e.g., due to a revert), the `HYPE` could be lost or stuck.

```
70     /*//////////////////////////////////////////////////////////////
71                                WRITE FUNCTIONS
72     //////////////////////////////////////////////////////////////*/
73     /// @notice Deposits HYPE and mints mHype, note the current implementation
    is consider only for native
74     /// token deposits
75     /// @param minMHYPETokenAmount Minimum amount of mHype to accept
76     /// @param referral Address of the referrer
77     function stakeHype(uint256 minMHYPETokenAmount, address referral) external
    payable whenNotPaused nonReentrant {
78         if (msg.value == 0 || msg.value < minAmountToDeposit) {
79             revert InvalidAmountToDeposit();
80         }
81 
82         uint256 mHYPEAmount = getMHYPEToMint(msg.value);
83         if (mHYPEAmount < minMHYPETokenAmount) {
84             revert MinimumAmountToReceiveNotMet();
```

```
85          }
86
87          // Mint mHype - this will set the stake time
88          address mHYPE =
    hyperpieConfig.getAddress(HyperpieConstants.MHYPE_TOKEN);
89          IMintableERC20(mHYPE).mint(msg.sender, mHYPEAmount);
90
91          // Check withdraw buffer deficit and fill if needed
92          address withdrawManagerAddr =
    hyperpieConfig.getAddress(HyperpieConstants.HYPERPIE_WITHDRAW_MANAGER);
93          IHyperpieWithdrawManager withdrawManager =
    IHyperpieWithdrawManager(withdrawManagerAddr);
94          uint256 withdrawDeficit = withdrawManager.getWithdrawDeficit();
95
96          if (withdrawDeficit > 0) {
97              uint256 amountToBuffer = withdrawDeficit > msg.value ? msg.value :
    withdrawDeficit;
98              // Send HYPE directly to the withdraw manager to fill the buffer
99              (bool success,) = withdrawManagerAddr.call{ value: amountToBuffer }
    ("");
100             if (!success) revert TransferFailed();
101         }
102
103         emit AssetDeposited(msg.sender,
    HyperpieConstants.PLATFORM_TOKEN_ADDRESS, msg.value, mHYPEAmount, referral);
104     }
```

HyperpieStaking.sol

Recommendation

Always validate the `withdrawManagerAddr` balance after transferring funds.

```
96          if (withdrawDeficit > 0) {
97              uint256 amountToBuffer = withdrawDeficit > msg.value ? msg.value :
    withdrawDeficit;
98              // Send HYPE directly to the withdraw manager to fill the buffer
99              uint256 balanceBefore = withdrawManagerAddr.balance;
100             (bool success,) = withdrawManagerAddr.call{ value: amountToBuffer }
    ("");
101             require(success && withdrawManagerAddr.balance >= balanceBefore +
    amountToBuffer);
102
```

HyperpieStaking.sol

2.2 Informational

3. Potential Exchange Rate Manipulation [Informational]

`Status: Acknowledged`

Description

The `getMHYPEToMint()` function calculates `mHYPE` to mint based on an exchange rate from `ImHYPE(mHYPE).exchangeRateToUnderlying()`. If this rate is manipulable (e.g., by an external

oracle) or not updated correctly, it could distort the `mHYPE` issuance. Users could receive more or fewer `mHYPE` tokens than expected, leading to unexpected results.

```
58      /*////////////////////////////////////////////////////////////////
59                              VIEW FUNCTIONS
60      ////////////////////////////////////////////////////////////////*/
61      /// @notice Calculates the amount of mHype to mint for HYPE deposit
62      /// @param amount Amount of HYPE being deposited
63      /// @return mHYPEAmount Amount of mHype to mint
64      function getMHYPEToMint(uint256 amount) public view returns (uint256
    mHYPEAmount) {
65          address mHYPE =
    hyperpieConfig.getAddress(HyperpieConstants.MHYPE_TOKEN);
66          uint256 rate = ImHYPE(mHYPE).exchangeRateToUnderlying();
67          return (amount * 1 ether) / rate;
68      }
```

HyperpieStaking.sol

Recommendation

Consider add bounds checking to prevent extreme rate values.

4. Lack of Minimum Deposit Check [Informational]

`Status: Fixed`

Description

While `stakeHype` checks `msg.value` against `minAmountToDeposit`, there's no validation in `setMinAmountToDeposit` to prevent setting it to zero. A zero minimum allows spam deposits, potentially clogging the contract with low-value transactions and affecting performance.

```
122     /// @notice Sets the minimum amount required for deposits
123     /// @param _minAmountToDeposit New minimum deposit amount
124     function setMinAmountToDeposit(uint256 _minAmountToDeposit) external
    onlyDefaultAdmin {
125         minAmountToDeposit = _minAmountToDeposit;
126         emit MinAmountToDepositUpdated(_minAmountToDeposit);
127     }
```

HyperpieStaking.sol

Recommendation

Consider add a check in `setMinAmountToDeposit()`.

Feedback: Fixed in `392ba80`.

5. Follow Check-Effects-Interactions Pattern [Informational]

`Status: Acknowledged`

Description

In the `HyperpieStaking::stakeHype()`, the minting of `mHYPE` does not follow the Check-Effects-Interactions Pattern.

```
70      /*////////////////////////////////////////////////////////////////
71                                WRITE FUNCTIONS
72      ////////////////////////////////////////////////////////////////*/
73      /// @notice Deposits HYPE and mints mHype, note the current implementation
    is consider only for native
74      /// token deposits
75      /// @param minMHYPETokenAmount Minimum amount of mHype to accept
76      /// @param referral Address of the referrer
77      function stakeHype(uint256 minMHYPETokenAmount, address referral) external
    payable whenNotPaused nonReentrant {
78          if (msg.value == 0 || msg.value < minAmountToDeposit) {
79              revert InvalidAmountToDeposit();
80          }
81
82          uint256 mHYPEAmount = getMHYPEToMint(msg.value);
83          if (mHYPEAmount < minMHYPETokenAmount) {
84              revert MinimumAmountToReceiveNotMet();
85          }
86
87          // Mint mHype - this will set the stake time
88          address mHYPE =
    hyperpieConfig.getAddress(HyperpieConstants.MHYPE_TOKEN);
89          IMintableERC20(mHYPE).mint(msg.sender, mHYPEAmount);
90
91          // Check withdraw buffer deficit and fill if needed
92          address withdrawManagerAddr =
    hyperpieConfig.getAddress(HyperpieConstants.HYPERPIE_WITHDRAW_MANAGER);
93          IHyperpieWithdrawManager withdrawManager =
    IHyperpieWithdrawManager(withdrawManagerAddr);
94          uint256 withdrawDeficit = withdrawManager.getWithdrawDeficit();
95
96          if (withdrawDeficit > 0) {
97              uint256 amountToBuffer = withdrawDeficit > msg.value ? msg.value :
    withdrawDeficit;
98              // Send HYPE directly to the withdraw manager to fill the buffer
99              (bool success,) = withdrawManagerAddr.call{ value: amountToBuffer }
    ("");
100             if (!success) revert TransferFailed();
101         }
102
103         emit AssetDeposited(msg.sender,
    HyperpieConstants.PLATFORM_TOKEN_ADDRESS, msg.value, mHYPEAmount, referral);
104     }
```

HyperpieStaking.sol

Recommendation

Revise the code logic accordingly.

3 Disclaimer

This security audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. This security audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues, also cannot make guarantees about any additional code added to the assessed project after the audit version. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contract(s). Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.