SMART CONTRACT AUDIT REPORT

for

Helio

Prepared By: Xiaomi Huang

PeckShield

May 25, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 25, 2022	Xuxian Jiang	Final Release
1.0-rc2	May 10, 2022	Xuxian Jiang	Release Candidate #2
1.0-rc1	May 6, 2022	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 smart contract source code of the Helio 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 audited protocol 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 Helio

The Helio protocol is implemented as a set of smart contracts with part of the logic relying on the MakerDAO smart contract set. In particular, it mints the stablecoin HAY with necessary collaterization. It also builds new modules, including the Earn module to allow users to stake HAY for accrued interest, as well as the Rewards module to claim rewards in HELIO. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Helio

Item	Description
Name	Helio
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 25, 2022

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

- https://github.com/helio-money/helio-smart-contracts (d543f08)

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

- https://github.com/helio-money/helio-smart-contracts (831b06c)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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) [9], 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.

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 `Helio` implementation. 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	1	■
Medium	3	■ ■ ■
Low	3	■ ■ ■
Informational	1	■
Total	8	

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 that 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 high-severity vulnerability, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Helio Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely rewardsPool Update From Reward Changes in HelioRewards	Business Logic	Resolved
PVE-002	Medium	Incorrect pendingRewards() Logic in HelioRewards	Business Logic	Resolved
PVE-003	High	Potential Flashloan/MEV For Increased Rewards	Time And State	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Informational	Improved ERC20 Compliance Of Jar	Coding Practices	Resolved
PVE-007	Low	Proper Allowance Adjustment in CerosRouter	Coding Practices	Resolved
PVE-008	Medium	Incorrect Logic of CerosRouter::withdrawWithSlippage()	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Timely rewardsPool Update From Reward Changes in HelioRewards

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `HelioRewards`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Helio` protocol provides an incentive mechanism that rewards the `HAY` minting. The rewards are carried out by designating a single pool from which the rewards are allocated based on the user debt amount. By design, the pool allows for the configuration of a collateral-specific, per-second reward rate. Our analysis shows that the update of the reward rate makes it necessary to timely collect rewards before the new rate becomes effective.

To elaborate, we show below the related two functions: `initPool()` and `setRate()`. The first function allows to initialize the pool while the second one is used to change the reward rate. It comes to our attention that the first function needs to validate whether it can only be invoked once while the second function needs to timely collect the overall pool reward (by calling `drip()` within the same contract) before applying the new reward rate.

```
    function initPool(bytes32 ilk, uint256 rate) external auth {
        ilks[ilk] = Ilk(rate, block.timestamp);
        poolIlk = ilk;
    }

    function setHelioToken(address helioToken_) external auth {
        helioToken = helioToken_;
    }

    function setRate(bytes32 ilk, uint256 newRate) external auth {
```

```
        Ilk storage pool = ilks[ilk];
        pool.rewardRate = newRate;
    }
```

Listing 3.1: `HelioRewards::initPool()/setRate()`

In the same vein, the protocol has another contract `DAOInteraction` that facilitates the user interactions. Our analysis shows that the interest accrual may need to be performed before the intended borrow/repay is executed.

```
    function borrow(address token, uint256 usbAmount) external returns(uint256) {
        CollateralType memory collateralType = collaterals[token];
        require(collateralType.live == 1, "Interaction/inactive collateral");

        (, uint256 rate,,,) = vat.ilks(collateralType.ilk);
        uint256 dart = (usbAmount * 10 ** 27) / rate;
        vat.frob(collateralType.ilk, msg.sender, msg.sender, msg.sender, 0, int256(dart));
        vat.move(msg.sender, address(this), usbAmount * 10**27);
        usbJoin.exit(msg.sender, usbAmount);

        drip(token);
        emit Borrow(msg.sender, usbAmount);
        return dart;
    }
```

Listing 3.2: `DAOInteraction::borrow()`

Recommendation Timely invoke `drip()` when the pool's reward rate is updated.

Status This issue has been fixed in the following commit: 472e83ce.

3.2 Incorrect pendingRewards() Logic in HelioRewards

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HelioRewards`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `Helio` protocol provides an incentive mechanism that rewards the `HAY` minting. The rewarding logic is to distribute the rewards pro-rata based on the minted `HAY` debt amount. While reviewing the related reward logic, we notice the current implementation needs to be corrected.

To elaborate, we show below the related `pendingRewards()` function. As the name indicates, this function is used to calculate the pending reward of the given user. It comes to our attention the denominator used for the share calculation is `totalDebt` while the numerator is `rate*art`. Note that `totalDebt` is the normalized total debt amount and the user's normalized debt amount is `art`, not the de-normalized one `rate*art`!

```
    function pendingRewards(address usr) public poolInit view returns(uint256) {
        (uint256 totalDebt, uint256 rate) = vat.ilks(poolIlk);
        (, uint256 art) = vat.urns(poolIlk, usr);
        uint256 usrDebt = hMath.mulDiv(art, rate, 10 ** 27);
        uint256 shares = hMath.mulDiv(usrDebt, rewardsPool, totalDebt);
        return unclaimedRewards[usr] + shares - claimedRewards[usr];
    }
```

Listing 3.3: `HelioRewards::pendingRewards()`

Recommendation Properly compute the reward share in the above `pendingRewards()` function.

Status This issue has been fixed in the following commit: 472e83ce.

3.3 Potential Flashloan/MEV For Increased Rewards

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `HelioRewards`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

In the last section, we cover a logic issue in the current rewarding logic, which basically distributes the rewards pro-rata based on the minted `HAY` debt amount. In the section, we further analyze the associated reward-claiming logic and show that the current logic may be exploited to steal rewards from others.

Specifically, we show below the related `claim()` function. This function implements a rather straightforward logic in computing the pending reward and then minting the rewards to the calling user. However, as mentioned earlier, the pending reward amount is computed based on percentage of the user debt amount among all debt amount. With that, it is possible to have a flashloan to mint a huge amount of debt amount right, then call this `claim()` function to claim rewards, and next return the flashloan by repaying back all debts!

```
    function claim(uint256 amount) external poolInit {
        require(amount <= pendingRewards(msg.sender), "Rewards/not-enough-rewards");
```

```
        if (unclaimedRewards[msg.sender] >= amount) {
            unclaimedRewards[msg.sender] -= amount;
        } else {
            uint256 diff = amount - unclaimedRewards[msg.sender];
            claimedRewards[msg.sender] = diff;
            unclaimedRewards[msg.sender] = 0;
        }
        Mintable(helioToken).mint(msg.sender, amount);

        emit Claimed(msg.sender, amount);
    }
```

Listing 3.4: `HelioRewards::claim()`

Recommendation Develop an effective mitigation to the above issue to fairly disseminate the rewards to protocol users.

Status This issue has been fixed in the following commit: 472e83ce.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DAOInteraction`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

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 `transfer()` routine and 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. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
```

```
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.5: USDT::transfer()

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer (), 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 approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

In current implementation, if we examine the DAOInteraction::deposit() routine that is designed to deposit the token as collateral. To accommodate the specific idiosyncrasy, there is a need to user safeTransferFrom(), instead of transferFrom() (line 168).

```
164     function deposit(address token, uint256 dink) external returns (uint256){
165         CollateralType memory collateralType = collaterals[token];
166         require(collateralType.live == 1, "Interaction/inactive collateral");

168         IERC20(token).transferFrom(msg.sender, address(this), dink);
169         collateralType.gem.join(msg.sender, dink);
170         vat.behalf(msg.sender, address(this));
171         vat.frob(collateralType.ilk, msg.sender, msg.sender, msg.sender, int256(dink),
            0);

173         deposits[token] += dink;

175         drip(token);

177         emit Deposit(msg.sender, dink);
178         return dink;
179     }
```

Listing 3.6: DAOInteraction::deposit()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom().

Status This issue has been fixed in the following commit: a3ca32ce.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Helio protocol, there are special administrative accounts (with the auth set). These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters and execute privileged operations). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```
    function initPool(bytes32 ilk, uint256 rate) external auth {
        ilks[ilk] = Ilk(rate, block.timestamp);
        poolIlk = ilk;
    }

    function setHelioToken(address helioToken_) external auth {
        helioToken = helioToken_;
    }

    function setRate(bytes32 ilk, uint256 newRate) external auth {
        Ilk storage pool = ilks[ilk];
        pool.rewardRate = newRate;
    }
```

Listing 3.7: Example Privileged Operations in HelioRewards

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated by the team by removing the Jar contract from being a part of core contracts, removing the permit() function, updating the flap() function in the vow contract, and sending the surplus amount to the multisig wallet if flapper is not used.

3.6 Improved ERC20 Compliance Of Jar

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Jar`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `Helio` protocol also comes with a `Jar` contract that is designed to assist the `HAY` distribution farming. In the following, we examine the ERC20 compliance of the `Jar` token contract.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓

Specifically, the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue. Specifically, the current implementation has defined the `decimals` state with the `uint` type. The ERC20 specification indicates the type of `uint8` for the `decimals` state. Note that this incompatibility issue does not necessarily affect the functionality of `Jar` in any negative way.

In addition, it should be highlighted that `Jar` serves the purpose of staking for rewards. By design, it cannot be transferred. Therefore, the related set of functions of `transfer()`, `transferFrom()`, and `approve()` are explicitly not supported!

Recommendation Revise the `Jar` implementation to improve its ERC20-compliance.

Status This issue has been fixed in the following commit: cc27700.

3.7 Proper Allowance Adjustment in CerosRouter

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CerosRouter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `Helio` protocol also comes with a `CerosRouter` contract that is designed to facilitate the user interaction. This `CerosRouter` contract requires the proper allowance setup to interact with external DEX engines. However, our analysis shows that when there is a need to update the current DEX, the allowance has not been updated accordingly.

To elaborate, we show below the `initialize()` function. It properly configures the allowance of `certToken` to `dexAddress`, `pool`, and `vault`. However, when these addresses are being updated, the allowances are not updated!

```
    function initialize(
        address certToken,
        address wBnbToken,
        address ceToken,
        address bondToken,
        address vault,
        address dexAddress,
        address pool
    ) public initializer {
        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();
        _certToken = ICertToken(certToken);
        _wBnbAddress = wBnbToken;
        _ceToken = IERC20(ceToken);
        _vault = IVault(vault);
        _dex = IDex(dexAddress);
        _pool = IBinancePool(pool);
        IERC20(wBnbToken).approve(dexAddress, type(uint256).max);
```

```
        IERC20(certToken).approve(dexAddress, type(uint256).max);
        IERC20(certToken).approve(bondToken, type(uint256).max);
        IERC20(certToken).approve(pool, type(uint256).max);
        IERC20(certToken).approve(vault, type(uint256).max);
    }
```

Listing 3.8: `CerosRouter::initialize()`

```
    function changeVault(address vault) external onlyOwner {
        _vault = IVault(vault);
        emit ChangeVault(vault);
    }

    function changeDex(address dex) external onlyOwner {
        _dex = IDex(dex);
        emit ChangeDex(dex);
    }

    function changePool(address pool) external onlyOwner {
        _pool = IBinancePool(pool);
        emit ChangePool(pool);
    }
```

Listing 3.9: `CerosRouter::changeVault()/changeDex()/changePool()`

Recommendation Update the allowance accordingly when the intended DEX is updated. Note this issue is also applicable to `_vault` and `_pool`.

Status This issue has been fixed in the following commit: 831b06c.

3.8 Incorrect Logic of CerosRouter::withdrawWithSlippage()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CerosRouter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `Helio` protocol also comes with a `CerosRouter` contract that is designed to facilitate the user interaction. While analyzing this `CerosRouter` contract, we notice a key funtion `withdrawWithSlippage()` contains an incorrect implementation.

To elaborate, we show below its implementation. It has a rather straightforward logic in withdrawing the `_certToken` from the vault and then swapping to `WBNB`. However, the swap operation still uses the (old) input `amount` for the vault amount, instead of the (new) amount – `realAmount` – after the withdrawal from the vault.

```
    function withdrawWithSlippage(
        address recipient,
        uint256 amount,
        uint256 outAmount
    ) external override nonReentrant returns (uint256) {
        uint256 realAmount = _vault.withdrawFor(
            msg.sender,
            address(this),
            amount
        );
        address[] memory path = new address[](2);
        path[0] = address(_certToken);
        path[1] = _wBnbAddress;
        uint256[] memory amounts = _dex.swapExactTokensForETH(
            amount,
            outAmount,
            path,
            recipient,
            block.timestamp + 300
        );
        emit Withdrawal(msg.sender, recipient, _wBnbAddress, amounts[1]);
        return amounts[1];
    }
```

Listing 3.10: `CerosRouter::withdrawWithSlippage()`

Recommendation Revise the above `withdrawWithSlippage()` logic to make use of the right amount for token conversion.

Status This issue has been fixed in the following commit: 831b06c.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Helio` protocol, which is implemented as a set of smart contracts with part of the logic relying on `MakerDAO`. In particular, it mints the stablecoin `HAY` with necessary collaterization with extensions of new modules, including the `Earn` module to allow users to stake `HAY` for accrued interest, as well as the `Rewards` module to claim rewards in `HELIO`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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.

PeckShield

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

PeckShield

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

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

PeckShield