SMART CONTRACT AUDIT REPORT

for

Helio

Prepared By: Xiaomi Huang

PeckShield

August 16, 2023

Document Properties

Client	Helio
Title	Smart Contract Audit Report
Target	Helio
Version	2.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
2.0	August 16, 2023	Xuxian Jiang	Final Release
2.0-rc1	July 6, 2023	Luck Hu	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-like functionalities. The new Helio protocol introduces the integration for liquid staking BNB tokens (BNBx, stkBNB, and SnBNB), which simplifies users experience in depositing and withdrawing BNBs. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Helio

Item	Description
Name	Helio
Website	https://helio.money/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2023

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.git (6a2a7c3)

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.git (ac47ebc)

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 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- 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.

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	1	■
High	3	■ ■ ■
Medium	3	■ ■ ■
Low	1	■
Undetermined	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, 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 critical-severity vulnerability, 3 high-severity vulnerabilities, 3 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 undetermined issue.

Table 2.1: Key Helio Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Revisited Withdrawal of Collateral in Interaction::withdraw()	Business Logic	Confirmed
PVE-002	High	Incorrect Logic in MasterVault::withdrawInTokenFromStrategy()	Business Logic	Fixed
PVE-003	High	Revised BNB Withdrawal in StkBnbStrategy::_withdraw()	Business Logic	Fixed
PVE-004	Medium	Improved Debt Maintenance for StkBnbStrategy	Business Logic	Fixed
PVE-005	High	Potential Denial-of-Service in Withdrawal from MasterVault	Business Logic	Fixed
PVE-006	Medium	Suggested Strategy Validation in MasterVault Deposit	Coding Practices	Fixed
PVE-007	Low	Improved Withdrawal Validation in CerosYieldConverterStrategy::_withdraw()	Coding Practices	Fixed
PVE-008	Critical	Lack of Access Control to CerosYieldConverterStrategy::withdrawInToken()	Security Features	Fixed
PVE-009	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 Revisited Withdrawal of Collateral in Interaction::withdraw()

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Interaction`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Helio` protocol, the `Interaction` contract services as a proxy that is designed to facilitate user deposit/withdraw/borrow/payback. While examining the collateral withdrawal logic, we notice there is a lack of deducting the withdrawal amount from the user's `gem` balance.

To elaborate, we show below the related two functions: `Interaction::withdraw()` and `GemJoin::exit()`. As the name indicates, the first function is used to withdraw collateral from the protocol. It calls the second function to exit the `GemJoin` adapter which deducts the desired amount from the `gem` balance of the caller (line 115), i.e., the `Interaction` contract, and transfers the collateral to the user (line 116). It comes to our attention that it does not properly move the desired amount of the `gem` balance from the user to the `Interaction` contract. As a result, the user can repeat to withdraw the collateral until the `gem` balance of the `Interaction` contract is exhausted.

```
    function withdraw(
        address participant,
        address token,
        uint256 dink
    ) external nonReentrant returns (uint256) {
        CollateralType memory collateralType = collaterals[token];
        _checkIsLive(collateralType.live);
        if (helioProviders[token] != address(0)) {...}

        uint256 unlocked = free(token, participant);
```

```
        if (unlocked < dink) {
            int256 diff = int256(dink) - int256(unlocked);
            vat.frob(collateralType.ilk, participant, participant, participant, - diff,
                0);
            vat.flux(collateralType.ilk, participant, address(this), uint256(diff));
        }
        // Collateral is actually transferred back to user inside ‘exit‘ operation.
        // See GemJoin.exit()
        collateralType.gem.exit(msg.sender, dink);
        deposits[token] -= dink;

        emit Withdraw(participant, dink);
        return dink;
    }
```

Listing 3.1: `Interaction::withdraw()`

```
    function exit(address usr, uint wad) external auth {
        require(wad <= (2 ** 255) - 1, "GemJoin/overflow");
        vat.slip(ilk, msg.sender, -int(wad));
        require(gem.transfer(usr, wad), "GemJoin/failed-transfer");
        emit Exit(usr, wad);
    }
```

Listing 3.2: `GemJoin::exit()`

Recommendation Properly move the desired amount of the `gem` balance from the user to the `Interaction` contract.

Status The team confirmed this is not an issue as the free balance of the participant is always 0.

3.2 Incorrect Logic in MasterVault::withdrawInTokenFromStrategy()

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

Description

The `Helio` protocol introduces the integration for liquid staking `BNB` tokens (e.g., `BNBx`, `stkBNB`, and `SnBNB`), and provides flexible withdrawal options for users. While examining the withdrawal logic of the liquid staking `BNB` tokens, we notice a logic issue where the withdraw fee is charged twice.

To elaborate, we show below the related `MasterVault::withdrawInTokenFromStrategy()/_assessFee()` routines. As the name indicates, the `MasterVault::withdrawInTokenFromStrategy()` routine is used to withdraw in `LSD` tokens. If the withdraw fee ratio is set, the withdraw fee is deducted from the withdrawal amount by calling the `_assessFee()` routine (line 247). Then both the fee and the remaining withdrawal amount are withdrawn from the strategy to the related receivers in `LSD` tokens. (lines 248 – 249).

However, we notice the withdraw fee is also added the `feeEarned` state in the `_assessFee()` routine (line 259). As a result, the withdraw fee is taken twice in a withdraw operation: one in `BNB` and another in `LSD` token. Our analysis shows that it should only take the withdraw fee in `LSD` token in the `MasterVault::withdrawInTokenFromStrategy()` routine.

```
236 function withdrawInTokenFromStrategy(address strategy, address recipient, uint256
        amount)
237 external
238 override
239 nonReentrant
240 whenNotPaused
241 onlyProvider returns(uint256) {
242     require(amount > 0, "invalid withdrawal amount");
243     require(strategyParams[strategy].debt >= amount, "insufficient assets in strategy")
            ;
244     address src = msg.sender;
245     ICertToken(vaultToken).burn(src, amount);
246     if (withdrawalFee > 0) {
247         uint256 shares = _assessFee(amount, withdrawalFee);
248         _withdrawInTokenFromStrategy(strategy, recipient, shares);
249         _withdrawInTokenFromStrategy(strategy, feeReceiver, amount - shares);
250         return shares;
251     }
252     _withdrawInTokenFromStrategy(strategy, recipient, amount);
253 }
254
255 function _assessFee(uint256 amount, uint256 fees) private returns(uint256 value) {
256     if(fees > 0) {
257         uint256 fee = (amount * fees) / 1e6;
258         value = amount - fee;
259         feeEarned += fee;
260     } else {
261         return amount;
262     }
263 }
```

Listing 3.3: `MasterVault::withdrawInTokenFromStrategy()/_assessFee()`

What's more, it desires a return value for the received `LSD` token amount in the `MasterVault::withdrawInTokenFromStrategy()` routine. However, there is no return statement when the withdraw fee is not set (line 252), and the returned `shares` is in `BNB`, not `LSD`, when the withdraw fee is set (line 250).

Recommendation Revise the `MasterVault::withdrawInTokenFromStrategy()` routine to take the withdraw fee in `LSD` token and return the received `LSD` token amount.

Status This issue has been fixed in the following commits: c4242d3 and ac47ebc.

3.3 Revised BNB Withdrawal in StkBnbStrategy::_withdraw()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `StkBnbStrategy`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Helio` protocol, the `StkBnbStrategy` contract supports the staking of `BNB` to `pStake` and receives `stkBNB` in return. When a user withdraws `BNB` from the strategy, it sends `BNB` to the user directly if there is available `BNB` in the strategy. Otherwise, it sends the withdraw request to `pStake` and claims the `BNB` after the cool-down period. While examining the `BNB` withdrawal from the strategy, we notice it does not correctly calculate the available `BNB` amount in the strategy and it does not correctly update the recorded `BNB` amount, i.e., `_bnbDepositsInStakePool`, that the strategy has deposited in `pStake`.

In the following, we show the code snippet of the `StkBnbStrategy::_withdraw()` routine. As the name indicates, it is used to withdraw `BNB` from the strategy. It first uses the available `BNB` in the strategy to fulfill part or all of the withdrawal. However, we notice it directly counts the `BNB` balance as the available `BNB` for withdrawal (line 183), which may contain the claimed `BNB` from `pStake` that will be distributed to the withdrawals after cool-down. As a result, the new withdraw request may be fulfilled right now with the claimed `BNB` from `pStake`, hence it does not have enough `BNB` to fulfill the withdrawals after cool-down. Based on this, we suggest to deduct the claimed `BNB` from the `BNB` balance in the strategy as the available `BNB` to fulfill new withdraw request.

```
180     function _withdraw(address recipient, uint256 amount) internal returns (uint256) {
181         require(amount > 0, "invalid amount");
182 
183         uint256 ethBalance = address(this).balance;
184         if (amount <= ethBalance) {
185             (bool sent, /*memory data*/) = recipient.call{ gas: 5000, value: amount }(""
                );
186             require(sent, "!sent");
187             return amount;
188         }
189 
```

```
        // otherwise, need to send all the balance of this strategy and also need to
            withdraw from the StakePool
        (bool sent, /*memory data*/) = recipient.call{ gas: 5000, value: ethBalance }(""
            );//Luck: ethBalance > 0?
        require(sent, "!sent");
        amount -= ethBalance;

        // TODO(pSTAKE):
        // 1. There should be a utility function in our StakePool that should tell how
            much stkBNB to withdraw if I want
        //    `x` amount of BNB back, taking care of the withdrawal fee that is involved
            .
        // 2. We should also have something that takes care of withdrawing to a
            recipient, and not to the msg.sender
        // For now, the implementation here works, but can be improved in future with
            above two points.
        IStakePool stakePool = IStakePool(_addressStore.getStakePool());
        IStakedBNBToken stkBNB = IStakedBNBToken(_addressStore.getStkBNB());

        // reverse the BNB amount calculation from StakePool to get the stkBNB to burn
        ExchangeRate.Data memory exchangeRate = stakePool.exchangeRate();
        uint256 poolTokensToBurn = exchangeRate._calcPoolTokensForDeposit(amount);
        uint256 poolTokens = (poolTokensToBurn * 1e11) / (1e11 - stakePool.config().fee.
            withdraw);
        // poolTokens = the amount of stkBNB that needs to be sent to StakePool in order
             to get back `amount` BNB.

        // now, ensure that these poolTokens pass the minimum requirements for
            withdrawals set in StakePool.
        // if poolTokens < min => StakePool will reject this withdrawal with a revert =>
             okay to let this condition be handled by StakePool.
        // if poolTokens have dust => we can remove that dust here, so that withdraw can
             happen if the poolTokens > min.
        poolTokens = poolTokens - (poolTokens % stakePool.config().minTokenWithdrawal);

        // now, this amount of poolTokens might not give us exactly the `amount` BNB we
            wanted to withdraw. So, better
        // calculate that again as we need to return the BNB amount that would actually
            get withdrawn.
        uint256 poolTokensFee = (poolTokens * stakePool.config().fee.withdraw) / 1e11;
        uint256 value = exchangeRate._calcWeiWithdrawAmount(poolTokens - poolTokensFee);
        require(value <= amount, "invalid out amount");

        // initiate withdrawal of stkBNB from StakePool for this strategy
        // this assumes that this strategy holds at least the amount of stkBNB
            poolTokens that we are trying to withdraw,
        // otherwise it will revert.
        stkBNB.send(address(stakePool), poolTokens, "");

        // save it so that we can later dispatch the amount to the recipient on claim
        withdrawReqs[_endIndex++] = WithdrawRequest(recipient, value);

```

```
228         // keep track of _netDeposits in StakePool
229         _bnbDepositsInStakePool -= value;
230
231         return value + ethBalance;
232     }
```

Listing 3.4: `StkBnbStrategy::_withdraw()`

What is more, at the end of the `StkBnbStrategy::_withdraw()` routine, it updates the total `BNB` amount, i.e., `_bnbDepositsInStakePool`, that the strategy has deposited in `pStake` (line 229). The `_bnbDepositsInStakePool` is updated by deducting the `value` which is the `BNB` amount the recipient can receive from the withdrawal. However, it comes to our attention that `pStake` takes a withdraw fee from the withdrawal, so the `value` is smaller than the original withdrawal amount from `pStake` . As a result, the `_bnbDepositsInStakePool` is not updated correctly. Our analysis shows that the `_bnbDepositsInStakePool` shall be deducted by the `value` and the withdraw fee.

Recommendation Revisit the `StkBnbStrategy::_withdraw()` routine to correct the available `BNB` amount that can be used to fulfill new withdrawal and take the withdraw fee into consideration to update the `_bnbDepositsInStakePool`.

Status This issue has been fixed in the following commit: c4242d3.

3.4 Improved Debt Maintenance for StkBnbStrategy

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StkBnbStrategy`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Helio` protocol, the `StkBnbStrategy` contract accepts the deposit of `BNB` from the `MasterVault` and returns the debt amount it owns to the `MasterVault`. The `MasterVault` records the debt amount and uses it as the maximum withdrawable amount. While reviewing the deposit in `StkBnbStrategy`, we notice the recorded debt amount in the `MasterVault` may become inaccurate after the dust is deposited in `pStake`.

In the following, we show the related code snippets of the `StkBnbStrategy::depositAll()/_deposit()` routines. As the name indicates, the `_deposit()` routine is used to deposit the given amount of `BNB` to `pStake`. Specifically, if there's dust for the input amount, it directly removes it (line 115) and the dust amount is counted into the new debt amount (line 122). The dust will keep accumulating in the contract and later can be deposited via the `depositAll()` routine. After the dust is deposited to `pStake`, it returns a new debt amount. However, it comes to our attention that the new debt amount generated from depositing the dust may be smaller than the dust amount because of the deposit fee taken in `pStake`. As a result, the recorded debt amount for the dust in `MasterVault` becomes larger than it's expected after the dust is deposited to `pStake`.

Based on this, we suggest to properly update the debt amount recorded in `MasterVault` in the `depositAll()` routine.

```
    function depositAll() onlyStrategist external {
        _deposit(address(this).balance - _bnbToDistribute);
    }

    /// @dev internal function to deposit the given amount of BNB tokens into stakePool
    /// @param amount amount of BNB to deposit
    /// @return amount of BNB that this strategy owes to the master vault
    function _deposit(uint256 amount) whenDepositNotPaused internal returns (uint256) {
        IStakePool stakePool = IStakePool(_addressStore.getStakePool());
        // we don't accept dust, so just remove that. That will keep accumulating in
            this strategy contract, and later
        // can be deposited via `depositAll` (if it sums up to be more than just dust)
            OR withdrawn.
        uint256 dust = amount % stakePool.config().minBNBDeposit;
        uint256 dustFreeAmount = amount - dust;
        if (canDeposit(dustFreeAmount)) {
            stakePool.deposit{value : dustFreeAmount}(); // deposit the amount to
                stakePool in the name of this strategy
            uint256 amountDeposited = assessDepositFee(dustFreeAmount);
            _bnbDepositsInStakePool += amountDeposited; // keep track of _netDeposits in
                 StakePool

            // add dust as that is still owed to the master vault
            return amountDeposited + dust;
        }
        // the amount was so small that it couldn't be deposited to destination but it
            would remain with this strategy,
        // => strategy still owes this to the master vault
        return amount;
    }
```

Listing 3.5: StkBnbStrategy::depositAll()/_deposit()

Recommendation Properly update the debt amount recorded in `MasterVault` after the accumulated dust is deposited to `pStake` in the `depositAll()` routine.

Status This issue has been fixed in the following commit: c4242d3.

3.5 Potential Denial-of-Service in Withdrawal from MasterVault

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: MasterVault/StkBnbStrategy
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the Helio protocol, the MasterVault contract accepts the deposit of BNB from the HelioProvider and mints vaultToken to the HelioProvider in return. The MasterVault will deposit the BNB among different strategies by the manager. The HelioProvider can withdraw BNB from the MasterVault per the shares of vaultToken, and the manager can withdraw BNB from the strategies. While examining the BNB withdrawal from MasterVault and the strategies, we notice the possibility of denial-of-service that may block the withdrawal.

To elaborate, we show below the related code snippet of the MasterVault::withdrawETH() routine, which is used to withdraw BNB from the MasterVault. It first calculates the available BNB amount in the contract that can be directly used to fulfill part or all of the withdrawal (line 133). If the available BNB is not enough to fulfill the whole withdrawal, the remaining part is withdrawn from the strategies (line 138). After that, there is a validation that requires the strategies must return the desired amount of BNB (line 139).

However, our study shows that there are two cases that may violate the validation, hence reverting the withdrawal.

```
124    function withdrawETH(address account, uint256 amount)
125    external
126    override
127    nonReentrant
128    whenNotPaused
129    onlyProvider
130    returns (uint256 shares) {
131        address src = msg.sender;
132        ICertToken(vaultToken).burn(src, amount);
133        uint256 ethBalance = totalAssetInVault();
134        shares = _assessFee(amount, withdrawalFee);
135        if(ethBalance < shares) {
136            (bool sent, ) = payable(account).call{gas: 5000, value: ethBalance}("");
137            require(sent, "transfer failed");
138            uint256 withdrawn = withdrawFromActiveStrategies(account, shares -
                   ethBalance);
139            require(withdrawn == shares - ethBalance, "insufficient withdrawn amount");
140            shares = ethBalance + withdrawn;
141        } else {
```

```
            (bool sent, ) = payable(account).call{gas: 5000, value: shares}("");
            require(sent, "transfer failed");
        }
        emit Withdraw(src, src, src, amount, shares);
        return amount;
    }
```

Listing 3.6: `MasterVault::withdrawETH()`

Firstly, as the below code snippet shows, the `StkBnbStrategy::_withdraw()` routine may not return the desired amount of `BNB` because of the dust that may be deducted from the required `stkBNB` amount (line 199) or the arithmetic precision loss to calculate the required `stkBNB` amount, hence violating the above mentioned validation and reverting the withdrawal.

```
180     function _withdraw(address recipient, uint256 amount) internal returns (uint256) {
181         require(amount > 0, "invalid amount");
182 
183         uint256 ethBalance = address(this).balance;
184         if (amount <= ethBalance) {...}
185 
186         // otherwise, need to send all the balance of this strategy and also need to
                withdraw from the StakePool
187         (bool sent, /*memory data*/) = recipient.call{ gas: 5000, value: ethBalance }(""
                );//Luck: ethBalance > 0?
188         require(sent, "!sent");
189         amount -= ethBalance;
190 
191         IStakePool stakePool = IStakePool(_addressStore.getStakePool());
192         IStakedBNBToken stkBNB = IStakedBNBToken(_addressStore.getStkBNB());
193 
194         // reverse the BNB amount calculation from StakePool to get the stkBNB to burn
195         ExchangeRate.Data memory exchangeRate = stakePool.exchangeRate();
196         uint256 poolTokensToBurn = exchangeRate._calcPoolTokensForDeposit(amount);
197         uint256 poolTokens = (poolTokensToBurn * 1e11) / (1e11 - stakePool.config().fee.
                withdraw);
198 
199         poolTokens = poolTokens - (poolTokens % stakePool.config().minTokenWithdrawal);
200 
201         uint256 poolTokensFee = (poolTokens * stakePool.config().fee.withdraw) / 1e11;
202         uint256 value = exchangeRate._calcWeiWithdrawAmount(poolTokens - poolTokensFee);
203         require(value <= amount, "invalid out amount");
204 
205         // initiate withdrawal of stkBNB from StakePool for this strategy
206         stkBNB.send(address(stakePool), poolTokens, "");
207 
208         // save it so that we can later dispatch the amount to the recipient on claim
209         withdrawReqs[_endIndex++] = WithdrawRequest(recipient, value);
210 
211         // keep track of _netDeposits in StakePool
212         _bnbDepositsInStakePool -= value;
213 
214         return value + ethBalance;
```

```
215     }
```

Listing 3.7: `StkBnbStrategy::_withdraw()`

Secondly, the deposit to most strategies will return the same amount of debt as the deposited `BNB` amount. But for `StkBnbStrategy`, the debt amount may be smaller than the deposited `BNB` amount because of the deposit fee taken by `pStake`. As a result, the total debt amount may be smaller than all the deposited `BNB` amount to the strategies. In particular, if the withdraw fee taken by the vault is smaller than the difference between the total deposited `BNB` amount and the total debt amount, the withdrawal of `BNB` can't give back the desired amount of `BNB`, hence violating the above mentioned validation and reverting the withdrawal.

Moreover, in order to withdraw as many as the desired `BNB` amount from `pStake` in the `StkBnbStrategy::_withdraw()` routine, it takes the withdraw fee of `pStake` into the calculation of the desired `stkBNB` amount to withdraw from `pStake` (line 197). However, our analysis shows that the strategy possibly does not own as much `stkBNB` as it is desired to cover the withdraw fee, unless it has accumulated enough rewards in `pStake`. As a result, the transfer of the desired amount of `stkBNB` to `pStake` may fail (line 206), hence reverting the withdrawal.

Recommendation Revise the withdrawal logic from the `MasterVault` and the `StkBnbStrategy` contracts to properly take the fees into the consideration of validating the withdrawn amount, and ensure the desired `stkBNB` amount to be withdrawn from `pStake` does not exceed the `stkBNB` balance of the strategy.

Status This issue has been fixed in the following commit: c4242d3.

3.6 Suggested Strategy Validation in MasterVault Deposit

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterVault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `Helio` protocol, the `MasterVault` contract maintains a list of strategies that can deposit the collected `BNB` to the liquid staking pools. The strategy can be added by the owner and deactivated by the manager. By design, it only allows to deposit `BNB` to the active strategies. While reviewing the deposit of `BNB` to the strategies, we notice it may deposit `BNB` to the invalid or deactivated strategies.

In the following, we show the related code snippets of the `MasterVault::depositAllToStrategy()/depositToStrategy()/_depositToStrategy()` routines. The `MasterVault::depositAllToStrategy()/depositToStrategy()` routines call the `_depositToStrategy()` routine to deposit the given amount of `BNB` to the given strategy. However, we notice that it does not properly validate if the given strategy is valid or not and if it is active or not. As a result, the `BNB` may be deposited to an unsupported strategy or a deactivated strategy.

```
function depositAllToStrategy(address strategy) public onlyManager {
    uint256 amount = totalAssetInVault();
    require(_depositToStrategy(strategy, amount));
}

function depositToStrategy(address strategy, uint256 amount) public onlyManager {
    require(_depositToStrategy(strategy, amount));
}

function _depositToStrategy(address strategy, uint256 amount) private returns (bool success){
    require(amount > 0, "invalid deposit amount");
    require(totalAssetInVault() >= amount, "insufficient balance");
    if (IBaseStrategy(strategy).canDeposit(amount)) {
        uint256 value = IBaseStrategy(strategy).deposit{value: amount}();
        if(value > 0) {
            totalDebt += value;
            strategyParams[strategy].debt += value;
            emit DepositedToStrategy(strategy, amount);
            return true;
        }
    }
}
```

Listing 3.8: `MasterVault.sol`

Recommendation Properly validate the strategy before depositing to it.

Status This issue has been fixed in the following commit: c4242d3.

3.7 Improved Withdrawal Validation in CerosYieldConverterStrategy::_withdraw()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CerosYieldConverterStrategy`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `Helio` protocol, the `CerosYieldConverterStrategy` contract supports the deposit/withdrawal of `BNB` to/from the `CerosRouter` contract. While reviewing the withdrawal of `BNB` from the `CerosRouter` contract, we notice a logic issue in the validation of the returned withdrawal amount, which may wrongly take the return `aBNBc` amount from the `CerosRouter` as the withdrawn `BNB` amount.

To elaborate, we show below the related code snippets of the `CerosYieldConverterStrategy::_withdraw()/CerosRouter::withdraw()` routines. The `CerosYieldConverterStrategy::_withdraw()` routine calls the `CerosRouter::withdraw()` routine (line 87) to withdraw `BNB` from the `CerosRouter`. By comparing the return `value` with the desired amount, it validates if the withdrawal is accepted or not (line 88). However, we notice that the return value of the `CerosRouter::withdraw()` routine is the amount of `aBNBc` (line 191) that is desired to be withdrawn from the `BNB` staking pool, and the `aBNBc` amount is not equal to the `BNB` amount. As a result, the comparison between the two values doesn't make any sense.

```
79     function _withdraw(address recipient, uint256 amount) internal returns (uint256
           value) {
80         require(amount > 0, "invalid amount");
81         uint256 ethBalance = address(this).balance;
82         if(amount < ethBalance) {
83             (bool sent, ) = payable(recipient).call{gas: 5000, value: amount}("");
84             require(sent, "transfer failed");
85             return amount;
86         } else {
87             value = _ceRouter.withdraw(recipient, amount);
88             require(value <= amount, "invalid out amount");
89             return amount;
90         }
91     }
```

Listing 3.9: `CerosYieldConverterStrategy::_withdraw()`

```
178    function withdraw(address recipient, uint256 amount)
179    external
180    override
```

```
    nonReentrant
    returns (uint256 realAmount)
    {
        require(
            amount >= _pool.getMinimumStake(),
            "value must be greater than min unstake amount"
        );
        realAmount = _vault.withdrawFor(msg.sender, address(this), amount);
        _pool.unstakeCertsFor(recipient, realAmount);
        emit Withdrawal(msg.sender, recipient, _wBnbAddress, amount);
        return realAmount;
    }
```

Listing 3.10: `CerosRouter::withdraw()`

Recommendation Properly validate the withdrawn `BNB` amount in the `CerosYieldConverterStrategy::_withdraw()` routine.

Status This issue has been fixed in the following commit: c4242d3.

3.8 Lack of Access Control to CerosYieldConverterStrategy::withdrawInToken()

- ID: PVE-008
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `CerosYieldConverterStrategy`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Helio` protocol, the `MasterVault` contract supports the withdrawal in `LSDs`, i.e., `BNBx/stkBNB/ankrBNB/SnBNB`). While examining the withdrawal logic in `ankrBNB` from the `CerosYieldConverterStrategy` contract, we notice there is a lack of proper access control which may lead to drain the `ankrBNB` funds.

To elaborate, we show below the related `CerosYieldConverterStrategy::withdrawInToken()` routine. As the name indicates, it is used to withdraw in `ankrBNB`. However, it comes to our attention that this routine does not properly validate the caller. As a result, anybody can call this function to withdraw the `ankrBNB` owned to the `MasterVault`.

Our analysis shows that it shall apply the `onlyVault` modifier to the `CerosYieldConverterStrategy::withdrawInToken()` routine.

```
function withdrawInToken(address recipient, uint256 amount)
```

```
external
override
nonReentrant
returns (uint256 realAmount)
{
    return _ceRouter.withdrawABNBc(recipient, amount);
}
```

Listing 3.11: `CerosYieldConverterStrategy::withdrawInToken()`

Recommendation Apply the `onlyVault` modifier to the `CerosYieldConverterStrategy::withdrawInToken()` routine.

Status This issue has been fixed in the following commit: c4242d3.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Helio` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., mint `ceToken`, update debt). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `MasterVault` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the `owner` is privileged to update the total debt and the debt of each strategy which gives the maximum `BNB` amount the `HelioProvider` can withdraw from the `MasterVault`, add new strategy which can accept `BNB` deposit, set the deposit/withdraw fees, change the fee receiver, change the allocation weight of each strategy, etc.

What's more, the `owner` manages a list of managers who can allocate the `BNB` depositing among the strategies, withdraw `BNB` from the strategies and deactivate a strategy, etc.

```
    function _updateCerosStrategyDebt(address strategy, uint256 amount) external
        onlyOwner {
        totalDebt = amount;
        strategyParams[strategy].debt = amount;
    }

    function setStrategy(
```

```
        address strategy,
        uint256 allocation
    ) external onlyOwner {
        require(strategy != address(0));
        require(strategies.length < MAX_STRATEGIES, "max strategies exceeded");
        require(address(IBaseStrategy(strategy).vault()) == address(this), "invalid
            strategy");
        uint256 totalAllocations;
        for(uint256 i = 0; i < strategies.length; i++) {...}

        require(totalAllocations + allocation <= 1e6, "allocations cannot be more than
            100%");

        StrategyParams memory params = StrategyParams({
            active: true,
            allocation: allocation,
            debt: 0
        });

        strategyParams[strategy] = params;
        strategies.push(strategy);
        emit StrategyAdded(strategy, allocation);
    }
    ...

    function withdrawFromStrategy(address strategy, uint256 amount) public onlyManager {
        _withdrawFromStrategy(strategy, address(this), amount);
    }

    function withdrawAllFromStrategy(address strategy) external onlyManager {
        _withdrawFromStrategy(strategy, address(this), strategyParams[strategy].debt);
    }

    function retireStrat(address strategy) external onlyManager {
        // require(strategyParams[strategy].active, "strategy is not active");
        if(_deactivateStrategy(strategy)) {
            return;
        }
        _withdrawFromStrategy(strategy, address(this), strategyParams[strategy].debt);
        _deactivateStrategy(strategy);
    }
```

Listing 3.12: Example Privileged Operations in `MasterVault`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account 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 as the team confirmed the `owner` is a multi-sig account.

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`. The new `Helio` protocol introduces the integration for liquid staking `BNB` tokens (`BNBx`, `stkBNB`, and `SnBNB`), which simplifies users experience in depositing `BNB` as well as provides flexible withdrawal options for users. 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.

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.