SMART CONTRACT AUDIT REPORT

for

OpenSky

Prepared By: Patrick Lou

PeckShield

Jun 09, 2022

Document Properties

Client	OpenSky Finance
Title	Smart Contract Audit Report
Target	OpenSky
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Luck Hu, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jun 09, 2022	Jing Wang	Final Release
1.0-rc	April 22, 2022	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the `OpenSky` protocol design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of `OpenSky` 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 OpenSky

`OpenSky` is a decentralized peer-to-pool pawnshop where `NFT` holders can borrow using their `NFT` assets as collateral and `DeFi` users can earn passive income. The deposits will be passed through directly to `AAVE` and held in their smart contracts. If the borrower fails to repay the loan by a certain time, anyone can start a timed auction and the revenue will be shared to the lenders.

Table 1.1: Basic Information of OpenSky

Item	Description
Name	OpenSky Finance
Website	https://home.opensky.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun 09, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the `OpenSky` protocol assumes a trusted price oracle with timely market price feeds for supported assets. The price manipulation of the oracle is not part of this audit.

- https://github.com/OpenSky-Finance/opensky-protocol.git (114ae17)

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

- https://github.com/OpenSky-Finance/opensky-protocol.git (269163c)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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 OpenSky protocol 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	0	
Medium	3	■■■
Low	3	■■■
Informational	1	■
Total	7	

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

Table 2.1: Key OpenSky Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Time and State	Fixed
PVE-002	Low	Potential Reentrancy Risk in OpenSkyDutchAuction	Time and State	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Removal Of Redundant Code	Coding Practices	Fixed
PVE-005	Medium	Proper Use of Borrow Rate in OpenSkyPool::borrow()	Business Logic	Fixed
PVE-006	Medium	Mismatched IncentiveController Invocation in OpenSkyOToken	Business Logic	Fixed
PVE-007	Low	Proper Interest Accrual on Parameter Changes	Business Logic	Confirmed

Beside the identified issues, we emphasize that the liquidation mechanism of the protocol is time based, not price based. When the borrower fail to repay the loan by a certain time, anyone can start a liquidation by create an auction. The reason behind this design is price based liquidation for `NFT` is vulnerable to `Black Swan` incidents.

Also, 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 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `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.

```
/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
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 3.1: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address to, uint tokens)virtual public returns (bool success)`. 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.

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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `claimERC20Airdrop()` routine in the `OpenSkyLoan` contract. If `USDT` is given as `token`, the unsafe version of `IERC20(token).transfer(to, amount)` (line 345) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
339     function claimERC20Airdrop(
340         address token,
341         address to,
342         uint256 amount
343     ) external override onlyAirdropOperator {
344         // make sure that params are checked in admin contract
345         IERC20(token).transfer(to, amount);
346         emit ClaimERC20Airdrop(token, to, amount);
347     }
```

Listing 3.2: `OpenSkyLoan::claimERC20Airdrop()`

Note that other routines `OpenSkyOToken::claimERC20Rewards()` and `WETHGateway::withdrawETH()` share the same issue.

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

Status This issue has been fixed in the commit: 97cf052.

3.2 Potential Reentrancy Risk in OpenSkyDutchAuction

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OpenSkyDutchAuction`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there is an occasions where the `checks-effects-interactions` principle is violated. Using the `OpenSkyDutchAuction` contract as an example, the `createAuction()` function (see the code snippet below) is provided to takes a `NFT` and start an auction. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 69) starts before effecting the update on internal states (lines 70), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same function.

```
45    function createAuction(
46        uint256 reservePrice,
47        address nftAddress,
48        uint256 tokenId
49    ) external override returns (uint256) {
50        address tokenOwner = IERC721(nftAddress).ownerOf(tokenId);
51        require(msg.sender == tokenOwner, 'AUCTION_CREATE_NOT_TOKEN_OWNER');
52        require(reservePrice > 0, 'AUCTION_CREATE_RESERVE_PRICE_NOT_ALLOWED');
53
54        uint256 auctionId = _auctionIdTracker.current();
55        uint256 startTime = block.timestamp;
56
57        auctions[auctionId] = Auction({
58            startTime: startTime,
59            reservePrice: reservePrice,
60            nftAddress: nftAddress,
61            tokenId: tokenId,
62            tokenOwner: tokenOwner,
```

```
            buyer: address(0),
            buyPrice: 0,
            buyTime: 0,
            status: Status.LIVE
        });

        IERC721(nftAddress).transferFrom(tokenOwner, address(this), tokenId);
        _auctionIdTracker.increment();

        emit AuctionCreated(auctionId, nftAddress, tokenId, tokenOwner, startTime,
            reservePrice);
        return auctionId;
    }
```

Listing 3.3: `OpenSkyDutchAuction::createAuction()`

Recommendation Apply the non-reentrancy protection in all above-mentioned routines.

Status This issue has been fixed by the commit: 34a415f.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `OpenSky` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol parameters and feed oracle price). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contract.

To elaborate, we show below the related function. The `updatePrice()` routine supports the configuration of the collateral `NFT` price, which is a key parameter in the calculation of the borrow limit in the `getBorrowLimitByOracle()` routine.

```
    function updatePrice(
        address nftAddress,
        uint256 price,
        uint256 timestamp
    ) public override onlyOwner {
```

```
        NFTPriceData[] storage prices = nftPriceFeedMap[nftAddress];
        NFTPriceData memory latestPriceData = prices.length > 0
            ? prices[prices.length - 1]
            : NFTPriceData({roundId: 0, price: 0, timestamp: 0, cumulativePrice: 0});
        require(timestamp > latestPriceData.timestamp, Errors.
            PRICE_ORACLE_INCORRECT_TIMESTAMP);
        uint256 cumulativePrice = latestPriceData.cumulativePrice +
            (timestamp - latestPriceData.timestamp) *
            latestPriceData.price;
        uint256 roundId = latestPriceData.roundId + 1;
        NFTPriceData memory data = NFTPriceData({
            price: price,
            timestamp: timestamp,
            roundId: roundId,
            cumulativePrice: cumulativePrice
        });
        prices.push(data);

        emit UpdatePrice(nftAddress, price, timestamp, roundId);
    }
```

Listing 3.4: `OpenSkyCollateralPriceOracle::updatePrice()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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

Status The issue has been confirmed by the team. The team clarifies the plan to use a multi-sig contract as the `owner`. Also, the team will search a better solution for the `NFT` price oracle. In the long run, the team confirms the plan to transfer the privileged account to the intended `DAO`-like governance contract.

3.4 Removal Of Redundant Code

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The OpenSky protocol makes good use of a number of reference contracts, such as ERC721Holder, ERC721Enumerable, and Ownable, to facilitate its code implementation and organization. For example, the OpenSkyLoan smart contract has so far imported at least three reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the OpenSkyLoan::updateStatus() implementation, the associated modifier onlyPool ensures the contract will only be called by the pool contract. However, there is no implementation to call the updateStatus() routine from the OpenSkyPool contract. This updateStatus() routine can be safely removed if there is no use case of it.

```
    function updateStatus(uint256 tokenId, DataTypes.LoanStatus status) external
        override onlyPool {
        _updateStatus(tokenId, status);
    }
```

Listing 3.5: OpenSkyLoan::updateStatus()

```
contract OpenSkyCollateralPriceOracle is Ownable, IOpenSkyCollateralPriceOracle {
    IOpenSkySettings public immutable SETTINGS;

    mapping(address => NFTPriceData[]) public nftPriceFeedMap;
    mapping(address => mapping(uint256 => uint256)) private _prices;

    uint256 internal _roundInterval = 100;
    ...
}
```

Listing 3.6: OpenSkyCollateralPriceOracle::_prices

In the same vein, the OpenSkyCollateralPriceOracle contract defines a mapping state _prices, which is never used in current protocol. This unused state can be safely removed as well.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: 41ce5ad.

3.5 Proper Use of Borrow Rate in OpenSkyPool::borrow()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: OpenSkyPool
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

As mentioned earlier, OpenSky is a decentralized peer-to-pool pawnshop where NFT holders can borrow using their NFT assets as collateral and DeFi users can earn passive income. While examining the current borrow-related logic, we notice the current use of the borrow rate needs to be revisited.

To elaborate, we show below the related borrow() function. As the name indicates, this function allows an NFT holder to borrow with the holding NFT assets as collateral. It comes to our attention that the borrow rate is computed by making use of the utilization rate before the borrow operation occurs. In fact, the proper borrow rate needs to be computed with the utilization rate after the borrow operation occurs! In other words, the current implementation may incur less cost for the borrowing user at the cost of collecting less fee for existing liquidity providers!

```
    function borrow(
        uint256 reserveId,
        uint256 amount,
        uint256 duration,
        address nftAddress,
        uint256 tokenId,
        address onBehalfOf
    ) public virtual override whenNotPaused nonReentrant checkReserveExists(reserveId)
        returns (uint256) {
        require(
            duration >= SETTINGS.minBorrowDuration() && duration <= SETTINGS.
                maxBorrowDuration(),
            Errors.BORROW_DURATION_NOT_ALLOWED
        );

        BorrowLocalParams memory vars;
        vars.borrowLimit = getBorrowLimitByOracle(reserveId, nftAddress, tokenId);
        vars.availableLiquidity = getAvailableLiquidity(reserveId);

        vars.amountToBorrow = amount;

        if (amount == type(uint256).max) {
            vars.amountToBorrow = (
                vars.borrowLimit < vars.availableLiquidity ? vars.borrowLimit : vars.
                    availableLiquidity
            );
        }
```

```

        require(vars.borrowLimit >= vars.amountToBorrow, Errors.
            BORROW_AMOUNT_EXCEED_BORROW_LIMIT);
        require(vars.availableLiquidity >= vars.amountToBorrow, Errors.
            RESERVE_LIQUIDITY_INSUFFICIENT);

        IERC721(nftAddress).safeTransferFrom(_msgSender(), SETTINGS.loanAddress(),
            tokenId);

        vars.borrowRate = reserves[reserveId].getBorrowRate();
        (uint256 loanId, DataTypes.LoanData memory loan) = IOpenSkyLoan(SETTINGS.
            loanAddress()).mint(
            reserveId,
            onBehalfOf,
            nftAddress,
            tokenId,
            vars.amountToBorrow,
            duration,
            vars.borrowRate
        );
        reserves[reserveId].borrow(loan);

        emit Borrow(
            reserveId,
            _msgSender(),
            onBehalfOf,
            nftAddress,
            tokenId,
            vars.amountToBorrow,
            duration,
            vars.borrowRate,
            loan.borrowOverdueTime,
            loanId
        );

        return loanId;
    }
```

Listing 3.7: `OpenSkyPool::borrow()`

Recommendation Properly revise the `borrow()` logic to compute the right borrow rate.

Status The issue has been fixed by this commit: `6fc5020`.

3.6 Mismatched IncentiveController Invocation in OpenSkyOToken

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OpenSkyOToken`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

The `OpenSky` protocol has the built-in, extensible incentive mechanism with the introduction of `_incentivesController` that is designed to keep the accounting logic of rewards for protocol users. In the following, we examine the specific interactions with `_incentivesController` and notice the inconsistent uses during the interactions.

```
5  interface IOpenSkyIncentivesController {
6      function handleAction(
7          address account,
8          uint256 userBalance,
9          uint256 totalSupply
10     ) external;
```

Listing 3.8: IOpenSkyIncentivesController :: handleAction()

To elaborate, we show above the `handleAction()` callback function that is invoked when an user interacts with the protocol. We notice that this function takes three arguments: the first one indicates the related `account` (line 7), the second one (line 8) shows the current balance at the specific moment when the user interacts with the protocol; and the last one (line 9) is the current total supply of related tokens.

In the following, we show one specific case when the above `handleAction()` function is invoked. Specifically, it happens when the supported `OpenSkyOToken` tokens are being transferred. The internal `_transfer()` routine properly takes a record of current balance of the user as well as the total supply, and then calls the above `handleAction()` function.

```
    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal override {
        uint256 index = IOpenSkyPool(_pool).getReserveNormalizedIncome(_reserveId);

        uint256 amountScaled = amount.rayDivTruncate(index);
        require(amountScaled != 0, Errors.AMOUNT_SCALED_IS_ZERO);
        require(amountScaled <= type(uint128).max, Errors.AMOUNT_TRANSFER_OWERFLOW);
```

```

        uint256 previousSenderBalance = super.balanceOf(sender);
        uint256 previousRecipientBalance = super.balanceOf(recipient);

        super._transfer(sender, recipient, amountScaled);

        if (address(_incentivesController) != address(0)) {
            uint256 currentTotalSupply = super.totalSupply();
            _incentivesController.handleAction(sender, currentTotalSupply,
                previousSenderBalance);
            if (sender != recipient) {
                _incentivesController.handleAction(recipient, currentTotalSupply,
                    previousRecipientBalance);
            }
        }
    }
```

Listing 3.9: `OpenSkyOToken::_transfer()`

It comes to our attention that the caller invokes `handleAction()` with an inconsistent list of arguments. Particularly, the second argument is `currentTotalSupply`, instead of the expected user balance. Also, the third argument is the user balance (`oldSenderBalance/oldRecipientBalance`), instead of the expected total supply. A correct function definition of `handleAction()` needs to switch the order of its current last two arguments.

Recommendation Properly revise the `_handleAction()` invocation with a correct argument order.

Status The issue has been fixed by this commit: `bb8e061`.

3.7 Proper Interest Accrual on Parameter Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OpenSkyPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `OpenSky` protocol is no exception. Specifically, if we examine the `OpenSkyPool` contract, it has defined a number of protocol-wide risk parameters, such as `treasuryFactor` and `interestModelAddress`. In the following, we show the corresponding routines that allow for their changes.

```
    /// @inheritdoc IOpenSkyPool
    function setTreasuryFactor(uint256 reserveId, uint256 factor)
        external
        override
        checkReserveExists(reserveId)
        onlyPoolAdmin
    {
        reserves[reserveId].treasuryFactor = factor;
        emit SetTreasuryFactor(reserveId, factor);
    }

    /// @inheritdoc IOpenSkyPool
    function setInterestModelAddress(uint256 reserveId, address interestModelAddress)
        external
        override
        checkReserveExists(reserveId)
        onlyPoolAdmin
    {
        reserves[reserveId].interestModelAddress = interestModelAddress;
        emit SetInterestModelAddress(reserveId, interestModelAddress);
    }
```

Listing 3.10: `OpenSkyPool::setTreasuryFactor()/setInterestModelAddress()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by properly applying their changes. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the current implementation of `setTreasuryFactor()` may apply the new treasury factor in the interest calculation for the past time period. In other words, there is a need to apply the old treasury factor for the past un-collected interest and the new treasury factor should be used starting the very moment it is updated.

Recommendation Properly apply the new treasury factor for the future interest-related calculation, not the past one.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the OpenSky protocol design and implementation. The OpenSky protocol is a decentralized NFT lending protocol, which allows users to borrow against their NFTs on one side of the marketplace or earn interest on their deposits on the other side. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

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

[13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[14] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.