SMART CONTRACT AUDIT REPORT

for

OpenSky Bespoke Loan

Prepared By: Xiaomi Huang

PeckShield

January 5, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	January 5, 2023	Jing Wang	Final Release
1.0-rc	December 25, 2022	Jing Wang	Release Candidate

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 OpenSky Bespoke Loan 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 version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About OpenSky Bespoke Loan

OpenSky Bespoke Loan is one of OpenSky's products, which is a customizable loan agreement tailored to the specific needs and circumstances of various borrowers. This type of loan contract allows for greater flexibility and control over the terms of the loan, including the repayment schedule, interest rate, and collateral requirements. The basic information of OpenSky Bespoke Loan is as follows:

Table 1.1: Basic Information of OpenSky Bespoke Loan

Item	Description
Issuer	OpenSky Finance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 5, 2023

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

- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/bespokemarket/ (82ab617)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/misc/ape-staking/ (82ab617)

- https://github.com/OpenSky-Finance/opensky-tokenomic/tree/main/contracts/token/ (d615541)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/moneymarkets/ApeCoinStakingMoneyMarket.sol (82ab617)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/misc/OpenSkyLoanDelegator.sol (392585c)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/bespokemarket (0f12a88)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/misc/ape-staking (78a6565)
- https://github.com/OpenSky-Finance/opensky-tokenomic/tree/main/contracts/token (d615541)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/moneymarkets/ApeCoinStakingMoneyMarket.sol (82ab617)
- https://github.com/OpenSky-Finance/opensky-protocol/tree/dev/contracts/misc/OpenSkyLoanDelegator.sol (3ee3a18)

1.2 About PeckShield

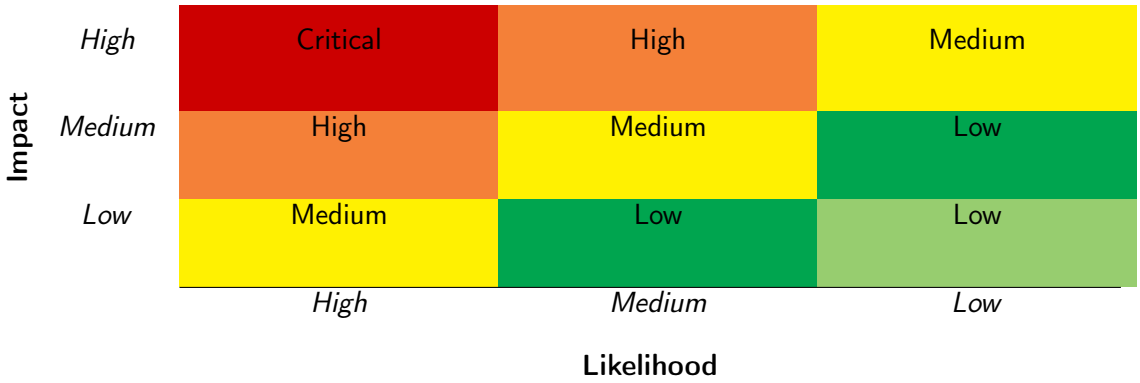
PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

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

Table 1.2: Vulnerability Severity Classification



Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- 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

Table 1.3: The Full Audit Checklist

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

better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

PeckShield

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the OpenSky protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	1	■
Undetermined	1	■
Total	4	

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

2.2 Key Findings

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

Table 2.1: Key Audit Findings of OpenSky Bespoke Loan

ID	Severity	Title	Category	Status
PVE-001	High	Revisited Logic Of OpenSkyApeCoinStakingHelper::depositBAYC()	Business Logic	Fixed
PVE-002	Undetermined	Proper Handling Of Ape Coin Staked for Liquidable BAYC	Business Logic	Confirmed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Potential Reentrancy Risk in flashLoanSimple()	Business Logic	Fixed

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.

PeckShield

3 Detailed Results

3.1 Revisited Logic Of OpenSkyApeCoinStakingHelper::depositBAYC()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `OpenSkyApeCoinStakingHelper`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

To support the `BAYC/MAYC` loans that are in the borrowing status to participate in `Ape Coin` staking through the `flashClaim()` method, the `OpenSkyLoan` and `TransferAdapterERC721Default` contracts have implemented the `flashClaim ABI` for `Instant Loans` and `Bespoke Loans`, respectively. Users can call the corresponding contract's `ABI` based on the type of loan.

In the original implementation of the `OpenSkyApeCoinStakingHelper` contract, the team notices that the `depositBAYC()` routine does not validate the recipient address when transferring `Ape Coins`. To elaborate, we show the related routine from the `OpenSkyApeCoinStakingHelper` contract.

```
52    function depositBAYC(IApeCoinStaking.SingleNft[] calldata _nfts, address _recipient)
          public onlySelf {
53        uint256 amount;
54        for (uint256 i; i < _nfts.length; ++i) {
55            amount += _nfts[i].amount;
56        }
57
58        apeCoin.safeTransferFrom(_recipient, address(this), amount);
59        apeCoin.safeApprove(address(apeCoinStaking), amount);
60
61        apeCoinStaking.depositBAYC(_nfts);
62    }
```

Listing 3.1: `OpenSkyApeCoinStakingHelper::depositBAYC()`

For example, Alice approves 200 Ape Coins to the OpenSkyApeCoinStakingHelper contract and then uses flashClaim() to call depositBAYC() and stakes 100 Ape Coins. At this point, if a bad actor knows that Alice has approved 200 Ape Coins and that there are still 100 Ape Coins that can be transferred by the OpenSkyApeCoinStakingHelper, they could also use flashClaim() to call depositBAYC() and stake 100 Ape Coins using Alice's 100 Ape Coins. To solve this issue, the team splits the OpenSkyApeCoinStakingHelper contract into separate contracts, with each operation being implemented in its own contract. This would allow each operation to perform parameter validation. However, when reviewing this change, we notice the problem is still there. To elaborate, we show the related routine from the OpenSkyApeCoinStakingHelper contract.

```
22      function executeOperation(
23          address[] calldata nftAddresses,
24          uint256[] calldata tokenIds,
25          address initiator,
26          address operator,
27          bytes calldata params
28      ) external override returns (bool) {
29          require(msg.sender == operator, "PARAMS_ERROR");
30
31          (uint256[] memory baycs, address recipient) = abi.decode(params, (uint256[],
                address));
32
33          apeCoinStaking.claimBAYC(baycs, recipient);
34
35          for (uint256 i; i < nftAddresses.length; i++) {
36              IERC721(nftAddresses[i]).approve(operator, tokenIds[i]);
37          }
38
39          return true;
40      }
```

Listing 3.2: OpenSkyClaimBAYCHelper::executeOperation()

It comes to our attention that within the executeOperation() routine, bad actors can write a contract to transfer his BAYC to the helper contract without using flashClaim(), and then directly call executeOperation() to use Alice's Ape Coins to do staking. Afterwards, bad actors can transfer their BAYC back to themselves as the operator is the same with msg.sender.

Recommendation It is necessary to add an additional msg.sender verification to limit the ability to call executeOperation() to only the OpenSkyLoan and TransferERC721Default contracts.

Status The issue has been fixed by this commit: 1210016.

3.2 Proper Handling Of Ape Coin Staked for Liquidable BAYC

- ID: PVE-002
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, users can use their `BAYC`/`MAYC` loans to do `Ape Coin` staking through the `flashClaim()` method. However, when reviewing this part of logic, we notice if users' loans are liquidable, they will be unable to withdraw their staked `Ape Coin` from the `Ape Coin Staking` contract. This means that the user will not only lose the borrowed `BAYC`, but also the `Ape Coin` they staked to the `Ape Coin Staking` contract. To elaborate, we show below the related routines.

```
78      function flashClaim(
79          address receiverAddress,
80          uint256[] calldata loanIds,
81          bytes calldata params
82      ) external override {
83          uint256 i;
84          IOpenSkyFlashClaimReceiver receiver = IOpenSkyFlashClaimReceiver(receiverAddress
                );
85          // !!!CAUTION: receiver contract may reentry mint, burn, flashClaim again
86
87          // only loan owner can do flashClaim
88          address[] memory nftAddresses = new address[](loanIds.length);
89          uint256[] memory tokenIds = new uint256[](loanIds.length);
90          for (i = 0; i < loanIds.length; i++) {
91              require(
92                  IERC721(BESPOKE_SETTINGS.borrowLoanAddress()).ownerOf(loanIds[i]) ==
                        _msgSender(),
93                  'BM_FLASHCLAIM_CALLER_IS_NOT_OWNER'
94              );
95              BespokeTypes.LoanData memory loanData = IOpenSkyBespokeMarket(
                    BESPOKE_SETTINGS.marketAddress()).getLoanData(
96                  loanIds[i]
97              );
98              require(loanData.status != BespokeTypes.LoanStatus.LIQUIDATABLE, '
                    BM_FLASHCLAIM_STATUS_ERROR');
99              nftAddresses[i] = loanData.tokenAddress;
100             tokenIds[i] = loanData.tokenId;
101         }
102         ...
103     }
```

Listing 3.3: `TransferAdapterCollateralBase::flashClaim()`

We understand the limitation to forbid a liquidable loan to use `flashClaim()`. However, as the deposit of `Ape Coin` by `flashClaim()` is made after the borrower's order has been taken, the platform might need to warn the user to withdraw their staked `Ape Coin` before the loan becomes liquidable. Otherwise they may suffer larger losses than expected.

Recommendation Add necessary warnings to the user on the possibilities losing both liquidable loans and staked `Ape Coin`.

Status This issue has been confirmed. The team clarifies they will add warnings to the user from frontend.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `OpenSky` protocol, there is a special administrative account, i.e., `admin`. This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., role setting). 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 `admin` account and its related privileged accesses in current contract.

To elaborate, we show the related routine from the `OpenSkyBespokeSettings` contract. This routine allows the `GOVERNANCE_ROLE` account defined by `ACLManager` contract to configure parameters like `_currencyTransferAdapters` and `_transferAdapters` which holds user funds.

```
15      modifier onlyGovernance() {
16          IACLManager ACLManager = IACLManager(ACLManagerAddress);
17          require(ACLManager.isGovernance(_msgSender()), 'BM_ACL_ONLY_GOVERNANCE_CAN_CALL'
                );
18          _;
19      }

21      function addCurrencyTransferAdapter(address currency, address adapterAddress)
            external onlyGovernance {
22          require(currency != address(0) && adapterAddress != address(0));
23          _currencyTransferAdapters[currency] = adapterAddress;
24          emit AddCurrencyTransferAdapter(msg.sender, currency, adapterAddress);
25      }
```

```
27      function addNftTransferAdapter(address nftAddress, address adapterAddress) external
            onlyGovernance {
28          require(nftAddress != address(0) && adapterAddress != address(0));
29          _transferAdapters[nftAddress] = adapterAddress;
30          emit AddNftTransferAdapter(msg.sender, nftAddress, adapterAddress);
31      }
```

Listing 3.4: Admin Controlled Routines

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `admin` 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 This issue has been confirmed. The team clarifies they plan on using a multi-sig contract to start, and eventually migrating ownership of sensitive contracts to DAO-like governance contract.

3.4 Potential Reentrancy Risk in flashLoanSimple()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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 [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `TransferAdapterCollateralBase` as an example, the `flashClaim()` function (see the code snippet

below) is provided to support the NFT loans that are in the borrowing status to participate in claiming/staking. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

In particular, the interaction with the external contract inside flashClaim() (line 110) starts before effecting the update on the internal state. More importantly, it carries over the stale cache states and apply them for the calculation of protocol-wide interest rates, while ignoring the possibility that it may re-enter to update the protocol state. These updates may be lost since they are overwritten by the stale states!

```
78  function flashClaim(
79      address receiverAddress,
80      uint256[] calldata loanIds,
81      bytes calldata params
82  ) external override {
83      uint256 i;
84      IOpenSkyFlashClaimReceiver receiver = IOpenSkyFlashClaimReceiver(receiverAddress);
85      // !!!CAUTION: receiver contract may reentry mint, burn, flashClaim again
86
87      // only loan owner can do flashClaim
88      address[] memory nftAddresses = new address[](loanIds.length);
89      uint256[] memory tokenIds = new uint256[](loanIds.length);
90      for (i = 0; i < loanIds.length; i++) {
91          require(
92              IERC721(BESPOKE_SETTINGS.borrowLoanAddress()).ownerOf(loanIds[i]) ==
                    _msgSender(),
93              'BM_FLASHCLAIM_CALLER_IS_NOT_OWNER'
94          );
95          BespokeTypes.LoanData memory loanData = IOpenSkyBespokeMarket(BESPOKE_SETTINGS.
                marketAddress()).getLoanData(
96              loanIds[i]
97          );
98          require(loanData.status != BespokeTypes.LoanStatus.LIQUIDATABLE, '
                BM_FLASHCLAIM_STATUS_ERROR');
99          nftAddresses[i] = loanData.tokenAddress;
100         tokenIds[i] = loanData.tokenId;
101     }
102
103     // step 1: moving underlying asset forward to receiver contract
104     for (i = 0; i < loanIds.length; i++) {
105         IERC721(nftAddresses[i]).safeTransferFrom(address(this), receiverAddress,
                tokenIds[i]);
106     }
107
108     // setup 2: execute receiver contract, doing something like airdrop
109     require(
110         receiver.executeOperation(nftAddresses, tokenIds, _msgSender(), address(this),
                params),
111         'BM_FLASHCLAIM_EXECUTOR_ERROR'
112     );
113
```

```
114     // setup 3: moving underlying asset backward from receiver contract
115     for (i = 0; i < loanIds.length; i++) {
116         IERC721(nftAddresses[i]).safeTransferFrom(receiverAddress, address(this),
                tokenIds[i]);
117         emit FlashClaim(receiverAddress, _msgSender(), nftAddresses[i], tokenIds[i]);
118     }
119 }
```

Listing 3.5: `TransferAdapterCollateralBase::flashClaim()`

Note other routines from the `OpenSkyLoanDelegator` contract share the same issue.

Recommendation Revise the above `flashClaim()` routine by applying necessary reentrancy prevention to avoid the use of cached state to overwrite legitimate protocol updates.

Status The issue has been fixed by the following PR: `3ee3a18`.

4 Conclusion

In this audit, we have analyzed the OpenSky Bespoke Loan design and implementation. OpenSky Bespoke Loan is one of OpenSky's products, which is a customizable loan agreement that is tailored to the specific needs and circumstances of the borrower. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

PeckShield

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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

PeckShield