SMART CONTRACT AUDIT REPORT

for

PawnFi ApeStaking

Prepared By: Xiaomi Huang

PeckShield

June 5, 2023

Document Properties

Client	PawnFi
Title	Smart Contract Audit Report
Target	ApeStaking
Version	1.2
Author	Xuxian Jiang
Auditors	Stephen Bie, Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.2	June 5, 2023	Xuxian Jiang	Post Release #2
1.1	May 29, 2023	Xuxian Jiang	Post Release #1
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc	May 5, 2023	Xuxian Jiang	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 design document and related smart contract source code of the PawnFi's ApeStaking 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 given version of PawnFi 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 ApeStaking

The ApeStaking protocol is part of the PawnFi ecosystem and is designed to streamline the staking process for PawnFi users, enabling them to effortlessly stake their Ape coins and benefit from compounded rewards through automatic reinvestment. Catering to both experienced NFT enthusiasts engaged with PawnFi's consignment, leverage, and lending modules, as well as newcomers seeking to maximize their Ape staking returns, the contract offers a seamless experience for all users. By interacting directly with the Horizen Labs Contract, ApeStaking ensures the most legitimate Ape coin rewards, providing users with a secure and efficient solution to optimize their staking investments. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ApeStaking

Item	Description
Name	ApeStaking
Website	https://pawnfi.com
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 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/PawnFi/ApeStaking.git (94b2bff)
- https://github.com/PawnFi/NFTFactory.git (966b049)

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

- https://github.com/PawnFi/ApeStaking.git (b39c6c6)
- https://github.com/PawnFi/NFTFactory.git (a5cbf8a)

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 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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `ApeStaking` 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	1
Low	2
Informational	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 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 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key ApeStaking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Borrow/Supply Rate Calculation	Business Logic	Resolved
PVE-002	Low	Empty Market Avoidance with MINIMUM_LIQUIDITY Enforcement	Numeric Errors	Resolved
PVE-003	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Resolved
PVE-004	Medium	Improved Owner Verification of Staking NFTs	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited Borrow/Supply Rate Calculation

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ApePool
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The ApeStaking protocol has a built-in lending component of ApePool, which accepts Ape coins as the underlying asset. While examining the on-chain per-block borrow rate and supply rate, we notice the current approach may be revisited.

To elaborate, we show below the implementation of two related routines: borrowRatePerBlock() and supplyRatePerBlock(). The first routine returns the current per-block borrow interest rate while the second routine returns the current per-block supply interest rate. It comes to our attention that each has a common part, i.e., getRewardRatePerBlock(). In the calculation of per-block supply interest rate, there is a need to take into consideration the current utilization as well as the reserve factor, which is missing in the common part.

```
/**
 * @notice Returns the current per-block borrow interest rate for this cToken
 * @return The borrow interest rate per block, scaled by 1e18
 */
function borrowRatePerBlock() external view returns (uint) {
    return interestRateModel.getBorrowRate(getCashPrior(), totalBorrows, 0) +
        getRewardRatePerBlock();
}

/**
 * @notice Returns the current per-block supply interest rate for this cToken
 * @return The supply interest rate per block, scaled by 1e18
 */
```

```
    function supplyRatePerBlock() external view returns (uint) {
        return interestRateModel.getSupplyRate(getCashPrior(), totalBorrows, 0,
            reserveFactorMantissa) + getRewardRatePerBlock();
    }
```

Listing 3.1: `ApePool::borrowRatePerBlock()/supplyRatePerBlock()`

Recommendation Revisit the logic to compute the per-block supply interest rate.

Status The issue has been fixed by this commit: `693fbec`.

3.2 Empty Market Avoidance With MINIMUM_LIQUIDITY Enforcement

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ApePool`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [2]

Description

As mentioned earlier, the `ApePool` contract is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. While reviewing the redeem logic, we notice the current implementation has a precision issue that has been reflected in a recent `HundredFinance` hack.

To elaborate, we show below the related `redeemFresh()` routine. As the name indicates, this routine is designed to redeems the pool tokens in exchange for the underlying asset. When the user indicates the underlying asset amount (via `redeemUnderlying()`), the respective `redeemTokens` is computed as `redeemTokens = div_(redeemAmountIn, exchangeRate)` (line 470). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the `redeemTokens` amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the recent `HundredFinance` hack: https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33.

```
447     function redeemFresh(address redeemer, uint redeemTokensIn, uint redeemAmountIn)
            internal {
448         require(redeemTokensIn == 0  redeemAmountIn == 0, "one of redeemTokensIn or
                redeemAmountIn must be zero");
```

```
        /* exchangeRate = invoke Exchange Rate Stored() */
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal() });

        uint redeemTokens;
        uint redeemAmount;
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be
                 redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            revert RedeemFreshnessCheck();
        }
        ...
    }
```

Listing 3.2: `ApePool::redeemFresh()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, we need to ensure that markets are never empty by minting small pool token balances at the time of market creation so that we can prevent the rounding error being used maliciously. A deposit as small as 1 wei is sufficient.

Status The issue has been resolved since it is the only supported market and the borrow is not directly possible.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ApeStaking`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `ApeStaking::unstakeAndRepay()` as an example. This routine is used to suspend staking for users with high health factor.

```
597     function unstakeAndRepay(address userAddr, address[] calldata nftAssets, uint256[]
            calldata nftIds) external nonReentrant {
598         require(nftAssets.length == nftIds.length, "size err");
599         uint256 totalIncome;
600         uint256 totalPay;
601         (totalIncome, totalPay) = getUserHealth(userAddr);
602         require(totalIncome < totalPay * stakingConfiguration.liquidateRate /
                BASE_PERCENTS, "income less");
603         for(uint256 i = 0; i < nftAssets.length; i++) {
604             require(userAddr == _nftInfo[nftAssets[i]].staker[nftIds[i]], "owner err");
605             _onStopStake(nftAssets[i], nftIds[i], RewardAction.STOPSTAKE);
606             (totalIncome, totalPay) = getUserHealth(userAddr);
607             if(totalIncome >= totalPay * stakingConfiguration.borrowSafeRate /
                    BASE_PERCENTS) {
608                 _transferAsset(pawnToken, msg.sender, stakingConfiguration.
                        liquidatePawnAmount);
609                 break;
610             }
611         }
612     }
```

Listing 3.3: `ApeStaking::unstakeAndRepay()`

We notice the comparison between `totalIncome` and `totalPay` (line 602) involves mixed multiplication and devision. For improved precision, it is better to revise as follows: `require(totalIncome * BASE_PERCENTS < totalPay * stakingConfiguration.liquidateRate)` (line 602). Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note the `if`-statement (line 607) shares the same issue.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `fa21184`.

3.4 Improved Owner Verification of Staking NFTs

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ApeStaking`
- Category: Business Logic [5]
- CWE subcategory: CWE-708 [3]

Description

The `ApeStaking` contract streamlines the staking process for `PawnFi` users, enabling them to effortlessly stake their `Ape` coins and benefit from compounded rewards through automatic reinvestment. In the process of reviewing the current staking logic, we notice the owner verification of staking `NFTs` should be improved.

In particular, we show below the related routine `_validOwner()`. As the name indicates, this routine is designed to verify the `NFT` owner. It has a rather straightforward logic in querying the possible holder in the `_nftInfo` array. If it is not recorded (line 363), it further queries the `ptokenStaking` contract for the current holding contract. The returned `nftOwner` is queried again for the actual owner. Note the current holding contract can be whitelisted to ensure only the approved holding contracts may be queried. Otherwise, the current owner verification may be bypassed.

```
353     /**
354      * @notice Verify NFT owner
355      * @param userAddr User address
356      * @param ptokenStaking Address of NFT staking agency
357      * @param nftAsset nft asset address
358      * @param nftId nft id
359      * @return bool true: Verification pass false: Verification fail
360      */
361     function _validOwner(address userAddr, address ptokenStaking, address nftAsset,
            uint256 nftId) internal view returns (bool) {
362         address holder = _nftInfo[nftAsset].depositor[nftId];
363         if(holder == address(0)) {
364             address nftOwner = IPTokenApeStaking(ptokenStaking).getNftOwner(nftId);
365             holder = INftGateway(nftOwner).nftOwner(userAddr, nftAsset, nftId);
366         }
```

```
        return holder == userAddr;
    }
```

Listing 3.4: `ApeStaking::_validOwner()`

Recommendation Improve the above owner verification logic to ensure the final holder is the intended one.

Status The issue has been fixed by the following commit: `e1c1fb7`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `ApeStaking` protocol, which is part of the `PawnFi` ecosystem and is designed to streamline the staking process for `PawnFi` users, enabling them to effortlessly stake their `Ape` coins and benefit from compounded rewards through automatic reinvestment. Catering to both experienced `NFT` enthusiasts engaged with `PawnFi`'s consignment, leverage, and lending modules, as well as newcomers seeking to maximize their Ape staking returns, the contract offers a seamless experience for all users. By interacting directly with the `Horizen Labs Contract`, `ApeStaking` ensures the most legitimate `Ape` coin rewards, providing users with a secure and efficient solution to optimize their staking investments. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

[6] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.