SMART CONTRACT AUDIT REPORT

for

TimeNFTs

Prepared By: Yiqun Chen

PeckShield

January 23, 2022

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	TimeNFTs
Version	1.0-rc
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	August 2, 2021	Luck Hu	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the source code of the **TimeNFTs** smart contracts, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, 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 TimeNFTs

`Non-Fungible Time` or `Time NFTs` are designed to represent an on-chain attestation of work for contributors, which can be an effective approach for organizations to recruit talent for specialized needs. In particular, the `NFTs` represent the time for performing gig work and other use cases and can be dynamically minted and purchased. While work is the initial use case for `Time NFTs`, this primitive is envisioned to be extended to other use cases, such as restaurant reservations, subleases, etc. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of `TimeNFTs`

Item	Description
Issuer	Aave
Type	ERC721 Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	January 23, 2022

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/WeAreNewt/NonFungibleTime.git (bc97f43)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 the OWASP Risk Rating Methodology [7]:

- 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
	Approve / TransferFrom Race Condition
ERC721 Compliance Checks	Compliance Checks (Section 3)
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
	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.
- ERC721 Compliance Checks: We also validate whether the implementation logic of the audited smart contract(s) follows the standard ERC721 specification and other best practices.
- 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) [6], 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 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 `TimeNFTs` contract design and 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 logics, examine system operations, and place ERC721-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	
Informational	1	
Total	3	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC721 specification and other known best practices, and validate its compatibility with other similar ERC721 tokens and current DeFi protocols. The detailed ERC721 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC721 compliance issue was found and our detailed checklist can be found in Section 3. Note that the smart contract implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of TimeNFTs

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	
PVE-002	Low	Royalty Fee Bypass With Direct ERC721 safeTransferFrom()	Business Logic	
PVE-003	Informational	Redundant Code Removal	Coding Practices	

In the meantime, we also need to emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks.

3 | ERC721 Compliance Checks

The ERC721 standard for non-fungible tokens, also known as deeds. Inspired by the ERC-20 token standard, the ERC721 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC721-compliant. Naturally, we examine the list of necessary API functions defined by the ERC721 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC721 Specification

Item	Description	Status
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
ownerOf()	Is declared as a public view function	✓
	Returns the address of the owner of the NFT	✓
getApproved()	Is declared as a public view function	✓
	Reverts while '_tokenId' does not exist	✓
	Returns the approved address for this NFT	✓
isApprovedForAll()	Is declared as a public view function	✓
	Returns a boolean value which check '_operator' is an approved operator	✓

Our analysis shows that there is no ERC721 inconsistency or incompatibility issue found in the audited TimeNFTs. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC721 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC721 Specification

Item	Description	Status
safeTransferFrom()	Is declared as a public function	✓
	Reverts while ‘to‘ refers to a smart contract and not implement IERC721Receiver-onERC721Received	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
transferFrom()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
approve()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Emits Approval() event when tokens are approved successfully	✓
setApprovalForAll()	Is declared as a public function	✓
	Reverts while not approving to caller	✓
	Emits ApprovalForAll() event when tokens are approved successfully	✓
Transfer() event	Is emitted when tokens are transferred	✓
Approval() event	Is emitted on any successful call to approve()	✓
ApprovalForAll() event	Is emitted on any successful call to setApprovalForAll()	✓

4 | Detailed Results

4.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: NonFungibleTimeCollection
- Category: Coding Practices [4]
- 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 `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
```

```
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 4.1: USDT::transferFrom()

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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.

In the following, we show the `_transferCurrency()` routine in the `NonFungibleTimeCollection` contract. To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (line 355).

```
345     function _transferCurrency(
346         address sender,
347         address payable receiver,
348         address currency,
349         uint256 amount
350     ) internal {
351         bool transferSucceed;
352         if (currency == address(0)) {
353             (transferSucceed, ) = receiver.call{value: amount}('');
354         } else {
355             transferSucceed = IERC20(currency).transferFrom(sender, receiver, amount);
356         }
357         if (!transferSucceed) revert TransferFailed();
358     }
```

Listing 4.2: NonFungibleTimeCollection::_transferCurrency()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status

4.2 Royalty Fee Bypass With Direct ERC721 safeTransferFrom()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: NonFungibleTimeCollection
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Each tradable asset supported in TimeNFTs is represented as an ERC721-based NFT token, which naturally has the standard implementation, e.g., transferFrom()/safeTransferFrom(). By design, each tradable asset is embedded with a price. Any interested user can buy it by fulfilling its set price. When a price is fulfilled, the NFT token is transferred to the buyer. Some percentage (represented by royaltyBasisPoints / BASIS_POINTS) of the funds from that buyer is transferred to the royaltyReceiver and the rest is transferred to the current token owner.

To elaborate, we show below the buyToken() routine. This routine is provided to support trading on TimeNFTs. It comes to our attention that instead of transferring a royaltyAmount amount of royalty to royaltyReceiver for each trade, it is possible for the current owner and the buyer to directly negotiate a price, without paying the royaltyReceiver. The NFT can then be arranged and delivered by the current owner to directly call transferFrom()/safeTransferFrom() with the buyer as the recipient.

```
    function buyToken(uint256 tokenId) external payable onlyExistingTokenId(tokenId) {
        if (msg.sender == address(0)) revert InvalidAddress(msg.sender);
        address payable owner = payable(ownerOf(tokenId));
        if (owner == msg.sender) revert CantBuyYourOwnToken(msg.sender, tokenId);
        Token memory token = tokens[tokenId];
        if (!isCurrencyAllowed[token.currency]) revert UnallowedCurrency(tokenId, token.
            currency);
        if (!token.forSale) revert NotForSale(tokenId);
        if (token.allowedBuyer != address(0) && msg.sender != token.allowedBuyer)
            revert NotAuthorizedBuyer(msg.sender, tokenId);
        token.forSale = false;
        tokens[tokenId] = token;
        _transfer(owner, msg.sender, tokenId);
        if (owner != token.royaltyReceiver) {
            uint256 royaltyAmount = (token.price * token.royaltyBasisPoints) /
                BASIS_POINTS;
            _transferCurrency(msg.sender, token.royaltyReceiver, token.currency,
                royaltyAmount);
            _transferCurrency(msg.sender, owner, token.currency, token.price -
                royaltyAmount);
        } else {
            _transferCurrency(msg.sender, owner, token.currency, token.price);
```

```
        }
        emit TokenBought(tokenId, owner, msg.sender);
    }
```

Listing 4.3: `NonFungibleTimeCollection::buyToken()`

Recommendation Implement a locking mechanism so that any `TimeNFTs` token needs to be locked in the `NonFungibleTimeCollection` contract in order to be only tradable in `TimeNFTs`.

Status

4.3 Redundant Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `NonFungibleTimeCollection`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

TimeNFTs makes good use of a number of reference contracts, such as `ERC721Upgradeable`, `Strings`, and `OwnableUpgradeable`, to facilitate its code implementation and organization. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

Specifically, if we examine closely the `NonFungibleTimeCollection::buyToken()` routine, we notice the validation of `msg.sender == address(0)` (line 139) will always be false, which means the code in the `if` statement will never be executed.

```
138    function buyToken(uint256 tokenId) external payable onlyExistingTokenId(tokenId) {
139        if (msg.sender == address(0)) revert InvalidAddress(msg.sender);
140        address payable owner = payable(ownerOf(tokenId));
141        if (owner == msg.sender) revert CantBuyYourOwnToken(msg.sender, tokenId);
142        Token memory token = tokens[tokenId];
143        if (!isCurrencyAllowed[token.currency]) revert UnallowedCurrency(tokenId, token.
               currency);
144        if (!token.forSale) revert NotForSale(tokenId);
145        if (token.allowedBuyer != address(0) && msg.sender != token.allowedBuyer)
146            revert NotAuthorizedBuyer(msg.sender, tokenId);
147        ...
```

Listing 4.4: `NonFungibleTimeCollection::buyToken()`

What is more, TimeNFTs defines many custom error instances which could be well used to describe errors. But it comes to our attention that the `NotEnoughFunds` (line 35) error instance is defined but never used.

```
    error TokenDoesntExist(uint256 tokenId);
    error OnlyTokenOwner(uint256 tokenId);
    error OnlyCurrentRoyaltyReceiver(uint256 tokenId);
    error InvalidAddress(address addr);
    error NotForSale(uint256 tokenId);
    error NotAuthorizedBuyer(address buyer, uint256 tokenId);
    error CantBuyYourOwnToken(address buyer, uint256 tokenId);
    error NotEnoughFunds(uint256 tokenId);
    error AlreadyRedeemed(uint256 tokenId);
    error UnallowedCurrency(uint256 tokenId, address currency);
    error TransferFailed();
    error InvalidRoyalty();
    error InvalidTimeParams();
```

Listing 4.5: `NonFungibleTimeCollection.sol`

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

Status

5 Conclusion

In this security audit, we have examined the `NonFungibleTimeCollection` contract design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC721 specification and other known ERC721 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, we found two low-severity issues and one informational recommendation which are promptly addressed by the team. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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