SMART CONTRACT AUDIT REPORT

for

ST404

Prepared By: Xiaomi Huang

PeckShield

March 22, 2024

Document Properties

Client	STL
Title	Smart Contract Audit Report
Target	ST404
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 22, 2024	Xuxian Jiang	Final Release
1.0-rc	March 21, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the ST404 token contract, 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 the smart contract can be further improved due to the presence of certain issues related to ERC20/ERC721-compliance, security, or performance. This document outlines our audit results.

1.1 About ST404

ST404 introduces an innovative token standard that builds upon the Pandora-404 works, aiming to optimize gas efficiency and incorporate dynamic, game-like elements into token transactions. It merges the liquidity and transferability of ERC20 tokens with the unique identification and collectibility of ERC721 tokens. It is designed to address the challenges of high gas costs in large token transfers and enhance user engagement through unique token characteristics. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of ST404

Item	Description
Name	STL
Type	ERC20/ERC721 Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 22, 2024

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

- https://github.com/SmartTokenLabs/ST404.git (d9e9e9f)

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

- https://github.com/SmartTokenLabs/ST404.git (8bb074d)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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 OWASP Risk Rating Methodology [5]:

- 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.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

We perform the audit according to the following procedures:

- 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.

- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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
ERC20 Compliance Checks	Compliance Checks (Section 3)
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

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 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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the ST404 token contract. 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 ERC20/ERC721-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20/ERC721 specification and other known best practices, and validate its compatibility with other similar ERC20/ERC721 tokens and current DeFi protocols. The detailed ERC20/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, there is no ERC20/ERC721 compliance issue and our detailed checklist can be found in Section 3. While there is no critical or high severity issue, the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key ST404 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved safeTransferFrom() Logic in ERC404Legacy	Coding Practices	Resolved
PVE-002	Low	Lack of Native NFT Adjustment Upon Account Whitelisting	Business Logic	Resolved
PVE-003	Medium	Revisited _transferERC721() Logic in ST404	Business Logic	Resolved
PVE-004	Low	Improved tokenOfOwnerByIndex()/_burnAllMalleables() Logic in ST404	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate the above issue(s), we also 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 and Section 4 for elaboration of reported issues.

3 | ERC20/ERC721 Compliance

The ERC20/ERC721 specifications define 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 ERC20/ERC721-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

3.1 ERC20 Compliance

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue in the audited `ST404` token contract. 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 ERC20 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements, but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` `ERC20` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Whitelistable	The token contract allows the owner or privileged users to whitelist a specific address such that only token transfers and other operations related to that address are allowed	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	—
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

3.2 ERC721 Compliance

The `ERC721` standard for non-fungible tokens, also known as deeds. Inspired by the `ERC20` 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 3.4: 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 the `balanceOf()` function is defined to be `ERC20-compliant`. Thus, this specific function does not count all NFTs assigned to an owner. And there is no other ERC721 inconsistency or incompatibility issue found in the audited Pandora token contract. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.4) and key state-changing functions (Table 3.5) according to the widely-adopted ERC721 specification.

Table 3.5: 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 Improved safeTransferFrom() Logic in ERC404Legacy

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC404Legacy
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

ST404 combines the functionalities of ERC20 and ERC721 and makes it necessary to revisit respective transfer support. In particular, when discerning whether a given transfer is intended for ERC20 or ERC721, we need to compare the transfer amount with minted – current mint counter. If the transfer amount is larger than minted, we can be certain that an ERC20 transfer is intended. Otherwise, it will be determined to be an ERC721 transfer. Our analysis shows the comparision with minted should be improved.

For example, in the safeTransferFrom() function, it is designed to transfer the given ERC721 token. However, it simply delegates the call to transferFrom() (line 238) without validating require(id>0 && id<=minted).

```
236     /// @notice Function for native transfers with contract support
237     function safeTransferFrom(address from, address to, uint256 id) public virtual {
238         transferFrom(from, to, id);
239 
240         if (
241             to.code.length != 0 &&
242             ERC721Receiver(to).onERC721Received(msg.sender, from, id, "") !=
                    ERC721Receiver.onERC721Received.selector
243         ) {
244             revert UnsafeRecipient();
245         }
246     }
```

Listing 4.1: ERC404Legacy::safeTransferFrom()

Similarly, the `transfer()` function in the same contract can be improved by enforcing `require(amount>minted)`. And another `transferFrom()` function can be improved by further comparing with `amountOrId > 0` in the existing comparision with `amountOrId<=minted`.

Recommendation Apply necessary validation in the above-mentioned transfer functions.

Status This issue has been fixed by the following commit: `42015f6`.

4.2 Lack of Native NFT Adjustment Upon Account Whitelisting

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC404Legacy`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `ST404` token has a `ERC404Legacy` parent contract, which supports the feature of whitelisting user accounts. The feature basically exempt certain addresses from `ERC721` transfer, typically for gas savings (pairs, routers, etc). However, our analysis shows that it also needs to adjust the `ERC721` balances of the affected user account to respect whitelisting rules.

To elaborate, we show below the implementation of the `setWhitelist()` function. It has a rather straightforward logic in whitelisting the given user account. However, if an account is whitelisted, we need to burn holding NFTs. If an account is not whitelisted anymore, we need to mint respective NFTs. The implications from whitelisting an account is not considered in current implementation in `ERC404Legacy`.

```
    /// @notice Initialization function to set pairs / etc
    ///         saving gas by avoiding mint / burn on unnecessary targets
    function setWhitelist(address target, bool state) public onlyOwner {
        whitelist[target] = state;
        emit SetERC721TransferExempt(target, state);
    }
```

Listing 4.2: `ERC404Legacy::setWhitelist()`

Recommendation Improve the above-mentioned routine to take into account of possible implications from whitelisting a user account.

Status This issue has been fixed by the following commit: `42015f6`.

4.3 Revisited _transferERC721() Logic in ST404

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ST404
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

ST404 combines the functionalities of ERC20 and ERC721, which effectively merges the liquidity and transferability of ERC20 tokens with the unique identification and collectibility of ERC721 tokens. While reviewing the ERC721-specific transfer logic, we notice current implementation should be improved.

To elaborate, we show below the implementation of the related `_transferERC721()` routine. As the name indicates, this routine is used to transfer an ERC721 NFT token. However, when the NFT is a malleable one, there is a need to further discern whether the given recipient is `address(0)` or not. If it is `address(0)`, we need to consider it is a burn operation and further call with `_burnERC721(tokenId)`.

```
    function _transferERC721(address from, address to, uint tokenId) internal returns (
        bool) {
        _checkAuthorized(from, msg.sender, tokenId);
        address ownedOwner = _ownerOf[tokenId];
        address nativeOwner;
        if (ownedOwner == address(0)) {
            nativeOwner = _getMalleableOwner(tokenId);

            if (nativeOwner == address(0)) {
                revert  InvalidToken();
            }
            if (from != nativeOwner) {
                revert InvalidSender();
            }
            _transferMalleable(nativeOwner, to, tokenId);
            _doTransferERC20(from, to, unit);
        } else {
        ...
    }
```

Listing 4.3: ST404::_transferERC721()

Recommendation Revisit the above routine for an improved ERC721 transfer logic.

Status This issue has been fixed by the following commits: a5301f5 and 8bb074d.

4.4 Improved tokenOfOwnerByIndex()/_burnAllMalleables() Logic in ST404

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: ST404
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the combined functionalities of ERC20 and ERC721 in ST404 make it necessary to revisit respective transfer support. In this section, we examine a few helper routines and report possible optimizations.

The first helper routine is `tokenOfOwnerByIndex()`, which is used to enumerate a given user's NFT token. We notice an internal `for`-loop which has maximum loop count `total + _solidified[owner].length()` (line 525), which can be optimized to be `total - owned + _solidified[owner].length()`.

```
514     function tokenOfOwnerByIndex(address owner, uint256 index) public view returns (
            uint256) {
515         uint owned = _owned[owner].length;
516         if (owned > index) {
517             return _owned[owner][index];
518         }
519         uint total = _balanceOf[owner] / unit;
520         if (total <= index) {
521             revert IndexOutOfBounds();
522         }
523         uint tokenId;
524         uint skipIndex = index - owned;
525         for (uint i = 0; i < total + _solidified[owner].length(); i++) {
526             tokenId = _encodeOwnerAndId(owner, i);
527             if (!_solidified[owner].contains(tokenId)) {
528                 if (skipIndex == 0) {
529                     return tokenId;
530                 }
531                 unchecked {
532                     skipIndex--;
533                 }
534             }
535         }
536         revert IndexOutOfBounds();
537     }
```

Listing 4.4: `ST404::tokenOfOwnerByIndex()`

The second function is `_burnAllMalleables()`, which supports to burn all malleables of the given account. Notice the `currentSubIdToBurn` should be computed as `tokensToBurn + _solidified[target_].length()`, not current `balanceOf(target_)/ unit + _solidified[target_].length()` (line 600).

```
598     function _burnAllMalleables(address target_) private {
599         uint tokensToBurn = balanceOf(target_) / unit - _owned[target_].length;
600         uint currentSubIdToBurn = balanceOf(target_) / unit + _solidified[target_].
                length();
601         while (tokensToBurn > 0) {
602             currentSubIdToBurn = _burnMaximalMalleable(currentSubIdToBurn, target_);
603             unchecked {
604                 tokensToBurn--;
605             }
606         }
607     }
```

Listing 4.5: `ST404::_burnAllMalleables()`

Recommendation Revisit the above-mentioned routines for improved token transfer logic.

Status This issue has been fixed by the following commit: `a5301f5`.

5 Conclusion

In this security audit, we have examined the ST404 token design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20/ERC721 specification and other known ERC20/ERC721 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, there are no critical level vulnerabilities discovered and other identified issues are promptly addressed.

References

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

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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