SMART CONTRACT AUDIT REPORT

for

Pandora (ERC404)

Prepared By: Xiaomi Huang

PeckShield

February 10, 2024

Document Properties

Client	Pandora
Title	Smart Contract Audit Report
Target	Pandora
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	February 10, 2024	Xuxian Jiang	Final Release
1.0-rc	February 9, 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 `Pandora` 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 Pandora

`Pandora` is an `ERC404` token, which is experimental and built on the `Ethereum` blockchain. It aims to combine the functionalities of `ERC20` tokens (fungible tokens) and `ERC721` tokens (non-fungible tokens, or `NFTs`) into a single standard. The audited version covers its compliance of `ERC20/ERC721` specifications as well as other known best practices and their security implications. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Pandora

Item	Description
Name	Pandora
Website	https://pandora.build
Type	ERC20/ERC721 Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 10, 2024

In the following, we show the deployment address of the audited token contract:

- PANDORA: https://etherscan.io/address/0x9E9FbDE7C7a83c43913BddC8779158F1368F0413

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

- https://github.com/Pandora-Labs-Org/erc404.git (b36d92c)

1.2 About PeckShield

PeckShield Inc. [8] 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 [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.

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 `Pandora` 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	2	
Low	2	
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 exists an 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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Pandora Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Lack of Return Value in transferFrom()	Coding Practices	Resolved
PVE-002	Medium	Improved Token Transfer Logic in Pandora	Business Logic	Resolved
PVE-003	Low	Possibly Ambiguous Approval Events	Coding Practices	Resolved
PVE-004	Low	Trust Issue Of Admin Keys	Security Features	Mitigated

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 an ERC20 inconsistency or incompatibility issue found in the audited `Pandora` token contract. Specifically, as detailed in Section 4.1, the `transferFrom()` function needs to be defined to have a return value in boolean. The lack of return value may pose issues for external integration, including token lockup and/or swap failure. 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	N/A
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 Lack of Return Value in transferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Pandora`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Pandora` token contract implements the experimental `ERC404` token standard on the `Ethereum` blockchain. It aims to combine the functionalities of `ERC20` tokens (fungible tokens) and `ERC721` tokens (non-fungible tokens, or `NFTs`). While examining its `ERC20` compliance, we notice there is an issue in current `transferFrom()` function.

To elaborate, we show below the implementation of the `transferFrom()` function. The official `ERC20` specification indicates that this `transferFrom()` function needs to be defined to have a return value in boolean, i.e., `function transferFrom(address _from, address _to, uint256 _value)public returns (bool success)`. The lack of return value may lead to unexpected consequence for external integration, including token lockup and/or swap failure.

```
    function transferFrom(
        address from,
        address to,
        uint256 amountOrId
    ) public virtual {
        if (amountOrId <= minted) {
            if (from != _ownerOf[amountOrId]) {
                revert InvalidSender();
            }

            if (to == address(0)) {
                revert InvalidRecipient();
            }
```

```
            if (
                msg.sender != from &&
                !isApprovedForAll[from][msg.sender] &&
                msg.sender != getApproved[amountOrId]
            ) {
                revert Unauthorized();
            }

            balanceOf[from] -= _getUnit();

            unchecked {
                balanceOf[to] += _getUnit();
            }

            _ownerOf[amountOrId] = to;
            delete getApproved[amountOrId];

            // update _owned for sender
            uint256 updatedId = _owned[from][_owned[from].length - 1];
            _owned[from][_ownedIndex[amountOrId]] = updatedId;
            // pop
            _owned[from].pop();
            // update index for the moved id
            _ownedIndex[updatedId] = _ownedIndex[amountOrId];
            // push token to to owned
            _owned[to].push(amountOrId);
            // update index for to owned
            _ownedIndex[amountOrId] = _owned[to].length - 1;

            emit Transfer(from, to, amountOrId);
            emit ERC20Transfer(from, to, _getUnit());
        } else {
            uint256 allowed = allowance[from][msg.sender];

            if (allowed != type(uint256).max)
                allowance[from][msg.sender] = allowed - amountOrId;

            _transfer(from, to, amountOrId);
        }
    }
```

Listing 4.1: `Pandora::transferFrom()`

Recommendation Redefine the above routine to have a return value of the boolean type.

Status This issue has been resolved in the following PR: 5.

4.2 Improved Token Transfer Logic in Pandora

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Pandora`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier 4.1, we have examined the definition of `transferFrom()` in `Pandora`. In this section, we further examine the actual implementation logic and report possible improvements that may be made to better support `ERC20` and `ERC721` functionalities.

```
    function safeTransferFrom(
        address from,
        address to,
        uint256 id
    ) public virtual {
        transferFrom(from, to, id);

        if (
            to.code.length != 0 &&
            ERC721Receiver(to).onERC721Received(msg.sender, from, id, "") !=
            ERC721Receiver.onERC721Received.selector
        ) {
            revert UnsafeRecipient();
        }
    }
```

Listing 4.2: `Pandora::safeTransferFrom()`

To elaborate, we show above the related `safeTransferFrom()` routine. This routine allows to transfer tokens from the specified sender to the intended recipient. However, it comes to our attention that it always calls `onERC721Received()` on the recipient if it is a contract – even when it may be standard `ERC20` (and non-native) token transfers. In other words, the above routine may be improved by further restricting the `onERC721Received()` check with the following requirements, i.e., `amountOrId <= minted && amountOrId > 0`.

Similarly, the regular `transferFrom()` routine may also be improved by validating `amountOrId <= minted && amountOrId > 0` when it is interpreted as native `ERC721` transfers.

Recommendation Improve the above-mentioned routines to better validate the conditions for native `ERC721` transfers.

Status This issue has been resolved in the following PR: `5`.

4.3 Possibly Ambiguous Approval Events

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pandora
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we examine the Pandora contract and notice the emitted Approval events may be overloaded or even misleading. Specifically, we show below the related approve() routine, which emits Approval for both native and regular token approvals. However, it is possible for non-native token approval event be interpreted as a native token approval if an earlier non-native small-amount will be later read as a native approval (especially when the non-native amount becomes actually smaller than minted).

```
    function approve(
        address spender,
        uint256 amountOrId
    ) public virtual returns (bool) {
        if (amountOrId <= minted && amountOrId > 0) {
            address owner = _ownerOf[amountOrId];

            if (msg.sender != owner && !isApprovedForAll[owner][msg.sender]) {
                revert Unauthorized();
            }

            getApproved[amountOrId] = spender;

            emit Approval(owner, spender, amountOrId);
        } else {
            allowance[msg.sender][spender] = amountOrId;

            emit Approval(msg.sender, spender, amountOrId);
        }

        return true;
    }
```

Listing 4.3: Pandora::approve()

Recommendation Accurately emit the `Approval` event unambiguously.

Status This issue has been resolved in the following PR: `5`.

4.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pandora`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Pandora` token contract, there is a privileged admin account `owner` that plays a critical role in regulating the token-wide operations (e.g., whitelist users and set token/data UI). In the following, we show the representative function potentially affected by this privilege.

```
    function setDataURI(string memory _dataURI) public onlyOwner {
        dataURI = _dataURI;
    }

    function setTokenURI(string memory _tokenURI) public onlyOwner {
        baseTokenURI = _tokenURI;
    }

    /// @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;
    }
```

Listing 4.4: Example Privileged Operations in `Pandora`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

Status The issue has been mitigated with the use of multi-sig to manage the admin key.

5 Conclusion

In this security audit, we have examined the Pandora 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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