SMART CONTRACT AUDIT REPORT

for

Pikaster Protocol

Prepared By: Patrick Lou

PeckShield

April 12, 2022

Document Properties

Client	Pikaster
Title	Smart Contract Audit Report
Target	Pikaster Protocol
Version	1.1
Author	Patrick Lou
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Luck Hu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.1	April 12, 2022	Patrick Lou	Final Release (Amended #1)
1.0	April 1, 2022	Patrick Lou	Final Release

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction **4**
- 1.1 About Pikaster Protocol . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings **8**
- 2.1 Summary . . . 8
- 2.2 Key Findings . . . 9

3 ERC20 Compliance Checks **10**

4 ERC721 Compliance Checks **13**

5 Detailed Results **15**
- 5.1 Incorrect Logic Of ERC721Template::remint() . . . 15
- 5.2 Safe-Version Replacement With safeTransfer() . . . 16
- 5.3 Trust Issue Of Admin Keys . . . 17

6 Conclusion **19**

References **20**

1 Introduction

Given the opportunity to review the design document and related source code of the `Pikaster` 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 either security or performance. This document outlines our audit results.

1.1 About Pikaster Protocol

`Pikaster` is a card battle game featuring `Pikaster` (NFT) with the goal to create a "truly-play and truly-earn" game through innovative product features to bring players both extraordinary gaming experience and good economic returns. The audited contracts include the `ERC20Template` and the `ERC721Template` contracts which define the `ERC20` and the `ERC721` tokens used in `Pikaster` protocol respectively. The basic information of the audited token contract is as follows:

Table 1.1: Basic Information of Pikaster Protocol

Item	Description
Issuer	Pikaster
Type	ERC20/ERC721 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	April 12, 2022

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

- https://github.com/pikasterdev/pikaster (bdacec6)

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

- https://github.com/pikasterdev/pikaster (ae6673b)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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

Likelihood

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 `Pikaster` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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 tokens and current DeFi protocols. The detailed ERC20/ERC721 compliance checks are reported in Sections 3 and 4. 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 5.

2.2 Key Findings

Overall, no ERC20 or ERC721 compliance issue was found, and our detailed checklist can be found in Sections 3 and 4. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Pikaster Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect Logic Of ERC721Template::remint()	Coding Practices	Fixed
PVE-002	Low	Safe-Version Replacement With safeTransfer()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 5 for details.

3 | ERC20 Compliance Checks

The ERC20 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 ERC20-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).

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 found in the audited Pikaster Protocol. 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-

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()	✓

adopted ERC20 specification. 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 (e.g., ERC777/ERC2222), 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` 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	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	✓
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	✓

4 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 4.1: 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 Pikaster Protocol. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 4.1) and key `state-changing` functions (Table 4.2) according to the widely-adopted ERC721 specification.

Table 4.2: 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()	✓

5 | Detailed Results

5.1 Incorrect Logic Of ERC721Template::remint()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC721Template`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

The `Pikaster` protocol defines the `ERC721Template` which is the `ERC721` token used in the `Pikaster` gaming ecosystem. While examining the `remint()` routine, we notice the current implementation can be improved.

To elaborate, we show below the `remint()` function. As the name indicates, this function is used to reassign an existing `ERC721` token to another owner. It comes to our attention that the current implementation just calls `"_mint(to, tokenId)"` to perform a `remint` action. As the existing `tokenId` has previously been assigned, the `"_mint(to, tokenId)"` (line 160) will always be reverted if the `tokenId` was not burned before by the `MINTER_ROLE`. To avoid it, we suggest adding the `"_burn(tokenId)"` before the `mint` operation to make sure the existing token will be burned first during the `remint()` process.

```
158     function remint(address to, uint256 tokenId) public onlyRole(MINTER_ROLE) {
159         require(tokenId < tokenIds, "Remint: tokenId must less than tokenIds");
160         _mint(to, tokenId);
161     }
```

Listing 5.1: `ERC721Template::remint()`

Recommendation Make sure `_burn()` is invoked before the `_mint()` operation as suggested above.

Status The issue has been fixed by this commit: `a6088fe`.

5.2 Safe-Version Replacement With safeTransfer()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC20Template, ERC721Template
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [2]

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 `transfer()` 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.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 5.2: `USDT Token Contract`

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address to, uint256 value)external returns (bool)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `withdrawERC20()` routine in the `ERC20Template` contract. If `USDT` is given as `token`, the unsafe version of `token.safeTransfer(to, balance)` (line 123) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
120     function withdrawERC20(address tokenAddress, address to) public onlyRole(
            DEFAULT_ADMIN_ROLE) {
121         IERC20 token = IERC20(tokenAddress);
122         uint256 balance = token.balanceOf(address(this));
123         token.transfer(to, balance);
124     }
```

Listing 5.3: `ERC20Template::withdrawERC20()`

Note that the routine `ERC721Template::withdrawERC20()` shares the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()`.

Status The issue has been fixed by this commit: `a6088fe`.

5.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ERC20Template`, `ERC721Template`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In the `Pikaster` protocol, there are privileged `DEFAULT_ADMIN_ROLE/MINTER/BURNER` accounts that play a critical role in governing and regulating the system-wide operations (e.g., mint/burn token, pause the contract, blacklist account, withdraw tokens, etc). Our analysis shows that the privileged accounts need to be scrutinized. In the following, we examine the privileged accounts and the related privileged accesses in current contracts.

```
function mint(address to, uint256 amount) public virtual onlyRole(MINTER_ROLE) {
    _mint(to, amount);
}

...
function pause() public virtual onlyRole(DEFAULT_ADMIN_ROLE) {
```

```
        _pause();
    }

    function unpause() public virtual onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }

    function blackListAccount(address account) public onlyRole(OPERATOR_ROLE) {
        blackListAccounts[account] = true;
        emit BlackListAccount(account);
    }

    function unblackListAccount(address account) public onlyRole(OPERATOR_ROLE) {
        blackListAccounts[account] = false;
        emit UnblackListAccount(account);
    }

    function blackListed(address account) public view returns (bool) {
        return blackListAccounts[account];
    }

    ...
    function withdrawERC20(address tokenAddress, address to) public onlyRole(
        DEFAULT_ADMIN_ROLE) {
        IERC20 token = IERC20(tokenAddress);
        uint256 balance = token.balanceOf(address(this));
        token.safeTransfer(to, balance);
    }

    function withdrawERC721(address tokenAddress, address to, uint256 tokenId) public
        onlyRole(DEFAULT_ADMIN_ROLE) {
        IERC721(tokenAddress).transferFrom(address(this), to, tokenId);
    }
```

Listing 5.4: ERC20Template::mint()/burn()

We understand the need of the privileged function for contract operation, but at the same time the extra power to the `DEFAULT_ADMIN_ROLE/MINTER/BURNER` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

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

Status This issue has been mitigated. The team decides to use a multi-sig contract for the privileged `DEFAULT_ADMIN_ROLE/MINTER/BURNER` accounts.

6 Conclusion

In this security audit, we have examined the design and implementation of the Pikaster contract. 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, although no critical or high level vulnerabilities were discovered, we identified three issues of varying severities. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

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