SMART CONTRACT AUDIT REPORT

for

Creator.Bid Protocol

Prepared By: Xiaomi Huang

PeckShield

August 16, 2024

Document Properties

Client	Creator.Bid
Title	Smart Contract Audit Report
Target	Creator.Bid
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 16, 2024	Xuxian Jiang	Final Release
1.0-rc1	August 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 smart contract source code of the `Creator.Bid` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of 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 Creator.Bid

`Creator.Bid` is the hub for the `AI creator` economy, allowing everyone to create, market, and co-own `AI Agents`. These `AI Agents` offer content automation while their human owners earn for guiding their creative direction via `Agent Keys`. `Creator.Bid` is uniquely positioned to play a key role in the transition towards an autonomous economy, with its ecosystem token, `$BID`, providing access to each `Agent Keys` launch. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Creator.Bid

Item	Description
Target	Creator.Bid
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2024

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

- https://github.com/creatorbid/smart-contracts.git (bf5fd22)

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

- https://github.com/creatorbid/smart-contracts.git (3fe8106, b1c4073)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Creator.Bid` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Creator.Bid Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Claim-Handling Logic in BID-Distributor	Business Logic	Resolved
PVE-002	Informational	Redundant Code/State Removal in AK-Endorsement	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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

3 | Detailed Results

3.1 Revisited Claim-Handling Logic in BIDDistributor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BIDDistributor
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The Creator.Bid protocol has a utility token (BID) and its distribution is managed via the BIDDistributor contract. While examining the related token distribution logic, we notice current implementation can be improved to claim distributed tokens.

In the following, we show the implementation of the related _handleClaim() routine. As the name indicates, this routine is used to handle user claims. It comes to our attention that each claim cid has an amount state to keep track of the available amount for distribution and this amount is not updated after each claim is successfully handled. With that, we suggest to add the following statement, i.e., distribution.amount -= _claim.amount.

```
    function _handleClaim(Claim calldata _claim) private {
        if (_claim.proof.length == 0) revert InvalidProof();
        if (!agentKeyCard.isAgentKey(_claim.ak)) revert Errors.InvalidAddress();
        if (_claim.amount == 0) revert Errors.InvalidAmount();
        Distribution storage distribution = _distributions[_claim.cid];
        if (distribution.amount == 0) revert Errors.Unavailable();
        if (hasClaimed[_msgSender()][_claim.cid][_claim.ak]) revert AlreadyClaimed();
        bytes32 leaf = keccak256(abi.encodePacked(_msgSender(), _claim.ak, _claim.amount
            ));
        bool isValidProof = MerkleProof.verifyCalldata(_claim.proof, distribution.root,
            leaf);
        if (!isValidProof) revert InvalidProof();
        hasClaimed[_msgSender()][_claim.cid][_claim.ak] = true;
    }
```

Listing 3.1: BIDDistributor::_handleClaim()

Recommendation Improve the above-mentioned routine to timely update the distribution amount so that the contract always maintains the latest available amount for distribution.

Status This issue has been resolved in the following commit: `3fe8106`.

3.2 Redundant Code/State Removal in AKEndorsement

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AKEndorsement`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Creator.Bid` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `MerkleProof`, and `Ownable`, to facilitate its code implementation and organization. For example, the `BIDDistributor` smart contract has so far imported at least four reference contracts. 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 `AKEndorsement` contract, it inherits from a parent `Ownable` contract. However, this `AKEndorsement` contract does not have any privileged functions that require the `owner`-based caller verification. With that, the `Ownable` inheritance can be removed.

```
contract AKEndorsement is Ownable {
    using SafeERC20 for IERC20;

    struct Cooldown {
        address ak;
        uint96 amount;
        uint64 claimableAt;
    }

    uint64 private constant TWO_WEEKS = uint64(14 days);
    IERC20 public immutable bid;
    address public immutable bidDistributor;
    AgentKeyCard public immutable agentKeyCard;
    ...
}
```

Listing 3.2: The `AKEndorsement` Contract

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been resolved in the following commit: `3fe8106`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Creator.Bid` protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role assignment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setEndorsementContract(AKEndorsement _endorsement) external onlyOwner {
        if (address(endorsement) != address(0)) revert Errors.Unavailable();
        if (address(_endorsement) == address(0)) revert Errors.InvalidAddress();
        endorsement = _endorsement;
        emit EndorsementContractUpdated(address(_endorsement));
    }

    /**
     * @notice Set the distributor address
     * @param _account Distributor address
     * @param _isDistributor True - Set as a distributor, False - Set it to no longer be a distributor
     */
    function setDistributor(address _account, bool _isDistributor) external onlyOwner {
        if (_account == address(0)) revert Errors.InvalidAddress();
        if (isDistributor[_account] == _isDistributor) revert Errors.Unavailable();
        isDistributor[_account] = _isDistributor;
        emit UpdatedDistributors(_account, _isDistributor);
    }
```

Listing 3.3: Example Privileged Functions in `BIDDistributor`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts 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 protocol users.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated as the team confirms the use of a multi-sig to manage the admin privilege.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Creator.Bid` protocol, which is the hub for the `AI creator` economy, allowing everyone to create, market, and co-own `AI Agents`. These `AI Agents` offer content automation while their human owners earn for guiding their creative direction via `Agent Keys`. `Creator.Bid` is uniquely positioned to play a key role in the transition towards an autonomous economy, with its ecosystem token, `$BID`, providing access to each `Agent Keys` launch. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.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] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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