SMART CONTRACT AUDIT REPORT

for

Winnables

Prepared By: Xiaomi Huang

PeckShield

August 5, 2024

Document Properties

Client	Winnables
Title	Smart Contract Audit Report
Target	Winnables
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 5, 2024	Xuxian Jiang	Final Release
1.0-rc	July 27, 2024	Xuxian Jiang	Release Candidate

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 `Winnables` 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 Winnables

The `Winnables` protocol allows the creation of a new `raffle` with parameters such as duration, minimum entries, maximum entries, and prizes. The prizes are locked before the `raffle` creation, and participants can enter the `raffle` by purchasing entries. Once the `raffle` is concluded, the winners are selected with the help of external randomness provided by `Chainlink VRF`, after which the winners can claim their prizes. The contract also provides functionalities for the participants to withdraw their funds in case the created `raffle` is cancelled. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Winnables

Item	Description
Name	Winnables
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 5, 2024

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

- https://github.com/Winnables/public-contracts.git (ae08bb4)

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

- https://github.com/Winnables/public-contracts.git (70971bf)

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.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would 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 `Winnables` protocol, implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	2	■ ■
Informational	0	
Total	4	

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 that 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 1 high-severity vulnerability, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Winnables Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Possible Repeated Prize Claims in WinnablesPrizeManager::claimPrize()	Time and State	Resolved
PVE-002	Low	Improved Raffle Cancellation Logic	Coding Practices	Resolved
PVE-003	Low	Improved Validation of Function Arguments	Business Logic	Resolved
PVE-004	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 3 for details.

3 Detailed Results

3.1 Possible Repeated Prize Claims in WinnablesPrizeManager::claimPrize()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `WinnablesPrizeManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Winnables` protocol has a core `WinnablesTicketManager` to manage the tickets for each `Raffle`. This ticket manager provides a number of privileged functions. Our analysis on one specific function to claim the winning `Raffle` prize shows it does not check the claim status and allows for repeated claims.

To elaborate, we show below the implementation of the related `claimPrize()` routine. As the name indicates, it is provided to claim the `Raffle` prize for the winning user. However, the invocation of related functions to transfer fund requires extra care in validating the claim status and avoiding repeated claims. Currently, a winning user can invoke this routine multiple times to repeatedly claim the prize.

```
    function claimPrize(uint256 raffleId) external {
        if (msg.sender != _winners[raffleId]) {
            revert UnauthorizedToClaim();
        }
        RaffleType raffleType = _raffleType[raffleId];
        if (raffleType == RaffleType.NONE) {
            revert InvalidRaffle();
        }
        if (raffleType == RaffleType.NFT) {
            NFTInfo storage raffle = _nftRaffles[raffleId];
            _nftLocked[raffle.contractAddress][raffle.tokenId] = false;
            _sendNFTPrize(raffle.contractAddress, raffle.tokenId, msg.sender);
```

```
        }
        if (raffleType == RaffleType.TOKEN) {
            TokenInfo storage raffle = _tokenRaffles[raffleId];
            unchecked { _tokensLocked[raffle.tokenAddress] -= raffle.amount; }
            _sendTokenPrize(raffle.tokenAddress, raffle.amount, msg.sender);
        }
        if (raffleType == RaffleType.ETH) {
            unchecked { _ethLocked -= _ethRaffles[raffleId]; }
            _sendETHPrize(_ethRaffles[raffleId], msg.sender);
        }
        emit PrizeClaimed(raffleId, msg.sender);
    }
```

Listing 3.1: `WinnablesPrizeManager::claimPrize()`

Recommendation Improve the above `claimPrize()` routine to ensure the prize can be claimed only once by the winner. To fix, we suggest the adherence of known `checks-effects-interactions` practice, which can also be followed to strengthen the `WinnablesTicketManager::buyTickets()` routine.

Status The issue has been fixed by this commit: `dd62e8a`.

3.2 Improved Raffle Cancellation Logic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WinnablesPrizeManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.1, the `Winnables` protocol has a built-in `WinnablesPrizeManager` contract that is designed to manage prizes for `Raffles`. While examining the related `Raffle`-cancellation logic, we notice an improvement that may be made to reduce the gas cost.

In the following, we show below the related `_cancelRaffle()` routine. As the name indicates, this routine allows the admin to cancel a specific `Raffle`. It comes to our attention that this routine checks the `raffleType` with three distinct `if` statements (lines 308, 312, and 316). The latter two `if` statements can be improved by making them as `else` branch of the previous `if` statement. Note the same improvement can also be made to another routine, i.e., `WinnablesPrizeManager::claimPrize()`

```
306     function _cancelRaffle(uint256 raffleId) internal {
307         RaffleType raffleType = _raffleType[raffleId];
308         if (raffleType == RaffleType.NFT) {
309             NFTInfo storage nftInfo = _nftRaffles[raffleId];
```

```
            _nftLocked[nftInfo.contractAddress][nftInfo.tokenId] = false;
        }
        if (raffleType == RaffleType.TOKEN) {
            TokenInfo storage tokenInfo = _tokenRaffles[raffleId];
            unchecked { _tokensLocked[tokenInfo.tokenAddress] -= tokenInfo.amount; }
        }
        if (raffleType == RaffleType.ETH) {
            unchecked { _ethLocked -= _ethRaffles[raffleId]; }
        }
        _raffleType[raffleId] = RaffleType.NONE;
        emit PrizeUnlocked(raffleId);
    }
```

Listing 3.2: `WinnablesPrizeManager::_cancelRaffle()`

Recommendation Revisit the above logic for improved gas efficiency.

Status The issue has been fixed by this commit: `dd62e8a`.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WinnablesPrizeManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `Winnables` protocol has a built-in `WinnablesPrizeManager` contract to manage `Raffle` prizes. We notice each `Raffle` has its unique id number and this id number has an implicit requirement of being non-zero. Our analysis shows this implicit requirement is better explicitly enforced.

In the following, we show the implementation of the related `lockNFT()` routine. As the name indicates, this routine is used to lock the `NFT`-based `Raffle` prize. While it is a privileged routine and can be called only by the admin user, there is still a need to validate the given `raffleId` is non-zero, i.e., `require(raffleId !=0)`. Note this issue affects a number of routines, including `lockNFT()`, `lockETH()`, `lockTokens()`, and `createRaffle()`.

```
    function lockNFT(
        address ticketManager,
        uint64 chainSelector,
        uint256 raffleId,
        address nft,
        uint256 tokenId
```

```
    ) external onlyRole(0) {
        if (_raffleType[raffleId] != RaffleType.NONE) {
            revert InvalidRaffleId();
        }
        if (IERC721(nft).ownerOf(tokenId) != address(this)) {
            revert InvalidPrize();
        }
        if (_nftLocked[nft][tokenId]) {
            revert InvalidPrize();
        }
        _raffleType[raffleId] = RaffleType.NFT;
        _nftLocked[nft][tokenId] = true;
        _nftRaffles[raffleId].contractAddress = nft;
        _nftRaffles[raffleId].tokenId = tokenId;

        _sendCCIPMessage(ticketManager, chainSelector, abi.encodePacked(raffleId));
        emit NFTPrizeLocked(raffleId, nft, tokenId);
    }
```

Listing 3.3: `WinnablesPrizeManager::lockNFT()`

Recommendation Improve the above-mentioned routines to ensure the given `raffleId` will not be zero.

Status The issue has been fixed by this commit: `dd62e8a`.

3.4 Trust Issue of Admin Keys

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

Description

In the `Winnables` protocol, there is a privileged admin account (with `onlyRole(0)`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., create/cancel raffles, configure various system parameters, and recover funds). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setCCIPCounterpart(
        address contractAddress,
        uint64 chainSelector,
        bool enabled
    ) external override onlyRole(0) {...}

```

```
    /// @notice (Admin) Create NFT Raffle for an prize NFT previously sent to this contract
    /// @param raffleId ID Of the raffle shared with the remote chain
    /// @param startsAt Epoch timestamp in seconds of the raffle start time
    /// @param endsAt Epoch timestamp in seconds of the raffle end time
    /// @param minTickets Minimum number of tickets required to be sold for this raffle
    /// @param maxHoldings Maximum number of tickets one player can hold
    function createRaffle(
        uint256 raffleId,
        uint64 startsAt,
        uint64 endsAt,
        uint32 minTickets,
        uint32 maxTickets,
        uint32 maxHoldings
    ) external onlyRole(0) {...}

    /// @notice (Admin) Cancel a raffle
    /// @param raffleId ID of the raffle to cancel
    function cancelRaffle(address prizeManager, uint64 chainSelector,
            uint256 raffleId) external {...}

    /// @notice (Admin) Withdraw Link or any ERC20 tokens accidentally sent here
    /// @param tokenAddress Address of the token contract
    function withdrawTokens(address tokenAddress, uint256 amount) external onlyRole(0) {...}

    /// @notice (Admin) Withdraw ETH from a canceled raffle or ticket sales
    function withdrawETH() external onlyRole(0) {...}
```

Listing 3.4: Example Privileged Operations in `WinnablesTicketManager`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be better if the privileged account is governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 confirmed by the team. The team intends to manage the admin keys with a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Winnables` protocol, which allows the creation of a new `raffle` with parameters such as duration, minimum entries, maximum entries, and prizes. The prizes are locked before the `raffle` creation, and participants can enter the `raffle` by purchasing entries. Once the `raffle` is concluded, the winners are selected with the help of external randomness provided by `Chainlink VRF`, after which the winners can claim their prizes. The contract also provides functionalities for the participants to withdraw their funds in case the created `raffle` is cancelled. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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] 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.