SMART CONTRACT AUDIT REPORT

for

JustCause Protocol

Prepared By: Xiaomi Huang

PeckShield

July 26, 2022

Document Properties

Client	JustCause
Title	Smart Contract Audit Report
Target	JustCause
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 26, 2022	Xuxian Jiang	Final Release
1.0-rc	July 18, 2022	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 JustCause 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 JustCause

JustCause is a crowdfunding platform that allows users to leverage the power of DeFi to fund causes that are important to them. It uses an innovative funding mechanism to allow users to contribute to public goods, charitable organizations, DAOs, local/global/personal causes, etc. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The JustCause Protocol

Item	Description
Issuer	JustCause
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 26, 2022

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

- https://github.com/smeee23/just_cause.git (6ea9cb7)

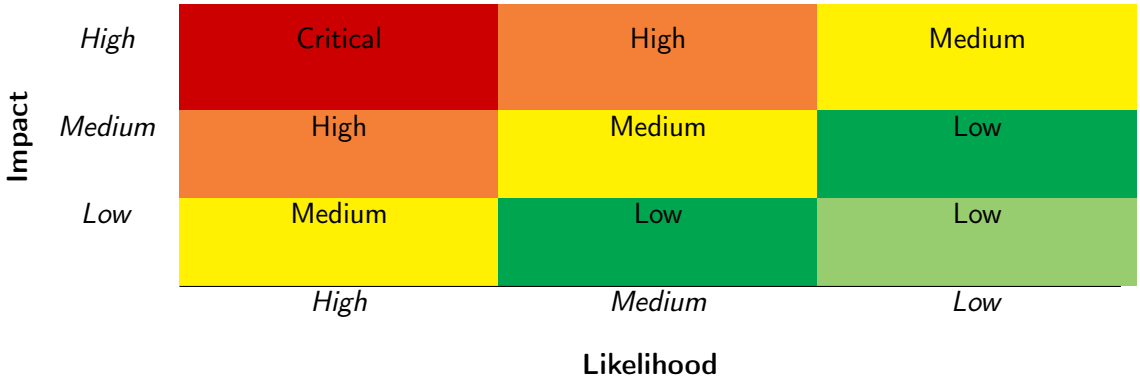
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/smeee23/just_cause.git (1baf536)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

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

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

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

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

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) [4], 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 design and implementation of the `JustCause` 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

We have so far identified a list of potential issues.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 low-severity vulnerability and and 2 informational recommendations.

Table 2.1: Key JustCause Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC2-Compliant Tokens	Coding Practices	Resolved
PVE-002	Informational	Suggested immutable Use in PoolTracker	Coding Practices	Resolved
PVE-003	Informational	Removal of Unused Event in PoolTracker	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, 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 details.

3 Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `PoolTracker`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
64    function transfer(address _to, uint _value) returns (bool) {
65        //Default assumes totalSupply can't be over max (2^256 - 1).
66        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67            balances[msg.sender] -= _value;
68            balances[_to] += _value;
69            Transfer(msg.sender, _to, _value);
70            return true;
71        } else { return false; }
72    }

74    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.1: ZRX.sol

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. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `addDeposit()` routine in the `PoolTracker` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transferFrom(msg.sender, address(this), _amount)` (line 154) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
135     function addDeposit(
136         uint256 _amount,
137         address _asset,
138         address _pool,
139         bool _isETH
140     ) onlyPools(_pool) nonReentrant() external payable {
141         tvl[_asset] += _amount;
142         string memory _metaHash = IJustCausePool(_pool).getMetaUri();
143         address poolAddr = IPoolAddressesProvider(poolAddressesProviderAddr).getPool();
144
145         if(_isETH){
146             require(IWETHGateway(wethGatewayAddr).getWETHAddress() == _asset, "asset
                    does not match WETHGateway");
147             require(msg.value > 0, "msg.value cannot be zero");
148             IWETHGateway(wethGatewayAddr).depositETH{value: msg.value}(poolAddr, _pool,
                    0);
149         }
150         else {
151             require(msg.value == 0, "msg.value is not zero");
152             IERC20 token = IERC20(_asset);
153             require(token.allowance(msg.sender, address(this)) >= _amount, "sender not
                    approved");
154             token.transferFrom(msg.sender, address(this), _amount);
155             token.approve(poolAddr, _amount);
156             IPool(poolAddr).deposit(address(token), _amount, _pool, 0);
157         }
158         IJustCausePool(_pool).deposit(_asset, _amount);
```

```
159         bool isFirstDeposit = IJCDepositorERC721(IJustCausePool(_pool).getERC721Address
                ()).addFunds(msg.sender, _amount, block.timestamp, _asset, _metaHash);
160         if(isFirstDeposit){
161             contributors[msg.sender].push(_pool);
162         }
163         emit AddDeposit(msg.sender, _pool, _asset, _amount);
164     }
```

Listing 3.2: `PoolTracker::addDeposit()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`. For the safe-version of `approve()`, there is a need to `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new allowance.

Status This issue has been fixed in the following commit: `9d62f25`.

3.2 Suggested immutable Use in PoolTracker

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PoolTracker`
- Category: Coding Practices [3]
- CWE subcategory: CWE-561 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `PoolTracker` contract, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency. Examples include `baseJCPool`, `baseERC721`, `poolAddressesProviderAddr`, and `wethGatewayAddr`.

```
36 contract PoolTracker is ReentrancyGuard {
37
```

```
38      JustCausePool baseJCPool;
39      JCDepositorERC721 baseERC721;
40
41      //contract addresses will point to bool if they exist
42      mapping(address => bool) private isPool;
43      mapping(string => address) private names;
44      mapping(address => uint256) private tvl;
45      mapping(address => uint256) private totalDonated;
46      mapping(address => address[]) private contributors;
47      mapping(address => address[]) private receivers;
48      address[] private verifiedPools;
49      uint256[5] private fees;
50      uint256 bpFee;
51      address multiSig;
52
53      address poolAddressesProviderAddr;
54      address wethGatewayAddr;
55      ...
56  }
```

Listing 3.3: Storage Variable Defined `PoolTracker`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status This issue has been fixed in the following commit: `9d62f25`.

3.3 Removal of Unused Event in PoolTracker

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PoolTracker`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [2]

Description

The `JustCause` protocol makes good use of a number of reference contracts, such as `ERC20`, `ReentrancyGuard`, and `Clones`, to facilitate its code implementation and organization. For example, the `PoolTracker` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `PoolTracker` contract, there is an event that is defined for testing purpose and should be removed for production deployment. Specifically, this event `Test` is currently not used in the protocol and can be safely removed.

```
56      event AddPool(address pool, string name, address receiver);
57      event AddDeposit(address userAddr, address pool, address asset, uint256 amount);
58      event WithdrawDeposit(address userAddr, address pool, address asset, uint256 amount)
            ;
59      event Claim(address userAddr, address receiver, address pool, address asset, uint256
            amount);
60      event Test(address[] aaveAccepted, address[] causeAccepted );
```

Listing 3.4: Example Events Defined in `PoolTracker`

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

Status This issue has been fixed in the following commit: `9d62f25`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the JustCause protocol, which is a crowdfunding platform that allows users to leverage the power of DeFi to fund causes that are important to them. It uses an innovative funding mechanism to allow users to contribute to public goods, charitable organizations, DAOs, local/global/personal causes, etc. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that Solidity-based 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-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

[4] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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