THE ARCADIA GROUP

XenGame
Security Audit Report

PREPARED FOR:

Xen.Game

ARCADIA CONTACT INFO
Email: audits@arcadiamgroup.com
Telegram: https://t.me/thearcadiagroup

Revision history

Date	Reason	Commit
9/28/2023	Initial Audit Scope	#54d16f2a23f13a8b6da81d34fc0eae87c5f681e0
10/12/2023	Review Of Remediations	#be1bec06cb254afc002a6844b562d085270a157a

Table of Contents

Executive Summary

Introduction

XENGame engaged Arcadia to perform a security audit of their main smart contracts version 2. Our review of their codebase occurred on the commit hash #54d16f2a23f13a8b6da81d34fc0eae87c5f681e0

Review Team

1. Tuan "Anhnt" Nguyen - Security Researcher and Engineer
2. Joel Farris - Project Manager

Project Background

XENGame is a decentralized jackpot game on Ethereum. Buy keys to extend the timer, and the last keyholder wins. Earn passive rewards, burn keys to withdraw. Revenue burns XEN tokens. No developer fees, admin keys, or pre-mined tokens, ensuring fairness.

Coverage

For this audit, we performed research, test coverage, investigation, and review of XENGame's main contract followed by issue reporting, along with mitigation and remediation instructions as outlined in this report. The following code repositories, files, and/or libraries are considered in scope for the review.

Contracts	Lines	nLines	nSLOC	Comment Lines	Complex. Score
src/XenGameV2.sol	1337	1297	655	401	358

Methodology

Arcadia completed this security review using various methods, primarily consisting of dynamic and static analysis. This process included a line-by-line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

The followings are the steps we have performed while auditing the smart contracts:

- Investigating the project and its technical architecture overview through its documentation
- Understanding the overview of the smart contracts, the functions of the contracts, the inheritance, and how the contracts interface with each others thanks to the graph created by Solidity Visual Developer
- Manual smart contract audit:
    - Review the code to find any issue that could be exploited by known attacks listed by Consensys
    - Identifying which existing projects the smart contracts are built upon and what are the known vulnerabilities and remediations to the existing projects
    - Line-by-line manual review of the code to find any algorithmic and arithmetic related vulnerabilities compared to what should be done based on the project's documentation
    - Find any potential code that could be refactored to save gas
    - Run through the unit-tests and test-coverage if exists
- Static Analysis:
    - Scanning for vulnerabilities in the smart contracts using Static Code Analysis Software
    - Making a static analysis of the smart contracts using Slither
- Additional review: a follow-up review is done when the smart contracts have any new update. The follow-up is done by reviewing all changes compared to the audited commit revision and its impact to the existing source code and found issues.

Summary

There were **10** issues found, **0** of which were deemed to be ‘critical’, and **2** of which were rated as ‘high’. At the end of these issues were found throughout the review of a rapidly changing codebase and not a final static point in time.

Severity Rating	Number of Original Occurrences	Number of Remaining Occurrences
CRITICAL	0	0
HIGH	2	0
MEDIUM	3	1
LOW	1	1
INFORMATIONAL	4	4

Findings in Manual Audit

(XG-1) Using checkForEarlyKeys() may prevent players from receiving earlyKeys in a Round.

Status

Resolved

Risk Level

Severity: High, likelihood: Low.

Code Segment

```
function withdrawRewards(uint256 roundNumber) public {
   // Get the player's storage reference
   Player storage player = players[msg.sender];
   // Convert the player's address to a payable address
   address payable senderPayable = payable(msg.sender);
   // Check for early keys received during the early buy-in period
   checkForEarlyKeys();
  …
}

function processRewards(uint256 roundNumber) private {
   Player storage player = players[msg.sender];
   // Check for early keys received during the early buy-in period
   checkForEarlyKeys();
  …
}
function checkForEarlyKeys() private {
    if (players[msg.sender].earlyBuyinPoints[currentRound] > 0 &&
!earlyKeysReceived[msg.sender][currentRound]) {
         uint256 totalPoints = rounds[currentRound].earlyBuyinEth;
         uint256 playerPoints = players[msg.sender].earlyBuyinPoints[currentRound];
```

Description

Inside the **checkForEarlyKeys** function, **currentRound** is used to check **earlyBuyinPoints**. However, functions that utilize **checkForEarlyKeys**, such as **withdrawRewards** and **processRewards**, can potentially execute logic for a different round. As a result, using **checkForEarlyKeys** may lead to gas wastage, and in a scenario like the following, users may miss checking for early rewards:

- User A performs an early buy-in.
- They wait for the round to end.
- User B extends the round.
- User B purchases some keys.
- They wait for the round to end again.
- User B triggers a new round.

In this sequence of steps, User A will never receive their earlyKeys.

Recommendation

Conduct a thorough review of the game logic.

(XG-2) Inconsistent referralReward calculations.

Status

Resolved

Risk Level

Severity: High, likelihood: High.

Code Segment

```
function buyWithReferral(
    string memory _referrerName,
    uint256 _numberOfKeys
) public payable {
…
     uint256 referralReward = (msg.value * REFERRAL_REWARD_PERCENTAGE) /
10000; // 10% of the incoming ETH
…
}
```

```
function distributeFunds(uint256 _amount) private {
    // Calculate the referral reward as a percentage of the incoming ETH
    uint256 referralReward = (_amount * REFERRAL_REWARD_PERCENTAGE) / 10000;
```

Description

While distributing funds, the **_amount** represents the total cost to purchase keys, which is consistently lower than the **msg.value**. As a result, the **referralReward** inside the **distributeFunds** function is consistently less than the real reward for the referrer, causing all distributed funds to be incorrectly calculated.

Recommendation

Discover a method for consistently computing the referralReward.

(XG-3) calculateMaxKeysToPurchase gets reverted when a new round starts.

Status

Resolved

Risk Level

Severity: Medium, likelihood: Medium

Code Segment

```
function calculateMaxKeysToPurchase(uint256 _amount) public view returns
(uint256 maxKeys, uint256 totalCost) {
     // Fetch the initial price of a key
     uint256 initialKeyPrice = getKeyPrice();
     // If the user's amount is less than the price of a single key, return as
no keys can be bought
    if (_amount < initialKeyPrice) {
       return (0, 0);
    }
    uint256 left = 0;
    uint256 right = _amount / initialKeyPrice;
```

Description

The public function **calculateMaxKeysToPurchase** encounters a revert when starting a new round due to the absence of initialization for **round.lastKeyPrice** (the value returned in **getKeyPrice()**). This issue could potentially confuse users when they attempt to calculate the maximum number of keys to purchase.

Recommendation

Initialize **round.lastKeyPrice** when starting a new round.

(XG-4) Arithmetic overflow/underflow when calling getPlayerInfo.

Status

Acknowledged

Risk Level

Severity: Medium, likelihood: Medium

Description

Scenario

- Start a new round
- Disregard the early buy-in time frame.
- Purchase keys.
- Call the **getPlayerInfo** function.

Recommendation

Initialize **round.rewardRatio** when starting a new round.

(XG-5) calculatePriceForKeys return zero when a new round starts.

Status

Resolved

Risk Level

Severity: Medium, likelihood: Medium

Code Segment

```
function calculatePriceForKeys(uint256 _keys) public view returns (uint256
totalPrice) {
     uint256 initialKeyPrice = getKeyPrice();
     uint256 increasePerKey = 0.000000009 ether;

    // Calculate the total price based on the number of keys
    if (_keys <= 1) {
       totalPrice = initialKeyPrice * _keys;
    } else {
       uint256 lastPrice = initialKeyPrice + ((_keys - 1) * increasePerKey);
     totalPrice = (_keys * (initialKeyPrice + lastPrice)) / 2;
    }
…
```

Description

The public function **calculatePriceForKeys** returns zero when users intend to purchase a single key, even though they cannot complete the purchase. This issue has the potential to confuse users who attempt to buy a single key but are unable to do so.

Recommendation

Initialize **round.lastKeyPrice** when starting a new round.

(XG-6) Improve early-stage reward withdrawal error message.

Status

Acknowledged

Risk Level

Severity: Medium, likelihood: Medium

Description

When users purchase keys during the early buying period and attempt withdrawal directly through the smart contract (rather than using the UI), the response should not generate an **'Arithmetic over/underflow'** error message.

Recommendation

Return a more friendly error message.

(XG-7) Divide before multiply

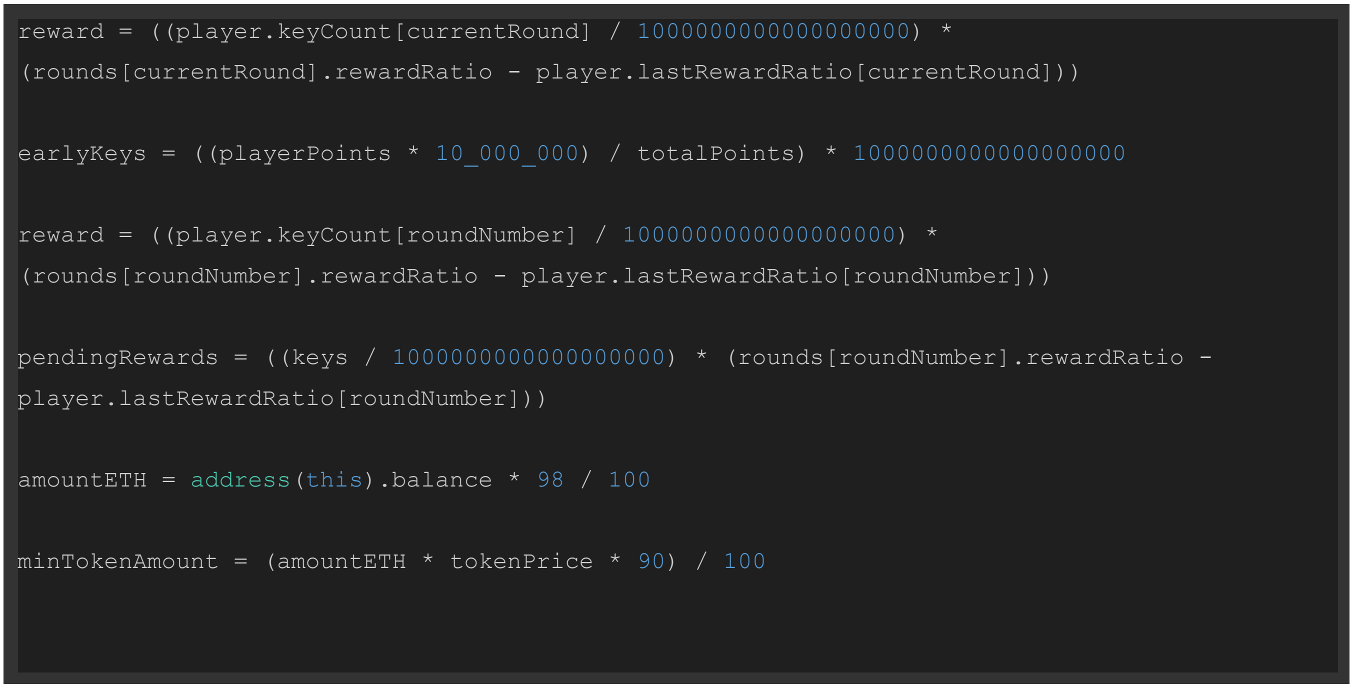
Status

Acknowledged

Risk Level

Severity: Informational, likelihood: High.

Code Segment



```
reward = ((player.keyCount[currentRound] / 1000000000000000000) *
(rounds[currentRound].rewardRatio - player.lastRewardRatio[currentRound]))

earlyKeys = ((playerPoints * 10_000_000) / totalPoints) * 1000000000000000000

reward = ((player.keyCount[roundNumber] / 1000000000000000000) *
(rounds[roundNumber].rewardRatio - player.lastRewardRatio[roundNumber]))

pendingRewards = ((keys / 1000000000000000000) * (rounds[roundNumber].rewardRatio -
player.lastRewardRatio[roundNumber]))

amountETH = address(this).balance * 98 / 100

minTokenAmount = (amountETH * tokenPrice * 90) / 100
```

Description

Solidity's integer division truncates. Thus, performing division before multiplication can lead to precision loss.

Recommendation

Consider ordering multiplication before division.

(XG-8) Game logic when the round not active

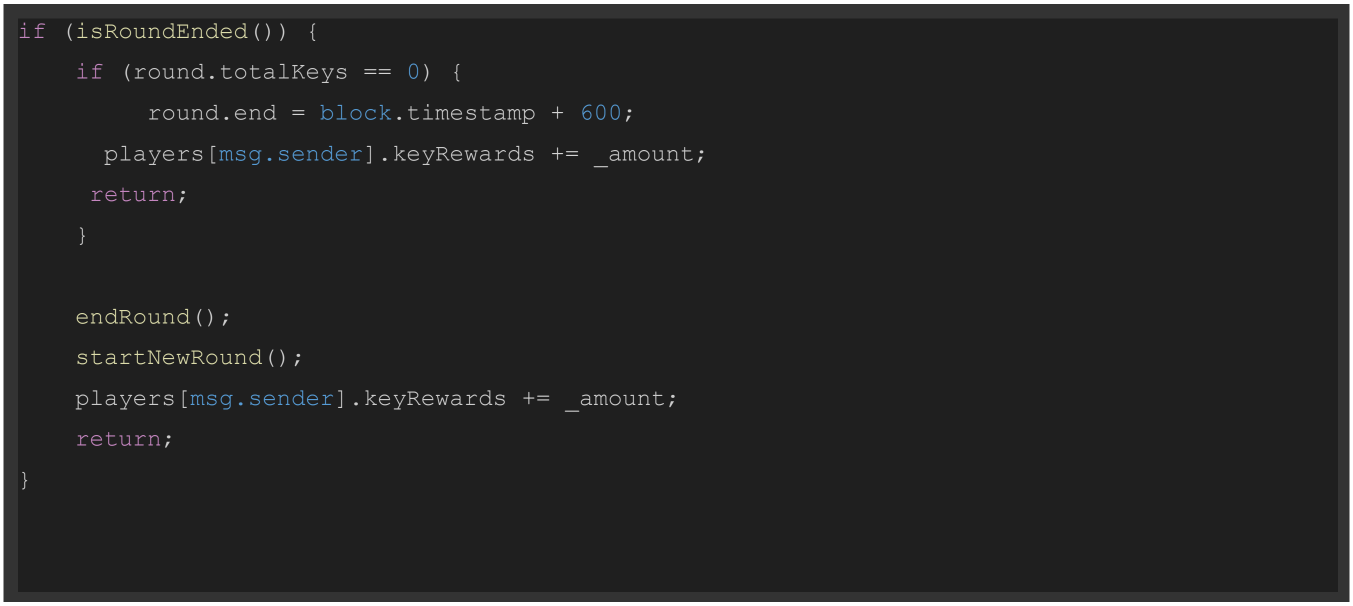
Status

Acknowledged

Risk Level

Severity: Informational, likelihood: High.

Code Segment



```
if (isRoundEnded()) {
    if (round.totalKeys == 0) {
         round.end = block.timestamp + 600;
      players[msg.sender].keyRewards += _amount;
     return;
    }

    endRound();
    startNewRound();
    players[msg.sender].keyRewards += _amount;
    return;
}
```

Description

When a player attempts to purchase keys during isRoundEnded(), they don't actually receive any keys but still incur gas costs to either extend or end the round. Even if a new round is triggered, they must wait for another user to make a purchase during the regular buying window to withdraw their funds. This logic is counterintuitive and can be confusing for users.

Recommendation

Conduct a thorough review of the game logic.

(XG-9) Enhancing code clarity and reducing redundant checks with modifiers

Status

Acknowledged

Risk Level

Severity: Informational, likelihood: High.

Description

Implement Modifiers (isRoundActive and isRoundEnded) for buyWithReferral, buyKeysWithRewards, calculateMaxKeysToPurchase, calculatePriceForKeys, and getKeyPrice Functions to Eliminate Redundant Internal Checks.

(XG-10) Code improvement

Status

Acknowledged

Risk Level

Severity: Informational, likelihood: High.

Description

The code quality could be improved by

- Eliminate the hardcoding of non-existent contract addresses, as this contract cannot be directly deployed on the mainnet.
- Aligning with strict coding conventions, such as maintaining consistent function naming (uppercase/lowercase) and utilizing the `? :` syntax for **`getKeyPrice`**.

- It's advisable to remove unnecessary comments when the code is self-explanatory.
- Notably, there is duplicated logic present in some functions (e.g., **`getUniquePlayers/getRoundUniquePlayers`**, **`getPlayerAddresses/playerAddresses`**).
- Furthermore, for improved efficiency, it is recommended to index event parameters.

There is ample room for enhancement in ensuring consistent adherence to coding conventions.

Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.