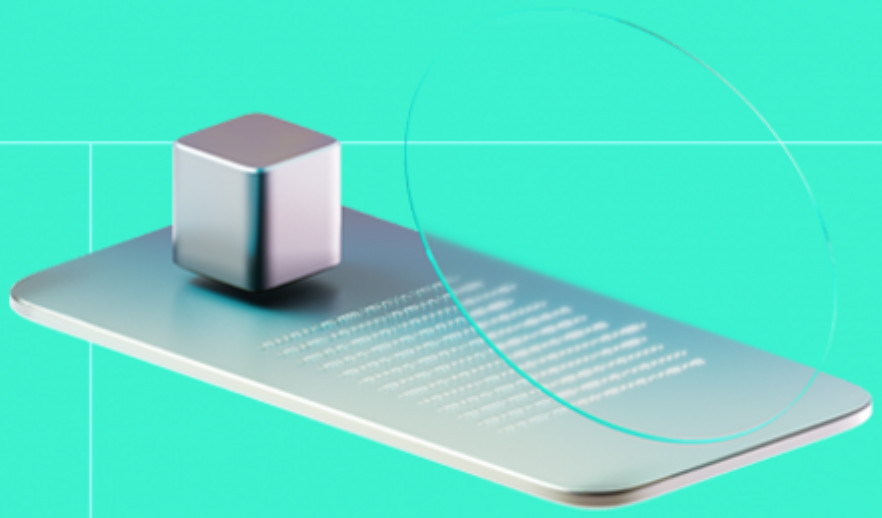
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: SpaceCatch

Date: 19/06/2024

We express our gratitude to the SpaceCatch team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

SpaceCatch is a GameFi project that is designed and developed by the professional gaming studio PIXELFIELD.

Document

Name	Smart Contract Code Review and Security Analysis Report for SpaceCatch
Audited By	Turgay Arda Usman
Approved By	Grzegorz Trawinski
Website	https://whitepaper.spacecatch.io
Changelog	14/06/2024 - Preliminary Report
	19/06/2024 - Final Report
Platform	Ethereum Mainnet
Language	Solidity
Tags	Staking, Vesting, ERC20, Airdrop
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/development-at-pixelfield/spacecatch-smart-contracts
Commit	fc5eba4a500a8bf621662ce2ce41a54bb2d5ec58

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

4	2	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	0
Low	3

Vulnerability	Status
F-2024-3931 - Lock Period is not Taken Into Account During Stake Claims	Mitigated
F-2024-3929 - Airdrops Can Be Set For a Past Date	Accepted
F-2024-3928 - Unchecked Transfer	Fixed
F-2024-3930 - Recovery Functionality Only Works for Registered Token Type	Fixed

Documentation quality

- Functional requirements are partially provided.
- Technical description is partially provided.

Code quality

- The code mostly follows best practices and style guidelines.
  - See observations and low issues for more details.
- The development environment is configured.

Test coverage

Code coverage of the project is **62.38%** (branch coverage),

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missed.
- Interactions by several users are not tested thoroughly.

Table of Contents

System Overview

SpaceCatch is a GameFi project that is designed and developed by the professional gaming studio PIXELFIELD. It has the following contracts

CatchAirdrop — manages a phased distribution of $CATCH tokens to specified beneficiaries.

Catch — Simple ERC20 token It has the following attributes:

- Name: Catch
- Symbol: CATCH
- Decimals: 18
- Total supply: 100m tokens.

CatchStaking — Contains the staking logic.

CatchVesting — Contains the vesting logic.

BatchTransfer —This contract allows the owner to transfer tokens to multiple address in a single transaction.

Privileged roles

- The Vesting Operator can upload vesting schedules for a specific round and set claimed amounts for the previous version integration.
- The airdrop admin can set the token allocations for a list of beneficiaries and start the airdrop season.
- The staking admin can start the staking, recover tokens, and send back stakes to stakers in case staking is cancelled.
- The staking manager can set the earned rewards for initial stakers and add additional stakes for the users.
- The Vesting admin can recover stuck tokens.

Risks

- Making external calls within loops increases the risk of gas exhaustion, potentially leading to failed transactions and reduced contract reliability, especially when processing large datasets.
- The project iterates over large dynamic arrays, which leads to excessive gas costs, risking denial of service due to out-of-gas errors, directly impacting contract usability and reliability.
- The token contract’s design allows for centralized control over the transfer process, posing a risk of unauthorized token issuance, potentially diluting the token value and undermining trust in the project's economic governance
- The project does not support non-standard ERC20 tokens. Adding such tokens in the future can cause additional risks.
- The current version of the code does not support fee-on-transfer tokens. Adding such tokens in the future can create risk

Findings

Vulnerability Details

F-2024-3931 - Lock Period is not Taken Into Account During Stake Claims - High

Description:

The project allows CATCH tokens to be staked. The staking has the following features:

> "Since we need to be able to set the staking contract with the data from the old contract, the staking contract has been modified to allow for this. The staking contract will be initiated with the stakes already made in the old contract, and also data about which staker claimed how much staking rewards. All the contract will be set with the same start time as the old contracts, so they will act as if they were the old contracts.
>
> added functions `setEarnedForInitialStakers()` and `addInitialStakes()` which will be used to set the staking contract with the data from the old contract.
>
> The `startStaking()` function now accepts argument `uint256 startTimestamp` which will be used to set the start time of the staking contract. The start time will be set to the same time as the old contract. Another argument `uint256 _rewardIndexUpdatedAt` in order to set up the reward computation argument so that it continues from the point just before the exploit of the old contract."

The staking logic contains a 90 days lock period where the staked amount locked and cannot be claimed. However, the calculation of rewards does not depend on the lock period. Instead, it calculates the entire earned reward up to the point of claiming. Here is a detailed analysis of how this works:

```
function claim()
    external
    nonReentrant
    stakingNotCanceled
    stakingStarted
    returns (uint256)
{
    _updateRewardIndex(block.timestamp);
    /// Compute the rewards earned by the staker between last time this function
    _updateRewards(msg.sender);
```

```
    uint256 reward = earned[msg.sender];
    require(reward > 0, "No rewards available for claim");

    earned[msg.sender] = 0;

    catchToken.safeTransfer(msg.sender, reward);
    emit Claimed(msg.sender, reward);

    return reward;
}
```

- The `claim` function calls `_updateRewardIndex(block.timestamp)` to ensure the global reward index is up-to-date with the current timestamp.
- The `_updateRewards` function is then called, which calculates the total rewards earned by the user up to the current timestamp. This function sums up the rewards for all stakes of the user, regardless of whether they are still locked or unlocked.

```
function _updateRewards(address account) private {
    uint256 _earned = earned[account];
    for (uint256 i = 0; i < stakes[account].length; i++) {
        _earned += _calculateRewardsForStake(account, i, rewardIndex);
    }
    addressRewardIndex[account] = rewardIndex;
    earned[account] = _earned;
}
```

- The `_calculateRewardsForStake` function calculates the rewards for each individual stake based on the difference between the current reward index and the user's last updated reward index.

```
function _calculateRewardsForStake(
    address _address,
    uint256 stakeIndex,
    uint256 _rewardIndex
    ) private view returns (uint256) {
    /// If the stake is already withdrawn, return 0
    if (stakes[_address][stakeIndex].withdrawn == true) return 0;

    uint256 staked = stakes[_address][stakeIndex].amount;

    return
    (staked * (_rewardIndex - addressRewardIndex[_address])) /
```

```
    MULTIPLIER;
}
```

The current implementation does not differentiate between locked and unlocked stakes when calculating rewards. This means:

- Users can claim the entire earned reward at any time, including during the lock period.
- The amount claimed is based on the total rewards earned up to the current timestamp, not the lock period status.

This setup might lead to users being able to claim the full reward even if their stakes are still locked.

Assets:

- StakingNew.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Mitigated

Classification

Impact: 4/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Modify the reward calculation logic to account for the lock period.

Resolution: This is a well known flow and it is reported to be feature:

> Being able to claim rewards even for locked stakes is correct according to requirements of the project. Locked stake just means that the stake cannot be unstaked for 90 days, but rewards can be claimed even in this locked period.

Evidences

PoC

Reproduce:

- At `T0`, Alice stakes 1000 tokens.
- At `T0 + 30 days`, Bob stakes 2000 tokens.
- At **`T0 + 60 days`**:
    - Alice has been staking for 60 days.
    - Bob has been staking for 30 days.
    - The rewards are distributed based on the total staked amount (3000 tokens in total).
    - Suppose the reward index at this point is calculated and let's assume the accumulated reward for simplicity is 600 tokens.
- Claim Rewards at `T0 + 60 days`:
    - Alice decides to claim her rewards at `T0 + 60 days`
    - Alice's share: ⅓ , Bob's share: ⅔.
    - Total rewards accumulated at `T0 + 60 days`: 600 tokens.
        - Alice's rewards: 600 * ⅓. ⇒ 200 tokens
        - Bob's rewards: 600*⅔. ⇒ 400 tokens.

Results:

In this scenario, Alice is able to claim her full reward amount (200 tokens) at `T0 + 60 days` even though her stake is still locked for another 30 days.

F-2024-3928 - Unchecked Transfer - Low

Description: The `BatchTransfer.sol` contract use the `transfer()` and `transferFrom()` functions of the ERC20 implementation of OpenZeppelin. Not all `IERC20` implementations `revert` when there is a failure in `transfer`/`transferFrom`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually making a payment.

```
function batchTransfer(
    address[] calldata recipients,
    uint256[] calldata amounts
) external onlyOwner {
    require(
    recipients.length == amounts.length,
    "Arrays must have the same length"
    );


    for (uint256 i = 0; i < recipients.length; i++) {
        uint256 amount = amounts[i];
        address recipient = recipients[i];


        token.transferFrom(msg.sender, recipient, amount)
    }
}

function recoverTokens(uint256 amount) external onlyOwner {
     token.transfer(msg.sender, amount);
}
```

Status: Fixed

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Use the SafeERC20 library to interact with tokens safely.

Resolution: Fixed in commit `3415f36`. The SafeERC20 library is utilized.

F-2024-3929 - Airdrops Can Be Set For a Past Date - Low

Description:

The `Airdrop.sol` contract allows owners to distribute CATCH tokens via airdrop. The `startAirdrop()` allows the owner to enable the airdrop process by setting its timestamp.

```
function startAirdrop(
    uint256 _airdropStartTimestamp
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(airdropStartTimestamp == 0, "Airdrop already started.");
    require(_airdropStartTimestamp > 0, "Invalid Airdrop start timestamp");
    require(
    totalAirdropAmount == TOTAL_AIRDROP_QUOTA,
    "Distributed amount mismatch."
    );
    airdropStartTimestamp = _airdropStartTimestamp;
}
```

As it can be seen from the implementation, the code allows owners to start an airdrop with a past date. This is due to the lack of checks applied to the given timestamp. The code should check if a given timestamp is newer than the current system epoch timestamp.

This can lead to unexpected behaviors and incorrect payments for the system.

Assets:

- Airdrop.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Accepted

Classification

Impact: 2/5

Likelihood: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement date checks for the airdrop start times.

Resolution: The client acknowledged the issue with the following statement:

> This is intended functionality. The reason is that we need this contract to have the same state as the old contract is being replaced due to the aforementioned hack. So the airdrop start time will be set to a past timestamp intentionally

F-2024-3930 - Recovery Functionality Only Works for Registered Token Type - Low

Description:

The `BatchTransfer.sol` contract aims to forward funds from the old version of the project to the current version. It contains a recovery functionality for the accidental token deposits.

```
function recoverTokens(uint256 amount) external onlyOwner {
   token.transfer(msg.sender, amount);
}
```

The `recover()` function can only return the registered ERC20 token to owners. However, the contract allows all kind of ERC20 tokens to be sent. This would mean the other tokens will be locked and cannot be returned to their rightful owners.

Assets:

- BatchTranfer.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Accept the token address as a parameter.

Resolution: Fixed in commit `3415f36`. The ERC20 token address is now a parameter.

Observation Details

F-2024-3925 - Floating Pragma - Info

Description: The project uses floating pragma `^0.8.23;`

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Assets:

- Airdrop.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- BatchTranfer.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- Catch.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- StakingNew.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- Vesting.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Fixed

Recommendations

Remediation: Lock the pragma version and consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Resolution: Fixed in commit `fdb302e`. The Solidity version is locked to `0.8.23`.

F-2024-3926 - Missing Zero Address Validation - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "**zero address**". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.

The "**Missing zero address Validation**" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.

For instance, consider a contract that includes a function to change its owner. This function is crucial, as it determines who has administrative access. However, if this function lacks proper validation checks, it might inadvertently permit the setting of the owner to the zero address. Consequently, the administrative functions will become unusable.

Missing checks were observed in the following functions:

- batchTransfer()

Assets:

- BatchTranfer.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Accepted

Recommendations

Remediation: Implement zero address checks for the aforementioned functions.

Resolution: The client acknowledged the issue with the following statement:

> We won't be including zero address validation to BatchTransfer in order to keep gas costs low as possible.

F-2024-3927 - Contract And File Name Mismatch - Info

Description:

The project has the files with the names `Airdrop.sol`, `StakingNew.sol` and `Vesting.sol`. These files' contract names, `CatchStaking` , `CatchVesting`, and `CatchAirdrop` do not match the file names .

This leads to the inability to run the project tests due to such inconsistencies.

Assets:

- Airdrop.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- StakingNew.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]
- Vesting.sol [https://github.com/development-at-pixelfield/spacecatch-smart-contracts]

Status: Accepted

Recommendations

Remediation: It is recommended to update the smart contract names.

Resolution: The client acknowledged the issue with the following statement:

> I don't want to change the file names now because there is another audit in progress, and I don't want to complicate things because they have their audit scope defined by the file names. Is this issue limiting your ability to run tests

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/development-at-pixelfield/spacecatch-smart-contracts
Commit	fc5eba4a500a8bf621662ce2ce41a54bb2d5ec58
Whitepaper	https://whitepaper.spacecatch.io
Requirements	Provided as files.
Technical Requirements	Provided as files.

Contracts in Scope
Airdrop.sol
BatchTranfer.sol
Catch.sol
StakingNew.sol
Vesting.sol