Zellic

MightyNet

Smart Contract Security Assessment

May 12, 2023

Prepared for:

Fadzuli Said

Mighty Bear Games

Prepared by:

Daniel Lu, Sina Pilehchiha, and Yuhang Wu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Mighty Bear Games on May 8th, 2023. During this engagement, Zellic reviewed MightyNet's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can the system be exploited by players? For example, is it possible for a player to receive rewards from staking while still being able to sell or transfer their locked tokens?
- Is there a risk of tokens being incorrectly restricted or becoming unrecoverable for players?
- Are there any other security vulnerabilities or concerns that need to be addressed?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

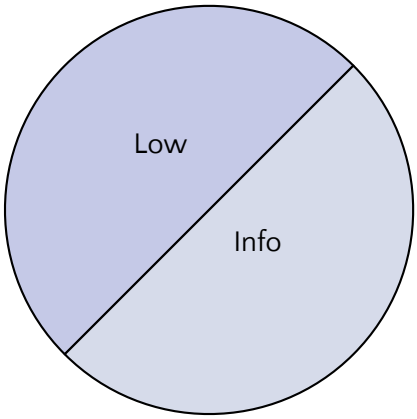
1.3 Results

During our assessment on the scoped MightyNet contracts, we discovered four findings. No critical issues were found. Of the four findings, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Mighty Bear Games's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	2



2 Introduction

2.1 About MightyNet

MightyNet is a brand new Web3 game ecosystem built by Mighty Bear Games.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign

it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

MightyNet Contracts

Repository	https://github.com/MightyBear/PolarSmartContracts
Version	PolarSmartContracts: `a53198fb95afcfdb0e0ac202458ed4f727719e18`
Programs	• ERC721Restrictable • MightyNetERC721RestrictedRegistry
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with one consultant for a total of one person-day. The assessment was conducted over the course of one calendar day.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Daniel Lu, Engineer
daniel@zellic.io

Sina Pilehchiha, Engineer
sina@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 8, 2023	Start of primary review period
May 9, 2023	End of primary review period

3 Detailed Findings

3.1 Missing registry check in `restrict`

- **Target**: MightyNetERC721RestrictedRegistry
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

Each ERC721Restrictable token has an associated registry contract for managing restrictions. The `restrict` function in MightyNetERC721RestrictedRegistry does not check whether the contract itself is set as the target token's registry.

```
function restrict(
    address tokenContract,
    uint256[] calldata tokenIds
) external override onlyRole(RESTRICTOR_ROLE) nonReentrant whenNotPaused {
    uint256 tokenCount = tokenIds.length;
    if (tokenCount == 0) {
        revert InvalidTokenCount(tokenCount);
    }
    for (uint256 i = 0; i < tokenCount; ++i) {
        uint256 tokenId = tokenIds[i];
        if (!ERC721Restrictable(tokenContract).exists(tokenId)) {
            revert InvalidToken(tokenContract, tokenId);
        }
        bytes32 tokenHash = keccak256(
            abi.encodePacked(tokenContract, tokenId)
        );
        if (_isRestricted(tokenHash)) {
            revert TokenAlreadyRestricted(tokenContract, tokenId);
        }
        _tokenRestrictions[tokenHash] = msg.sender;
    }
    emit Restricted(tokenContract, tokenIds);
}
```

Impact

This behavior would exacerbate upgrade or configuration issues in other contracts that interact with ERC721Restrictable tokens. If a contract tries to restrict a token using the incorrect registry contract, the action will fail silently. This might allow users to earn rewards on unlocked tokens.

Recommendations

The `restrict` function should include an assertion that `restrictedRegistry` in the token contract indeed matches `address(this)`. Alternatively, Mighty Bear Games could add a separate `safeRestrict` function that includes this check.

Remediation

Mighty Bear Games acknowledges this finding.They added an assertion as recommended to the beginning of the scope of the `restrict` function in the `V2` contract. If the assertion fails, it reverts with a newly introduced error `ContractNotUsingThisRestrictedRegistry(address tokenContract)`.

Mighty Bear Games has provided the response below:

> by adding the registry check to MightyNetERC721RestrictedRegistryV2.
>
> MightyNetERC721RestrictedRegistry was already deployed on ethereum.
>
> MightyNetERC721RestrictedRegistryV2 will be deployed on our L2 chain.

3.2 Restriction pattern creates centralization risk

- **Target**: MightyNetERC721RestrictedRegistry
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The MightyNetERC721RestrictedRegistry contract gives approved users or contracts the ability to restrict specific tokens.

```
function restrict(
    address tokenContract,
    uint256[] calldata tokenIds
) external override onlyRole(RESTRICTOR_ROLE) nonReentrant whenNotPaused {
    uint256 tokenCount = tokenIds.length;
    if (tokenCount == 0) {
        revert InvalidTokenCount(tokenCount);
    }
    for (uint256 i = 0; i < tokenCount; ++i) {
        uint256 tokenId = tokenIds[i];
        if (!ERC721Restrictable(tokenContract).exists(tokenId)) {
            revert InvalidToken(tokenContract, tokenId);
        }
        bytes32 tokenHash = keccak256(
            abi.encodePacked(tokenContract, tokenId)
        );
        if (_isRestricted(tokenHash)) {
            revert TokenAlreadyRestricted(tokenContract, tokenId);
        }
        _tokenRestrictions[tokenHash] = msg.sender;
    }
    emit Restricted(tokenContract, tokenIds);
}
```

Any address with the `RESTRICTOR_ROLE` can invoke this function to restrict any token in any token contract, without approval by users. Further, only the address that added a token's restriction is able to remove the restriction.

Impact

This exposes all assets to risks in approved contracts. If any such contracts experience key compromises, upgrade issues, or implementation vulnerabilities, then arbitrary assets might become locked. Additionally, this restriction pattern requires that both the admin and all approved contracts are highly trusted by users.

Recommendations

We recommend that Mighty Bear Games

- implement a system where users first approve restrictions,
- use token transfers to hold staked assets, or
- clearly document trust assumptions associated with restrictors.

Remediation

Mighty Bear Games acknowledges this. As per their response, they have decided to go with the third recommendation and clearly document trust assumptions with restrictors.

The following is their provided response:

> We are updating the readme to address this. Once the next deployment is done, we are planning to create a public github repository so that this readme can be player facing. We will also link to the repository from the whitepaper and a public facing FAQ page.

3.3 Unnecessary complexity in `_tokenRestrictions` structure

- **Target**: MightyNetERC721RestrictedRegistry
- **Category**: Code Maturity
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: Informational

Description

The MightyNetERC721RestrictedRegistry contract tracks restricted tokens by hashing a token's `tokenContract` address and `tokenId` value together, resulting in a `tokenHash` that is then stored in the contract.

For instance, in the `isRestricted` function:

```
bytes32 tokenHash = keccak256(abi.encodePacked(tokenContract, tokenId));
```

Then, `tokenHash` is used as a key for the `_tokenRestrictions` mapping to account for restricted tokens.

```
mapping(bytes32 => address) private _tokenRestrictions;
```

Together, they are used in the form `_tokenRestrictions[tokenHash]` multiple times in the contract.

Impact

This adds unnecessary complexity in terms of maintainability and readability of the contract code. Additionally, the current implementation consumes slightly more gas than is needed.

Recommendations

We recommend using a traditional nested mapping in order to improve the maintainability, readability, and gas efficiency of the contract:

```
mapping(address => mapping(uint256 => address))
    private _tokenRestrictions;
```

Then, the state of a given token can be accessed with `_tokenRestrictions[tokenContract][tokenId]`.

Remediation

might Bear Games acknowledges this. They have replaced their previously implemented `_tokenRestrictions` structure with a more efficient one as per our recommendation.

The following is theire provided response:

> We have made this change to MightyNetERC721RestrictedRegistryV2.

3.4 Redundant reserved __gap space

- **Target**: MightyNetERC721RestrictedRegistry
- **Category**: Code Maturity
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: Informational

Description

The reserved __gap space is unnecessary:

```
contract MightyNetERC721RestrictedRegistry ... {

 ...
    uint256[50] private __gap;
}
```

This is because the MightyNetERC721RestrictedRegistry contract is designed to be called directly, rather than through inheritance. As a result, the __gap storage space at the end of the contract remains unused, as there are no concerns of storage layout conflicts arising from future variable additions.

Impact

The redundant reserved space can lead to some inefficiency, as the contract has allocated resources that are not being used and maintaining unnecessary code in a contract might result in slightly higher deployment and execution gas costs due to the larger bytecode size.

Recommendations

We recommend removing the __gap array.

Remediation

might Bear Games acknowledges this. They have removed the redundant reserved storage space in the V2 version of the contract.

The following is theire provided response:

> We have made this change to MightyNetERC721RestrictedRegistryV2.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Contracts are upgradeable

As all contracts in scope are upgradeable, this inherently creates some level of risk for the protocol and its users. Therefore, we strongly recommend that Mighty Bear Games maintains transparency regarding governance and the protocol's trust assumptions by clearly documenting them on their website and in developer documentation. This will help users better understand the protocol's operations and any potential risks associated with using it, ultimately improving overall user trust and adoption.

4.2 Transfer restriction logic omitted in ERC721Restrictable

The ERC721Restrictable abstract contract provides hooks for checking whether a token is restricted, but using these checks to restrict transfers is left as the responsibility of concrete implementations. This may result in some duplicated logic and maintenance risk. We recommend adding the relevant logic to this or another abstract class for future assets to inherit.

4.3 Documentation

Though the contracts in scope are thoroughly tested, there is little explicit documentation of their behavior. Some design choices may result in future misuse: for instance, the `onlyAllowUnrestricted` modifier in ERC721Restrictable checks only whether a given token has been restricted, regardless of whether it actually exists. We recommend adding comments about behavior and design choices for both developers and users.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: MightyNetERC721RestrictedRegistry.sol

Function: `initialize()`

Initializes the contract.

Branches and code coverage (including function calls)

Intended branches

- Properly initializes the contract and gives the sender the admin role.
  ☑ Test coverage

Negative behavior

- Reverts if the contract is already initialized.
  ☐ Negative test

Function: `pause()`

Allows the admin to pause token restriction and unrestriction.

Branches and code coverage (including function calls)

Intended branches

- Restricting and unrestricting tokens no longer works when paused.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not the admin.
  ☑ Negative test

Function: `restrict(address tokenContract, uint256[] tokenIds)`

Given a token contract and list of IDs, marks them as restricted.

Inputs

- `tokenContract`
  - **Control**: Fully controllable by caller.
  - **Constraints**: Must successfully receive calls to `exists`.
  - **Impact**: The contract must be able to check if a token exists.
- `tokenIds`
  - **Control**: Fully controllable by caller.
  - **Constraints**: Must be non-empty.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- Properly restricts tokens if each token is valid and not restricted.
  ☑ Test coverage

Negative behavior

- Reverts if the `tokenIds` list is empty.
  ☑ Negative test
- Reverts if any of the tokens does not exist in the given contract.
  ☑ Negative test
- Reverts if any of the tokens is already restricted.
  ☑ Negative test
- Reverts if the caller is not a restrictor.
  ☑ Negative test

Function call analysis

- `restrict` → `ERC721Restrictable(tokenContract).exists`
  - **What is controllable?** The target contract and the token ID.
  - **If return value controllable, how is it used and how can it go wrong?** Used to check if the token exists.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function will revert.

Function: `unpause()`

Allows the admin to unpause the contract.

Branches and code coverage (including function calls)

Intended branches

- Restricting and unrestricting tokens works again when unpaused.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not the admin.
  ☑ Negative test

Function: `unrestrict(address tokenContract, uint256[] tokenIds)`

Given a token contract and list of IDs, marks them as unrestricted.

Inputs

- `tokenContract`
  - **Control**: Fully controllable by caller.
  - **Constraints**: Must successfully receive calls to `exists`.
  - **Impact**: The contract must be able to check if a token exists.
- `tokenIds`
  - **Control**: Fully controllable by caller.
  - **Constraints**: Must be non-empty.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- Successfully restricts tokens if all conditions are met.
  ☑ Test coverage

Negative behavior

- Reverts if the `tokenIds` list is empty.
  ☑ Negative test
- Reverts if any of the tokens does not exist in the given contract.
  ☑ Negative test
- Reverts if any of the tokens is not restricted.

☑ Negative test

- Reverts if the caller is not the original restrictor.
  ☑ Negative test
- Reverts if the caller is not a restrictor.
  ☑ Negative test

Function call analysis

- `unrestrict` → `ERC721Restrictable(tokenContract).exists`
  - **What is controllable?** The target contract and the token ID.
  - **If return value controllable, how is it used and how can it go wrong?** Used to check if the token exists.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function will revert.

6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet EVM.

During our assessment on the scoped MightyNet contracts, we discovered four findings. No critical issues were found. Two were of low impact and the remaining findings were informational in nature. Mighty Bear Games acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.