MightyNetERC1155Claimer

Smart Contract Security Assessment

July 25, 2023

Prepared for:

Fadzuli Said

Mighty Bear Games

Prepared by:

Nipun Gupta

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 from July 24th to July 25th, 2023. During this engagement, Zellic reviewed MightyNetERC1155Claimer'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:

- Are there any vulnerabilities enabling users to claim multiple times?
- Can an attacker prevent legitimate users from claiming their tokens?
- Are Merkle proofs properly validated?

1.2 Non-goals and Limitations

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

- Off-chain components
- 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.

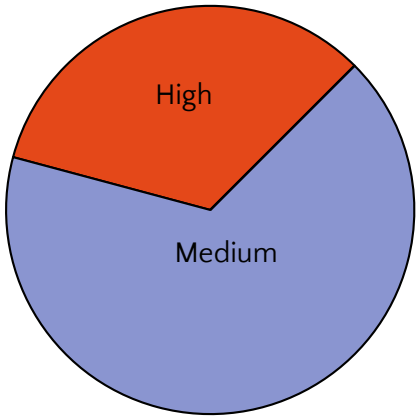
1.3 Results

During our assessment on the scoped MightyNetERC1155Claimer contracts, we discovered three findings. No critical issues were found. One was of high impact and two were of medium impact.

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	1
Medium	2
Low	0
Informational	0



2 Introduction

2.1 About MightyNetERC1155Claimer

MightyNetERC1155Claimer is a contract that allows eligible users to claim items from a MightyNetERC1155 contract.

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 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations —– found in the Discussion (4) section of the document —– may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

MightyNetERC1155Claimer Contracts

Repository	https://github.com/MightyBear/PolarSmartContracts
Version	PolarSmartContracts: `9e912e6a979d707f543bb502602bf82654aaa140`
Program	MightyNetERC1155Claimer
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 consultant was engaged to conduct the assessment:

Nipun Gupta, Engineer
nipun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 24, 2023	Start of primary review period
July 25, 2023	End of primary review period

3 Detailed Findings

3.1 Risk of unintended token minting

- **Target**: MightyNetERC1155Claimer
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The leaf nodes of the Merkle tree contain only the user addresses and do not include the `tokenId` or the address of `mnERC1155` to be minted. As a result, a user can potentially use a Merkle proof expected for minting tokens with `tokenId x` to mint tokens with `tokenId y`, or even mint tokens on an entirely different mnERC1155 contract.

Here is an example. In this scenario, we will consider `tokenId y` to be more valuable than `tokenId x`, and the `claimWhitelist` array contains a Merkle root where the user is eligible to mint `n` number of tokens with `tokenId x`.

The potential issue arises from the fact that even though it might be expected for the user to call `claim` using the correct Merkle proof to mint their `x` tokens, they might choose not to do so and instead wait for the admin to change the `tokenId` using the function `setTokenId`. If the admin later changes the `tokenId` from `x` to `y`, the user can now simply call `claim` to claim `n` number of `y` tokens instead of `x` tokens.

This behavior could lead to unintended economic consequences, as the user could take advantage of the situation to obtain more valuable `y` tokens rather than the originally intended `x` tokens.

Impact

If either `setTokenId` or `setMightyNetERC1155Address` is called to change the `tokenId` or `mnERC1155` address before all the tokens are claimed, and the `claimWhitelist` is not fully cleared out, it could potentially result in the minting of different tokens than originally expected.

Recommendations

To address this issue, it is recommended to ensure that `claimWhitelist` is completely cleared out before invoking `setTokenId` or `setMightyNetERC1155Address`. By doing so, any potential misuse of old Merkle proofs to mint new tokens can be prevented. Alternatively, you can consider including the `tokenId` and the address of ERC-1155 in the

Merkle trees, which can also help mitigate the problem.

Remediation

This issue has been acknowledged by Mighty Bear Games.

Mighty Bear Games provided the following response:

> We acknowledge the concerns related to the possibility of unintended token minting. However, it's important to note that this contract is designed for a specific use case, where only one item from one collection can be claimed. We assure you that we will not reuse the same contract for multiple claim or mint events. Instead, for each new event, a fresh contract will be deployed.
>
> The reason for implementing the SetTokenId function is to maximize flexibility in case we encounter any misconfigurations or issues after deployment. Should any problems arise, we will be able to pause the contract, make the necessary adjustments, and then resume its functionality.

3.2 Possible DOS while claiming ERC-1155

- **Target**: MightyNetERC1155Claimer
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: Medium
- **Impact**: Medium

Description

The `claimWhitelist` array stores the `MerkleProofWhitelist` struct containing the root hash of the Merkle tree. Each element in the array corresponds to a specific number of claimable tokens, and the Merkle tree contains addresses eligible to mint that number of tokens.

If this array is large enough, the users that have a large amount of claimable tokens would need to spend too much gas to claim their tokens or the function `claim` might entirely revert for them due to exceeding the gas limit, as the code loops through the dynamic array.

```
function claim(bytes32[] calldata merkleProof){
...
    uint256 size = claimWhitelist.length;
    bool whitelisted = false;
    uint256 toMint = 0;
    for (; toMint < size; ++toMint) {
        if (claimWhitelist[toMint].isWhitelisted(msg.sender,
    merkleProof)) {
            whitelisted = true;
            break;
        }
    }
...
}
```

Impact

The transaction might fail if the `claimWhitelist` array becomes too large and the gas exceeds the maximum gas limit. Additionally, users with a substantial number of claimable tokens would be required to spend a significant amount of gas to execute the transaction successfully. This gas consumption can become burdensome for users with a large number of tokens to claim.

Recommendations

Consider modifying the `claim` function to accept the mint amount as an argument and use it directly to calculate the array index where `isWhitelisted` should be called. This adjustment can improve the efficiency of the function and avoid unnecessary iterations through the `claimWhitelist` array, especially in scenarios with a large number of claimable tokens.

Remediation

This issue has been acknowledged by Mighty Bear Games.

Mighty Bear Games provided the following response:

> We have assessed the gas costs associated with claiming different amounts of ERC-1155 tokens, and our findings indicate that the increase in cost follows a linear pattern. We have taken this into consideration while designing the claiming process. Additionally, it is important to note that we have set a limit on the maximum number of tokens that can be claimed to just 3.

3.3 No storage gap for upgradable contract

- **Target**: MightyNetERC1155Claimer
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The MightyNetERC1155Claimer contract is intended to be the logic contract behind a certain proxy contract. Implementation contracts should have a storage gap to allow for upgradability / adding new variables in future upgrades, which are missing in these contracts.

Impact

If care is not taken, new variables added in a child contract could accidentally overwrite variables in its parent contract.

Recommendations

Consider adding an appropriate storage gap at the end of upgradable contracts such as this below.

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

Remediation

This issue has been acknowledged by Mighty Bear Games.

Mighty Bear Games provided the following response:

> We don't intend to inherit from this contract.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Unnecessary modifier, `onlyWhitelisted`

The modifier `onlyWhitelisted` is empty and serves no purpose since the Merkle proofs are already being verified in the `isWhitelisted` function, which is subsequently called. To simplify the code and avoid unnecessary checks, the `onlyWhitelisted` modifier can be safely removed.

```
modifier onlyWhitelisted(
        address user,
        bytes32[] calldata merkleProof,
        Whitelists.MerkleProofWhitelist[] storage whitelist
    ) {
        _;
    }
```

This issue has been acknowledged by Mighty Bear Games, and fixes were implemented in the following commits:

- 6dfe0373
- cf1cca87

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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: MightyNetERC1155Claimer.sol

Function: `amountClaimable(address user, byte[32][] merkleProof)`

The function is used to calculate the number of tokens a user can mint, provided the correct Merkle proof.

Inputs

- `user`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: N/A.
- `merkleProof`
  - **Control**: Fully controlled.
  - **Constraints**: Should be a valid Merkle proofs array.
  - **Impact**: Used to determine the amount of tokens to mint for the user, if the proofs are valid.

Branches and code coverage (including function calls)

Intended branches

- Anyone should be able to check the claimed amount, provided the correct Merkle proof.
  ☑ Checked
- Should return 0 if the user has already claimed.
  ☑ Checked

Negative behavior

- If the wrong proof is provided, the amount should be zero.
  ☑ Checked

Function: `claim(byte[32][] merkleProof)`

The function used by users to claim their mnERC1155 tokens.

Inputs

- `merkleProof`
  - **Control**: Fully controlled.
  - **Constraints**: Should be a valid Merkle proofs array.
  - **Impact**: Used to determine the amount of tokens to mint for the user, if the proofs are valid.

Branches and code coverage (including function calls)

Intended branches

- Users should be able to claim if the correct proof is provided.
  ☑ Checked

Negative behavior

- Users cannot claim multiple times.
  ☑ Checked
- Revert if user is not whitelisted.
  ☐ Unchecked

Function call analysis

- `claim -> mnERC1155.mint(msg.sender, tokenId, toMint + 1)`
  - **What is controllable?** `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the `mint` call reverts, the entire function would revert. Check-effects-interactions along with `nonReentrant` modifier is implemented correctly, so reentrancy is not an issue.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped MightyNetERC1155Claimer contracts, we discovered three findings. No critical issues were found. One was of high impact and two were of medium impact. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.