Zellic

MUNI

Smart Contract Security Assessment

May 31, 2022

Prepared for:

Adrian Li

DFX Finance

Prepared by:

Aaron Esau and Vlad Toie

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About MUNI

MUNI (“Managed Uniswap v3”) allows liquidity positions to be represented using a fungible ERC20 token.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform’s design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract’s possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase 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 forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

MUNI Core Contracts

Repository	https://github.com/dfx-finance/asc
Versions	`6e67e4b63ccd6a91f66c5e3cbf1aecaffca3e096`
Programs	• StakingRewards • MUNI • MUNILogicV1 • MUNIState
Type	Solidity
Platform	EVM-compatible

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

May 23, 2022	Start of primary review period
May 27, 2022	End of primary review period

1.6 Disclaimer

This assessment does not provide any warranties on 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 on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for DFX Finance from May 23rd to May 27th, 2022, on the scoped contracts and discovered 3 findings. Fortunately, no critical issues were found. We applaud DFX Finance for their attention to detail and diligence in maintaining high code quality standards in the development of MUNI.

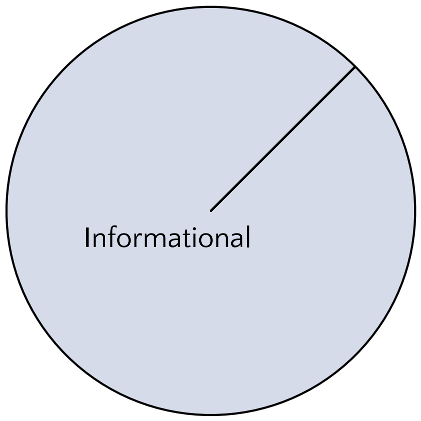
All 3 findings were informational in nature. Additionally, Zellic recorded its notes and observations from the audit for DFX Finance's benefit at the end of the document.

Zellic thoroughly reviewed the MUNI codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (1.2) of this document. Specifically, taking into account MUNI's threat model, we focused heavily on issues that would lead to loss of pool funds or arbitrary minting of MUNI tokens.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	3



3 Detailed Findings

3.1 Multiple contracts provide a function to renounce ownership

- **Target**: StakingRewards, MUNIState, MUNI
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `StakingRewards`, `MUNIState`, and `MUNI` contracts implement Ownable, which provides a method named `renounceOwnership` that removes the current owner (reference). This is likely not a desired feature.

Impact

If `renounceOwnership` were called, the contract would be left without an owner.

Recommendation

Override the `renounceOwnership` function:

```
function renounceOwnership() public {
    revert(“This feature is not available.”);
}
```

Remediation

DFX Finance acknowledged this finding and created a fix in pull request #35.

3.2 Lack of interfaces for `MUNILogicV1` and `MUNI`

- **Target**: MUNILogicV1, MUNI
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Interfaces for the public APIs of `MUNILogicV1` and `MUNI` do not exist.

Impact

Interactions with smart contracts may be more difficult; it is a composability issue for future developers who want to build upon or understand the codebase.

Recommendation

We recommend adding all exposed/public APIs to interfaces in a way that accurately reflects the underlying code.

Remediation

DFX Finance acknowledged this finding and created a fix in pull request #37.

3.3 Centralization risk in `StakingRewards` and `MUNI` and `MUNILogicV1`

- **Target**: StakingRewards, MUNI, MUNILogicV1
- **Category**: Business Logic
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

Using the `setRewardsDuration` function, the owner can set the duration to a low value to redeem rewards quickly. The rewards must be claimed in a separate block because the rewards duration cannot be 0; if it were, the following code in `notifyRewardAmount` would result in a division by zero error—preventing rewards variables from being updated:

```
if (block.timestamp ≥ periodFinish) {
    rewardRate = reward / rewardsDuration;
} else {
    uint256 remaining = periodFinish - block.timestamp;
    uint256 leftover = remaining * rewardRate;
    rewardRate = (reward + leftover) / rewardsDuration;
}
```

Although any user has the same ability to claim rewards as the owner after the owner modifies the rewards duration, only the owner has the ability to cause reward tokens to change in value at any time by simply changing the rewards duration.

For example, By changing the duration to a low value, the supply of rewards tokens can quickly increase—causing the token to be worth less.

Additionally, the `StakingRewards` contract does not own the reward token; that is, a different party owns it and can mint tokens.

There is also an artificial limit on the amount of rewards the `StakingRewards` contract can provide because transfers would eventually fail when the contract's balance is zero.

Impact

A malicious or compromised owner could potentially drain the pool or render the rewards token worthless.

Recommendations

Consider using a timelock for additional safety.

Also, the `StakingRewards` contract should own and manage the rewards token itself.

Remediation

DFX Finance acknowledged this finding and created a fix in pull request #40.

4 Discussion

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

4.1 Incomplete test coverage

Testing coverage is adequate but not comprehensive. There are a few functions where there is very little or no coverage and where the testing is not comprehensive.

We recommend ensuring complete coverage of the following functions because they are integral to the protocol:

- In **StakingRewards**, all the functions that deal with rewards (both issuance and transfer): `stake`, `withdraw`, `getReward`.
- In **MUNI** and **MUNILogicV1**, all the functions that deal with rebalancing: `executiveRebalance`, `rebalance`.

DFX Finance acknowledged this issue and created a fix in pull request #41.

4.2 Problems in code documentation

We noted a few references to `gelato` in the code documentation, for example in variables' names:

```
Muni.sol/ MUNILogicV1.sol

/// @notice withdraw manager fees accrued, only gelato executors can call.
/// Target account to receive fees is managerTreasury, alterable by
    manager.
/// Frequency of withdrawals configured with gelatoWithdrawBPS, alterable
    by manager.
function withdrawManagerBalance(uint256 feeAmount, address feeToken)
    external
{
```

We recommend changing the variable names in the code documentation so that they match the code to improve clarity.

We also recommend ensuring all code has documentation. For example, adding documentation to the **StakingRewards** contract would allow non-technical users to understand how the contract works. Documentation improves code composability for developers.

DFX Finance acknowledged this issue and created a fix in pull request #40.

4.3 Unchecked external call returns

In the `MUNI.sol` file, the `mint` function calls `pool.mint` without checking the return values of the function call:

```
function mint(uint256 mintAmount, address receiver)
    external
    nonReentrant
    returns (
        uint256 amount0,
        uint256 amount1,
        uint128 liquidityMinted
    )
{
    // ...
    } else {
        // if supply is 0 mintAmount == liquidity to deposit
        (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
            sqrtRatioX96,
            lowerTick.getSqrtRatioAtTick,
            upperTick.getSqrtRatioAtTick,
            SafeCast.toUint128(mintAmount)
        );
    }

    // transfer amounts owed to contract
    if (amount0 > 0) {
        token0.safeTransferFrom(msg.sender, address(this), amount0);
    }
    if (amount1 > 0) {
        token1.safeTransferFrom(msg.sender, address(this), amount1);
    }

    // ...
    pool.mint(address(this), lowerTick, upperTick, liquidityMinted, “”);
```

Although this is not necessarily a security issue, the `pool.mint` call returns the amount of `amount0` of `token0` and `amount1` of `token1` that the caller paid for the subsequent mint. Should these values not match the amount the protocol asked the user to send in the following code, the contract would incur monetary loss:

```
// ...

// transfer amounts owed to contract
if (amount0 > 0) {
    token0.safeTransferFrom(msg.sender, address(this), amount0);
}
if (amount1 > 0) {
    token1.safeTransferFrom(msg.sender, address(this), amount1);
}

// ...
```

We recommend adding an additional check just to ensure that what was asked from the user is indeed the amount that was finally paid to the issuing `pool.mint`.

```
// ...
(paidAmount0, paidAmount1) = pool.mint(address(this), lowerTick,
    upperTick, liquidityMinted, “”);

require(paidAmount0 == amount0 && paidAmount1 == amount1,
   “Mint prices differ”);
```

DFX Finance acknowledged this issue and created a fix in pull request #39.