Zellic

EtherFi

Smart Contract Security Assessment

February 27, 2023

Prepared for:

Vecheslav Silagadze

Gadze Finance SEZC

Prepared by:

Syed Faraz Abrar

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 Gadze Finance SEZC on February 27th, 2023. During this engagement, Zellic reviewed EtherFi'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 customer deposits safe?
- Can the customers withdraw their deposits at any time?
- Is it possible for user funds to get locked in any way?

1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

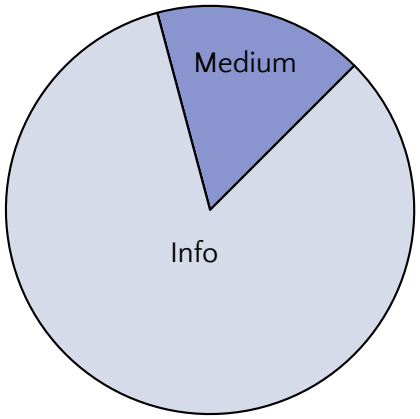
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped EtherFi contracts, we discovered six findings. No critical issues were found. Of the six findings, one was of medium impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	5



2 Introduction

2.1 About EtherFi

EtherFi is a decentralized and noncustodial Ethereum staking protocol. The contract being audited is the EarlyAdopterPool contract. It enables customers to deposit and withdraw while tracking points (for future benefits in their larger protocol project).

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:

EtherFi Contracts

Repository	https://github.com/GadzeFinance/dappContracts
Version	dappContracts: `dfe36485c3ef91f9e428b4d7c09503f38efb0e7b`
Program	• EarlyAdopterPool
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar, Engineer
faraz@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

February 27, 2022	Kick-off call
February 27, 2022	Start of primary review period
February 27, 2022	End of primary review period

3 Detailed Findings

3.1 The `claimReceiverContract` variable is not fully validated

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

Description

When the user is claiming funds through the `claim()` function, all of the user's deposited funds are sent to the `claimReceiverContract`, which is set by the owner. This is a storage variable that is set using the `setClaimReceiverContract()` function.

Within the `setClaimReceiverContract()` function, the only validation done on the address of the contract is to ensure that it is not `address(0)`. This validation is not enough, as it is possible for the owner to set the address to a contract that is not able to transfer out any ETH or ERC20 tokens that it receives. In this instance, the user's funds would be lost forever.

Impact

There is a risk that user funds may become permanently locked either by accident or as a result of deliberate actions taken by a malicious owner.

Recommendations

See Ethereum Improvement Proposal EIP-165 for a way to determine whether a contract implements a certain interface. This will prevent the owner from making a mistake, but it will not prevent a malicious owner from locking user funds forever.

Alternatively, consider not allowing this contract address to be modified by the owner. It should be made immutable. If the receiver contract's implementation needs to change in the future, consider using a proxy pattern to do that.

Remediation

Gadze Finance SEZC acknowledged this finding and stated that they understand the risk, but have mitigated it by ensuring that multiple parties are involved when setting the receiver contract. Their official response is produced below.

> The receiver contract has not been set yet and will be set through multiple parties being involved with the decision, we do understand the risk however, we have mitigated this with multiple parties being involved. We do understand it only takes 1 address to make the call and this is a risk.

3.2 Using values from emitted events may not be fully accurate

- **Target**: EarlyAdopterPool
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `getContractTVL()` function uses the contract's balance of ERC20 tokens and Ether to determine the TVL of the pool.

```
function getContractTVL() public view returns (uint256 tvl) {
    tvl = (rETHInstance.balanceOf(address(this)) +
        wstETHInstance.balanceOf(address(this)) +
        sfrxETHInstance.balanceOf(address(this)) +
        cbETHInstance.balanceOf(address(this)) +
        address(this).balance);
}
```

This function is then used when emitting events related to TVL.

Impact

The issue is that the balance of the ERC20 tokens in the contract, as well as the balance of Ether in the contract, can be manipulated by any user by sending tokens / Ether directly to the contract (as opposed to going through the `deposit()` function). Therefore, depending on the TVL values returned from this function (and, by extension, emitted through the events such as `ERC20TVLUpdated`) may be inaccurate.

Without knowing how the values emitted through these events are used off chain, it is impossible to determine the impact.

Recommendations

Consider tracking the balance of tokens and Ether in the contract separately through storage variables. This will prevent directly transferred tokens and Ether from being counted towards the TVL.

Otherwise, ensure that the values emitted by TVL-related events are not used for critical operations off chain.

Remediation

Gadze Finance SEZC acknowledged this finding and stated that they want to include any funds sent to the contract to be included in the TVL. Their official response is produced below.

> We understand the issue revolving an inaccurate TVL due to the contract being able to receive funds through direct transfers, however, we would still like to include any funds sent to the contract in our total value locked.

3.3 Magic numbers should be replaced with immutable constants

- **Target**: EarlyAdopterPool
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

There are a number of places in the code where magic numbers are used. For brevity's sake, the following is a list of line numbers where magic numbers are being used:

- 187
- 218
- 225 to 228
- 233 to 235

Impact

The use of magic numbers makes the code confusing, both for the developers in the future and for auditors.

Recommendations

Consider replacing magic numbers with immutable constants.

In instances where the magic number is used as a flag to determine which branch a function should take, consider using either an enum or separating the logic out into multiple functions.

Remediation

Gadze Finance SEZC acknowledged this finding and stated that they are not worried about this issue. The finding was partially remediated by refactoring the way the magic numbers are used in commit ebd3f11a. Their official response is produced below.

> The magic numbers have either been marked immutable or removed and simplified. It was often the use of the numbers to simplify large numbers for the reader.

3.4 Use the correct function modifiers

- **Target**: EarlyAdopterPool
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `withdraw()` function is marked as `payable`. This is incorrect as it does not make use of any ETH that it might (accidentally or otherwise) receive.

The `withdraw()` and `claim()` functions are marked as `public`, although they are not used anywhere else in the contract.

Impact

Functions that are marked as `payable` expect that ETH may be received. If the function does not account for this, then users may accidentally send ETH when invoking these functions, leading to a loss of funds.

Functions that are only called externally should be marked as `external`.

Recommendations

Remove the `payable` modifier from the `withdraw()` function.

Replace the `public` modifier with the `external` modifier in the `withdraw()` and `claim()` functions.

Remediation

Gadze Finance SEZC acknowledged and partially remediated this finding by removing the `payable` modifier from the `withdraw()` function in commit b7be224c.

3.5 Use safe ERC20 functions

- **Target**: EarlyAdopterPool
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

This contract makes use of the ERC20 `transfer()` and `transferFrom()` functions. Not all ERC20 tokens adhere to the standard definition of these functions.

Impact

The tokens that are used in this contract (rETH, wstETH, sfrxETH, cbETH) all adhere to the ERC20 token standard, so there is no impact. However, the cbETH token contract specifically uses a proxy pattern, which means that the contract is upgradable. If it were ever to upgrade to a new implementation where the `transfer()` or `transferFrom()` functions did not adhere to the standard anymore, then the contract would stop functioning.

Recommendations

Consider replacing the use of `transfer()` and `transferFrom()` with the safe ERC20 `safeTransfer()` and `safeTransferFrom()` functions.

Remediation

Gadze Finance SEZC acknowledged this issue and contacted the Coinbase team to ensure there were no planned upgrades to the cbETH token contract that would change the `transfer()` and `transferFrom()` function definitions. The Coinbase team confirmed that this was the case. Their official response is produced below.

> The ERC20 tokens being used with transfers have been checked by the team and they all follow the same patterns. Due to these being the only tokens used, with no option to add others, we are happy with the implementation.

3.6 Unused variables should be removed

- **Target**: EarlyAdopterPool
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following storage variables are not used anywhere in the contract:

1. `SCALE`
2. `multiplierCoefficient`

In the `calculateUserPoints()` function, the `numberOfMultiplierMilestones` variable is initialized but not used:

```
function calculateUserPoints(address _user) public view returns (uint256)
    {
    // [ ... ]

    //Variable to store how many milestones (3 days) the user deposit
    lasted
    uint256 numberOfMultiplierMilestones = lengthOfDeposit / 259200;

    if (numberOfMultiplierMilestones > 10) {
        numberOfMultiplierMilestones = 10;
    }

    // [ ... ]
}
```

Impact

Unused variables introduce unnecessary complexity to the code and may lead to programmer error in the future.

Recommendations

Remove the variables unless there are plans to use them in the future.

Remediation

The `SCALE` variable was removed in commit 8d080521

The `multiplierCoefficient` variable was removed in commit 1e5e61bc

The `numberOfMultiplierMilestones` variable was removed in commit f23285b7

4 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.

4.1 Module: EarlyAdopterPool.sol

Function: `claim()`

Used to claim user funds.

Branches and code coverage (including function calls)

Intended branches

- Allows user to claim rewarded funds successfully.
  ☑ Test coverage

Negative behavior

- Should fail if claiming is not open.
  ☑ Negative test
- Should fail if the `claimReceiverContract` is not set.
  ☑ Negative test
- Should fail if the `claimDeadline` has been reached.
  ☑ Negative test
- Should fail if the user has not deposited anything.
  ☐ Negative test

Function: `depositEther()`

Used to deposit Ether into the contract.

Branches and code coverage (including function calls)

Intended branches

- User is able to deposit Ether successfully.
  ☑ Test coverage
- The correct events are successfully emitted.

Negative behavior

- Deposit should fail if claiming is open (i.e., depositing is closed).
  ☐ Negative test

Function: `deposit(address _erc20Contract, uint256 _amount)`

Used to deposit ERC20 tokens into the contract.

Inputs

- `_erc20Contract`
  - **Control**: Fully controlled.
  - **Constraints**: Must be one of the whitelisted tokens (rETH, sfrxETH, wstETH, cbETH).
  - **Impact**: This is the token that is transferred out of the user's wallet to this contract.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: Must be between `minDeposit` (0.1 Ether) and `maxDeposit` (100 Ether).
  - **Impact**: This is the amount of tokens transferred out of the user's wallet to this contract.

Branches and code coverage (including function calls)

Intended branches

- User is successfully able to deposit all four tokens into the contract.
  ☑ Test coverage
- The correct events are successfully emitted.
  ☑ Test coverage

Negative behavior

- Deposit should fail if the user provides an unsupported token contract address.
  ☐ Negative test
- Deposit should fail if claiming is open (i.e., depositing is closed).
  ☐ Negative test

Function call analysis

- `deposit` → `_erc20Contract.transferFrom(msg.sender, address(this), _amount)`
  - **What is controllable?**: `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If it reverts, nothing happens. If it reenters, no harm can be done as the checks-effects-interactions pattern is used.

Function: `setClaimReceiverContract(address _receiverContract)`

Sets the contract that will receive claimed funds.

Inputs

- `_receiverContract`
  - **Control**: Fully controlled.
  - **Constraints**: Cannot be `address(0)`.
  - **Impact**: User funds are transferred to this contract when funds are claimed.

Branches and code coverage (including function calls)

Intended branches

- The claim receiver contract address is set successfully.
  ☑ Test coverage
- The required events are emitted.
  ☐ Test coverage

Negative behavior

- Should fail if not called by the owner.
  ☑ Negative test
- Should fail if the address of the contract is address(0).
  ☑ Negative test

Function: `setClaimingOpen(uint256 _claimDeadline)`

Sets claiming to open with a specified `_claimDeadline`.

Inputs

- `_claimDeadline`

- **Control**: Fully controlled.
- **Constraints**: N/A.
- **Impact**: Claiming will close when this deadline is reached.

Branches and code coverage (including function calls)

Intended branches

- Should open claiming and set the deadline successfully.
  ☐ Test coverage
- Should emit the required events successfully.
  ☐ Test coverage

Negative behavior

- Should fail if not called by the contract owner.
  ☑ Negative test

Function: `withdraw()`

Used to withdraw all funds the user may have deposited into this contract.

Branches and code coverage (including function calls)

Intended branches

- User is able to withdraw funds successfully.
  ☑ Test coverage

5 Audit Results

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

During our audit, we discovered six findings. Of these, one was of medium risk and five were suggestions (informational). Gadze Finance SEZC acknowledged all findings and implemented fixes for some of them.

5.1 Disclaimers

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 additional code added to the assessed 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 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.