Zellic

Prepared for
Jacob Firek
EtherFi

Prepared by
Filippo Cremonese
Sylvain Pelissier
Zellic

September 16, 2024

Cash contracts Pre-order Update
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for EtherFi from September 13th to September 15th, 2024. During this engagement, Zellic reviewed Cash contracts Pre-order Update's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

The goal of this security assessment was to ensure the changes made in PR #13 did not introduce any new security issues in the existing contract.

1.3. 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.

1.4. Results

During our assessment on the scoped Cash contracts Pre-order Update contracts, we discovered two findings. No critical issues were found. One finding was of high impact and the other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of EtherFi in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	0
🟩 Low	0
⬜ Informational	1

2. Introduction

2.1. About the project

EtherFi contributed the following description of the project:

> Ether.fi ↗ is a liquid restaking protocol that is building other products on top of this base layer.

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 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, upgradability 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:

Cash contracts Pre-order Update Contracts

Type	Solidity
Platform	EVM-compatible
Target	cash-contracts
Repository	https://github.com/etherfi-protocol/cash-contracts ↗
Version	`PR #23`
Programs	`PreOrder`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-days. The assessment was conducted by two consultants over the course of three calendar days.

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:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 13, 2024	Start of primary review period
September 15, 2024	End of primary review period

3. Detailed Findings

3.1. Wrong minted tokens recipient

Target	PreOrder		
Category	Coding Mistakes	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

The `mint` and `MintWithPermit` function were modified by the PR to receive a `_buyer` argument that specifies the address that should receive the token being minted.

However, `MintWithPermit` was not modified to make proper use of the argument: the call to `permit` is using `msg.sender` as the source of the fees used to buy the minted token, and the minted token is always sent to `msg.sender`. Despite the tokens are minted to `msg.sender`, the emitted event contains the `_buyer` address.

```
function MintWithPermit(
    uint8 _type,
    uint8 _tier,
    address _buyer,
    uint256 _amount,
    uint256 _deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external whenNotPaused {
    require(
        tiers[_tier].mintCount < tiers[_tier].maxSupply,
        "Tier sold out"
    );

    uint256 tokenId = tiers[_tier].startId + tiers[_tier].mintCount;
    tiers[_tier].mintCount += 1;

    if (_type != uint8(Type.FIAT_ORDER)){
        require(_amount == tiers[_tier].costWei, "Incorrect amount sent");
        IERC20Permit(eEthToken).permit(
            msg.sender,
            address(this),
            _amount,
            _deadline,
            v,
            r,
            s
        );
       IERC20(eEthToken).transferFrom(msg.sender, gnosisSafe, _amount);
        if (_type == uint8(Type.PRE_ORDER)) {
            emit PreOrderMint(_buyer, _tier, _amount, tokenId);
        }
        if (_type == uint8(Type.CRYPTO_ORDER)) {
            emit OrderCryptoMint(_buyer, _tier, _amount, tokenId);
        }
    } else {
        require(msg.sender == fiatMinter, "Not the fiatMinter");
        emit OrderFiatMint(_buyer, _tier, 0, tokenId);
    }

   safeMint(msg.sender, _tier, tokenId);
}
```

Impact

Currently, `MintWithPermit` only works if `msg.sender` corresponds to the minted token recipient. This prevents proper usage of `MintWithPermit`.

If the `MintWithPermit` function is used with the `FIAT_ORDER` flow, the token will be minted to `fiatMinter` (the only address that can use this order type), instead of the intended `_buyer`.

Recommendations

Use the `_buyer` argument instead of `msg.sender` in the call to `permit`, `transferFrom`, and as the recipient of the minted token in `safeMint`.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in PR #27 ↗.

We note that the introduction of the `_buyer` parameter requires careful management of approvals. No user approval for eETH should be given to the contract, outside of the approval granted via `permit` by `MintWithPermit`.

3.2. Lack of type check

Target	PreOrder		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The functions `mint` and `MintWithPermit` supports three different types of orders `FIAT_ORDER`, `PRE_ORDER` and `CRYPTO_ORDER`. However, nothing prevents a user from calling, the two functions with a type value not corresponding to any order type. In the case of `mint`, the logic handling the various order types is as follows:

```
if (_type != uint8(Type.FIAT_ORDER)){
    require(msg.value == tiers[_tier].costWei, "Incorrect amount sent");
    (bool success, ) = gnosisSafe.call{value: msg.value}("");
    require(success, "Transfer failed");

    if (_type == uint8(Type.PRE_ORDER))
        emit PreOrderMint(_buyer, _tier, msg.value, tokenId);

    if (_type == uint8(Type.CRYPTO_ORDER))
        emit OrderCryptoMint(_buyer, _tier, msg.value, tokenId);
} else {
    require(msg.sender == fiatMinter, "Not the fiatMinter");
    emit OrderFiatMint(_buyer, _tier, 0, tokenId);
}

safeMint(_buyer, _tier, tokenId);
```

Thus, for an unknown value, the minting is done but no events are emitted as it should for the defined types.

Impact

This issue could lead to user confusion, and could cause wrong accounting of minted tokens in offchain applications relying on mint events.

Recommendations

The contract should revert if an unknown type is used for minting.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in PR #27 ↗.

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. Discussion of the changes

The pull request modified the `mint` and `MintWithPermit` functions to support three flows for minting tier-tokens: `PRE_ORDER`, `CRYPTO_ORDER`, `FIAT_ORDER`.

The `PRE_ORDER` and `CRYPTO_ORDER` flows can be used to pay for fees directly on-chain, by using either native ETH (via `mint`) or by using eETH (via `MintWithPermit`). The `FIAT_ORDER` flow is only accessible to a privileged configurable address (`fiatMinter`).

The `FIAT_ORDER` flow required the introduction of a new argument which specifies the recipient of the minted token.

4.2. Constructor left uninitialized

The Cash contracts Pre-order Update is upgradable and uses the UUPS Proxy mechanism. Thus, the contract is initialized by the `initialize` function and not the constructor. However, as mentioned in the OpenZepplin documentation ↗, the implementation contract should not be left uninitialized.

For modern versions of UUPSUpgradable this is not a security issue anymore. However, as a best practice a constructor that invokes the `_disableInitializers` function to lock it when it is deployed can be added.

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: PreOrder.sol

Function: **`function MintWithPermit(uint8 _type, uint8 _tier, address _buyer, uint256 _amount, uint256 _deadline, uint8 v, bytes32 r, bytes32 s)`**

Mints a token with eETH as payment.

Inputs

- `_type`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Selects the type of order.
- `_tier`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Selects tier for which a token is to be minted.
- `_buyer`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Address of the token buyer.
- `_amount`
  - **Control**: Completely controlled by caller.
  - **Constraints**: Must be equal to the configured price of the tier.
  - **Impact**: Amount of eETH to transfer.
- `_deadline, v, r, s`
  - **Control**: Completely controlled by caller.
  - **Constraints**: Must be a valid ERC-20 permit signature.
  - **Impact**: ERC-20 permit signature for eETH.

Branches and code coverage (including function calls)

Intended branches

- ☑ Token is minted upon payment.
- ☑ Payment is transferred to GnosisSafe.

Negative behavior

- ☑ Reverts if signature is incorrect.
- ☑ Reverts if incorrect amount is transferred.
- ☑ Reverts if ETH is transferred directly to the contract.
- ☑ Reverts if tier is sold out.

Function: `mint(uint8 _type, uint8 _tier, address _buyer)`

Mints a token with ETH as payment.

Inputs

- `_type`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Selects the type of order.
- `_tier`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Selects tier for which a token is to be minted.
- `_buyer`
  - **Control**: Completely controlled by caller.
  - **Constraints**: None.
  - **Impact**: Address of the token buyer.

Branches and code coverage (including function calls)

Intended branches

- ☑ Token is minted upon ETH payment.
- ☑ Token is minted upon fiat payment with fiat minter as sender.
- ☑ Payment is transferred to GnosisSafe.

Negative behaviour

- ☑ Reverts if incorrect amount is transferred.
- ☑ Reverts if ETH is transferred directly to the contract.
- ☑ Reverts if minting with fiat payment when fiat minter is not the sender.
- ☑ Reverts if tier is sold out.

6. Assessment Results

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

During our assessment on the scoped Cash contracts Pre-order Update contracts, we discovered two findings. No critical issues were found. One finding was of high impact and the other finding was informational in nature.

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.