Zellic

Beefy Wrapper

Smart Contract Security Assessment

August 3, 2023

Prepared for:

Kexley

Beefy Finance

Prepared by:

Filippo Cremonese and Sina Pilehchiha

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 Beefy Finance from August 1st to August 2nd, 2023. During this engagement, Zellic reviewed Beefy Wrapper'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:

- Could an exploiter withdraw more assets than they own via share manipulation?
- Is the implementation of the ERC-4626 vault according to the standard specifications?
- Could a vulnerable logic flow trigger a reentrancy?
- Could incorrect/invalid accounting lead to loss of funds?

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

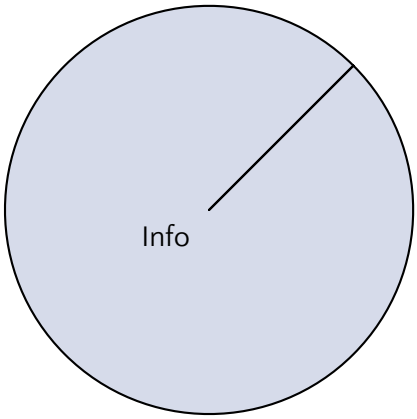
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Beefy Wrapper contracts, we discovered two findings, all of which were informational in nature.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Beefy Wrapper

Beefy Wrapper is a decentralized, multichain yield optimizer that allows its users to earn compound interest on their crypto holdings. Beefy earns users the highest APYs with safety and efficiency in mind.

Through a set of investment strategies secured and enforced by smart contracts, Beefy automatically maximizes the user rewards from various liquidity pools (LPs), automated market maker (AMM) projects, and other yield-farming opportunities in the DeFi ecosystem.

The main product offered by Beefy Finance are the vaults in which a user stakes their crypto tokens. The investment strategy tied to the vault will automatically increase their deposited token amount by compounding arbitrary yield-farm reward tokens back into their initially deposited asset. Despite the name "vault" suggests, a user's funds are never locked in any vault on Beefy; they can always withdraw at any moment in time.

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 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:

Beefy Wrapper Contracts

Repository	https://github.com/beefyfinance/beefy-contracts
Version	beefy-contracts: `0620b5c07946313f6b49adbe4ebca7ccbafef58a`
Programs	BeefyWrapper BeefyWrapperFactory BIFI
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-days. 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:

Filippo Cremonese, Engineer
fcremo@zellic.io

Sina Pilehchiha, Engineer
sina@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 1, 2023	Kick-off call
August 1, 2023	Start of primary review period
August 2, 2023	End of primary review period

3 Detailed Findings

3.1 Function should revert if `assets > balance` state is reached

- **Target**: BeefyWrapper
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

In the provided code snippet at BeefyWrapper.sol, line #142, within the `_withdraw()` function, a valid execution logic would involve reverting the function if the value of `assets` exceeds the `balance`. This condition should ideally never evaluate to true.

```
function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    ) internal virtual override {
        if (caller ≠ owner) {
            _spendAllowance(owner, caller, shares);
        }
        _burn(owner, shares);

        IVault(vault).withdraw(shares);
        uint balance
    = IERC20Upgradeable(asset()).balanceOf(address(this));
        if (assets > balance) {
            assets = balance;
        }

        IERC20Upgradeable(asset()).safeTransfer(receiver, assets);

        emit Withdraw(caller, receiver, owner, assets, shares);
    }
```

Impact

If the condition where `assets` is greater than `balance` is not handled properly and allowed to proceed, it might result in unexpected behavior and cause confusion.

Recommendations

Consider changing the logic of the function to revert in such a scenario.

Remediation

This finding has been acknowledged by Beefy Finance. Their official response is paraphrased below:

> We will sometimes withdraw less from a vault than calculated as the ERC4626 preview calculation doesn't take into account any withdrawal fees, either from our strategy contracts or from underlying contracts. Any underlying contract's fees are not currently fetched dynamically and would require very custom code for each vault to fetch. Even a 0.01% fee on the strategy will cause the revert. We would like to avoid reverts in any withdrawal, which is exactly what the existing logic prevents. We're not swapping in the withdrawal so don't expect slippage, only withdraw fees.

3.2 Risk of ERC-4626 inflation attack

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

Description

Empty ERC-4626 vaults can be manipulated to inflate the price of a share and cause depositors to lose their deposits due to rounding in favor of the vault.

In empty (or nearly empty) ERC-4626 vaults, deposits are at high risk of being stolen through front-running with a donation to the vault that inflates the price of a share. This is variously known as a donation or inflation attack and is essentially a problem of slippage.

Impact

This attack allows malicious actors to steal deposits into pools, which will result in potentially notable losses for users.

Recommendations

For protection against inflation attacks, we suggest upgrading the ERC4626Upgradeable OpenZeppelin contract used in BeefyWrapper.sol to the current version (4.9).

The latest version of the ERC4626Upgradeable OpenZeppelin contract explains their proposed solution to this type of attack:

> The `_decimalsOffset()` corresponds to an offset in the decimal representation between the underlying asset's decimals and the vault decimals. This offset also determines the rate of virtual shares to virtual assets in the vault, which itself determines the initial exchange rate.
>
> While not fully preventing the attack, analysis shows that the default offset (0) makes it non-profitable, as a result of the value being captured by the virtual shares (out of the attacker's donation) matching the attacker's expected gains. With a larger offset, the attack becomes orders of magnitude more expensive than it is profitable.

Remediation

This issue has been fixed by Beefy Finance in commit 39a7e1a.

4 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.

4.1 Module: BeefyWrapperFactory.sol

Function: `clone(address _vault)`

This function can be used to create a new clone of the BeefyWrapper contract using an immutable proxy that `delegatecall`s the wrapper contract code.

Inputs

- `_vault`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the contract to clone.

Branches and code coverage (including function calls)

Intended branches

- Deploys the immutable proxy contract and calls `initialize` on it.
  ☑ Test coverage

Function call analysis

- `rootFunction → IWrapper(proxy).initialize( ... )`
  - **What is controllable?** All arguments; `_vault` is controlled and `name` and `symbol` are obtained by calling `_vault`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts would abort the transaction; reentrancy is possible but not a concern.

4.2 Module: BeefyWrapper.sol

Function: `unwrap(uint256 amount)`

This function can be used to unwrap a given amount of wrapped tokens in exchange for the original Beefy tokens.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None directly (user balance must be sufficient).
  - **Impact**: Amount of tokens to be unwrapped.

Branches and code coverage (including function calls)

Intended branches

- Burns the specified amount of wrapped tokens and transfers the corresponding amount of vault tokens to the caller.
  ☐ Test coverage

Negative behavior

- Reverts if the user balance is insufficient.
  ☐ Negative test
- Reverts if the transfer of unwrapped tokens fails.
  ☐ Negative test

Function call analysis

- `rootFunction → _burn(msg.sender, amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible.
- `rootFunction → IERC20Upgradeable(vault).safeTransfer(msg.sender, amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts bubble up (even though they should not be possible); reentrancy is not possible (`vault` is considered trusted).

Function: `wrap(uint256 amount)`

This function can be used to wrap a given amount of Beefy vault tokens into ERC-4626 wrapped tokens.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None (directly, caller balance must be sufficient and the wrapper must be approved).
  - **Impact**: Amount to wrap.

Branches and code coverage (including function calls)

Intended branches

- Transfers vault tokens to the wrapper contract and mints the corresponding amount of wrapper tokens.
  - ☐ Test coverage

Negative behavior

- Reverts if the vault token transfer fails (e.g., user balance is insufficient).
  - ☐ Negative test

Function call analysis

- `rootFunction → IERC20Upgradeable(vault).safeTransferFrom(msg.sender, address(this), amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`vault` is considered trusted).
- `rootFunction → _mint(msg.sender, amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts and reentrancy are not possible.

Function: `_deposit(address caller, address receiver, uint256 assets, uint256 shares)`

This internal function overrides the default ERC-4626 implementation and is invoked by the public, inherited functions `deposit` and `mint`.

Inputs

- `caller`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Caller performing the deposit.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Receiver of the minted shares.
- `asset`
  - **Control**: Arbitrary (when coming from `deposit`).
  - **Constraints**: None (directly, caller balance must be sufficient).
  - **Impact**: Amount of assets to wrap.
- `shares`
  - **Control**: Arbitrary (when coming from `mint`).
  - **Constraints**: None (directly, corresponding caller asset balance must be sufficient).
  - **Impact**: Intended to be the amount of shares to mint but ignored and recomputed internally.

Branches and code coverage (including function calls)

Intended branches

- Transfers asset from the caller to the wrapper contract, calls the vault to deposit the asset, and mints the corresponding amount of shares to the receiver.
  - ☐ Test coverage

Negative behavior

- Reverts if the asset transfer fails (e.g., caller balance is insufficient).
  - ☐ Negative test
- Reverts if the vault deposit fails.

☐ Negative test

Function call analysis

- `rootFunction → IERC20Upgradeable(asset()).safeTransferFrom(caller, address(this), assets)`
  - **What is controllable?** `assets`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`asset` is considered trusted).
- `rootFunction → IERC20Upgradeable(vault).balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the initial vault tokens' balance.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`vault` is considered trusted).
- `rootFunction → IVault(vault).deposit(assets)`
  - **What is controllable?** `assets`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`vault` is considered trusted).
- `rootFunction → shares = IERC20Upgradeable(vault).balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the final vault balance — the difference between final and initial balance is used as the amount of shares to be minted.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`vault` is considered trusted).
- `rootFunction → _mint(receiver, shares)`
  - **What is controllable?** Nothing directly.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts and reentrancy are not possible.

Function: `_withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)`

This internal function overrides the default ERC-4626 implementation and is invoked by the public, inherited functions `withdraw` and `redeem`.

Inputs

- `caller`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Caller performing the withdrawal.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Receiver of the withdrawal.
- `owner`
  - **Control**: None.
  - **Constraints**: If not the sender, `caller` must have allowance.
  - **Impact**: Owner of the shares to withdraw.
- `assets`
  - **Control**: Arbitrary (when coming from `withdraw`).
  - **Constraints**: None (directly, owner share balance must be sufficient).
  - **Impact**: Amount of assets to unwrap.
- `shares`
  - **Control**: Arbitrary (when coming from `redeem`).
  - **Constraints**: None (directly, owner share balance must be sufficient).
  - **Impact**: Amount of shares to unwrap.

Branches and code coverage (including function calls)

Intended branches

- Spends allowance if caller is not owner.
  - ☐ Test coverage
- Burns owner shares, withdraws shares from the vault, transfers `min(assets, balance)` to the receiver.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have sufficient allowance.

  - ☐ Negative test
- Reverts if owner balance is insufficient.
  - ☐ Negative test
- Reverts if vault withdrawal fails.
  - ☐ Negative test
- Reverts if asset transfer fails (should be impossible).
  - ☐ Negative test

Function call analysis

- `rootFunction → _spendAllowance(owner, caller, shares)`
  - **What is controllable?** `owner` and `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible.
- `rootFunction → _burn(owner, shares)`
  - **What is controllable?** `owner` and `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible.
- `rootFunction → IVault(vault).withdraw(shares)`
  - **What is controllable?** `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`vault` is considered trusted).
- `rootFunction → IERC20Upgradeable(asset()).balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Used to limit the maximum withdrawal.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is not possible (`asset` is considered trusted).
- `rootFunction → IERC20Upgradeable(asset()).safeTransfer(receiver, assets)`
  - **What is controllable?** `receiver` and `assets`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts bubble up; reentrancy is not possible (`asset` is considered trusted).

5 Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet.

During our assessment on the scoped Beefy Wrapper contracts, we discovered two findings, all of which were informational in nature. Beefy Finance acknowledged all findings and implemented fixes.

5.1 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.