Zellic

SpiceFi Vaults

Smart Contract Security Assessment

December 14, 2022

Prepared for:

Niyant Narang

Spice Finance Inc.

Prepared by:

Ayaz Mammadov and Filippo Cremonese

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 an audit for Spice Finance Inc. from November 11, 2022 to November 22, 2022.

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 good. The code is easy to comprehend, and in most cases, intuitive.

We applaud Spice Finance Inc. for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of SpiceFi Vaults, specifically on the implementation of the ERC4626 vaults, which are incredibly compliant with the specification.

Zellic thoroughly reviewed the SpiceFi Vaults codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

Specifically, taking into account SpiceFi Vaults's threat model, we focused heavily on issues that would break core invariants, especially those that would allow unauthorized access to funds within vaults, such as

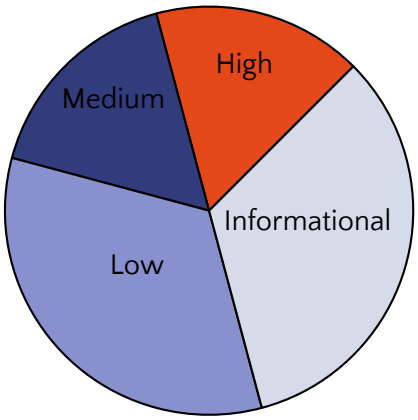
- The manipulation of vault funds through the manipulation of the underlying pools/assets.
- The access control of the vaults to ensure that strategists are not able to route vault funds to unapproved destinations/sources.
- DoS that would result in frozen funds or result in decreased withdrawals.

During our assessment on the scoped SpiceFi Vaults contracts, we discovered six findings. Fortunately, no critical issues were found. Of the six findings, one was of high impact, one was of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Spice Finance Inc.'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	1
Low	2
Informational	2



2 Introduction

2.1 About SpiceFi Vaults

SpiceFi Vaults aggregates NFT lending liquidity. To do so, users set up aggregator vaults (or use those set up by Spice) that route capital to NFT loans across various marketplaces, both peer2peer and peer2pool. Peer2pool marketplaces have ERC4626 wrappers built on top of them for a standardized interface for Spice vaults. Peer2peer marketplaces are largely off-chain, and ERC4626 vaults are built for them with an open source off-chain bidder deciding how bids should be placed using the funds from the vault. Open-source off-chain strategists determine how to rebalance the portfolio of aggregator vaults on behalf of users.

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 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.

2.3 Scope

The engagement involved a review of the following targets:

SpiceFi Vaults Contracts

Repository	https://github.com/teamspice/vault
Versions	`1eed022eaf835ad9704775821593fecf647ed5d8`
Contracts	• SpiceFi4626 • Bend4626 • Drops4626 • SpiceFiFactory • Vault
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 1.5 person-weeks. The assessment was conducted over the course of two calendar weeks.

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:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Filippo Cremonese, Engineer
fcremo@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 11, 2022	Start of primary review period
November 22, 2022	End of primary review period
December 6, 2022	Closing call

3 Detailed Findings

3.1 `totalAssets` in SpiceFi4626 does not account for assets in VAULT_RECEIVER

- **Target**: `SpiceFi4626`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The SpiceFi4626 vault contains functionality that allows strategists to invest capital on behalf of the vault. Strategists are allowed to withdraw invested capital to addresses marked `VAULT_RECEIVER_ROLE`.

An issue arises when calculating `totalAssets`. The vault only measures the assets it has in other vaults that belong to the vault itself (`address(this)`); it does not measure the assets that were withdrawn to `VAULT_RECEIVER` addresses.

```
function totalAssets() public view override returns (uint256) {
    ...
        balance += vault.previewRedeem(vault.balanceOf(address(this)));
    ...
}
```

Impact

Users who withdraw while assets are in `VAULT_RECEIVER` addresses will be returned less asset than they invested. Users who invested into the vault while assets are in `VAULT_RECEIVER` addresses will end up massively rewarded if those vault assets are returned to the vault, making money by taking the capital of other users.

Recommendations

Account for capital/assets in `VAULT_RECEIVER` addresses.

Remediation

This was remediated in commit a0bcff20, the strategist functions only deposit/withdraw to the vault, resulting in the AUM calculation working as intended.

3.2 Slippage/manipulated exchange rates when depositing

- **Target**: `Drops4626`
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: High
- **Impact**: Medium

Description

Certain vaults contain logic to exchange deposited assets (e.g., WETH) to the vault asset (CEther).

The amount of CEther received by the `mint` called in `deposit` is determined by the current exchange rate that can be manipulated by minting and redeeming.

A MEV user could use these techniques to sandwich a large deposit and extract/steal the deposit of a vault user by following the actions below:

- The MEV user mints a lot of cETH.
- The large deposit goes through, but the vault user receives few cETH due to the bad exchange rate.
- The MEV user redeems the cETH getting back more ETH than they started with, essentially eating into the deposit of the vault user.

Impact

The deposits/withdrawals of vault users are at risk of being stolen.

Recommendations

Add deposit call interfaces that allow users to specifiy minimum exchange rates.

Remediation

Spice Finance Inc. acknowledged and addressed the issue in commit 0d49a0b2 by implementing a slippage limited interface in the `SpiceFi4626` contract through which users are supposed to interact with directly. We note that slippage protection is not implemented in the underlying `Bend4626` and `Drops4626` contracts, which are still potentially vulnerable if used directly; our understanding is that those contracts are not intended to be called directly.

3.3 Potentially uninitialized implementation contracts

- **Target**: `Bend4626`, `Drops4626`, `SpiceFi4626`, `Vault`
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Informational
- **Impact**: Medium

Description

Implementation contracts designed to be called by a proxy should always be initialized to prevent potential takeovers.

Impact

If an implementation contract is not initialized, an attacker could be able to initialize it and perform a `selfdestruct`, deleting the implementation contract and causing a denial of service.

Recommendations

Ensure the implementation contract is always initialized.

Following official OpenZeppelin documentation, this can be accomplished by defining a constructor on the contract:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

For more information, refer to these openZeppelin documents.

Remediation

This was remediated in commit 5dfead1b by adding `_disableInitializers`

3.4 MaxWithdraw does not account for fees

- **Target**: `SpiceFi4626`
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

In vault `Spice4626`, the check for maximum withdrawals can pass but the call to `_withdraw` can still fail because fees are not accounted for.

```
function maxWithdraw(address owner)
    public view override returns (uint256) {
     ...
    return paused() ? 0 : _convertToAssets(
                balanceOf(owner),
                MathUpgradeable.Rounding.Down
            ).min(balance);
}
```

The code above returns the maximum between the owner's balance and the liquid capital of the vault. In the case where a user specifies a withdrawal equal to the available vault balance, this check passes; however, later in `_withdraw`, all of the available capital is used in the call to `super.withdraw`, but then fee transfers are done, which would revert due to the lack of capital in the vault.

```
function _withdraw( ... ) internal override {
    address feesAddr1 = getRoleMember(ASSET_RECEIVER_ROLE, 0);
    address feesAddr2 = getRoleMember(SPICE_ROLE, 0);
    uint256 fees = _convertToAssets(shares,
    MathUpgradeable.Rounding.Down) -
        assets;
    uint256 fees1 = fees.div(2);

    // Uses up entire available capital
    super._withdraw(caller, receiver, owner, assets, shares);


    // These calls will fail due to lack of capital.
    SafeERC20Upgradeable.safeTransfer(
        IERC20MetadataUpgradeable(asset()),
```

```
        feesAddr1,
        fees1
    );
    SafeERC20Upgradeable.safeTransfer(
        IERC20MetadataUpgradeable(asset()),
        feesAddr2,
        fees.sub(fees1)
        );
}
```

Impact

In the edge case of a user having a balance corresponding to an amount higher than the capital available in the vault and would like to withdraw close to the maximum possible withdrawal, the withdrawal will revert with an incorrect message. Other smart contracts building on top of SpiceFi will receive incorrect quantities for `maxWithdraws` resulting in reverts.

Recommendations

Account for the fees in `maxWithdraw`.

Remediation

This was remediated in commit 37e0d2db by accounting for fees.

3.5 Potential rounding error

- **Target**: `SpiceFi4626`
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `SpiceFi4626::maxDeposit` function computes the maximum amount of assets a user should be allowed to deposit, starting from the maximum amount of shares they are allowed to receive in order to not go above the maximum supply. The conversion between shares and assets is performed by rounding up, potentially leading to a slightly higher-than-expected limit.

```
function maxDeposit(address) public view override returns (uint256) {
  return
    paused()
    ? 0
    : _convertToAssets(
      maxTotalSupply - totalSupply(),
      MathUpgradeable.Rounding.Up
      );
}
```

Impact

It might be possible to deposit slightly more assets than intended into the contract.

Recommendations

Round down the conversion from shares to assets.

Remediation

This was remediated in commit bced4a44 by rounding down.

3.6 Fallback function unnecessarily payable by anyone

- **Target**: `Drops4626`
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The fallback function of `Drops4626` is payable and accepts payments from any source.

Impact

It is possible to send Ether to `Drops4626` by mistake.

Recommendations

The fallback function is needed to allow cETH withdrawals, so it needs to be marked payable. The fallback function could revert if `msg.sender` is not cETH.

Remediation

Spice Finance Inc. acknowledged the issue, and decided not to apply a remediation since this issue does not represent a risk of loss of funds for the project.

4 Discussion

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

4.1 Unchecked code

The functions `SpiceFi4626::totalAssets` and `SpiceFiFactory::createVault` contain unchecked code blocks incrementing a `uint8` loop iteration variable. It is technically possible for these variables to overflow, but we believe these overflows to be harmless, causing an infinite loop and reverting due to gas exhaustion.

4.2 Reentrancy guards

All contracts lack reentrancy guards and contain multiple instances where the checks-effects-interactions pattern is not followed. We do not believe these potential issues to be exploitable since we assume the external contracts being invoked do not offer functionality that allows to reenter into the SpiceFi contracts. Due to the lack of reentrancy mitigations, the SpiceFi team needs to continuously evaluate every external call for potential security issues, especially when the external contract being called can be upgraded. We recommend considering to add reentrancy guards to all public and external functions.

4.3 Function `exchangeRateStored`

The usage of `exchangeRateStored`, while not dangerous, can result in a tiny variance in the exchange rate that is retrieved; `exchangeRateCurrent` is the function that returns the real current exchange rate. However, this does not pose a security risk as it is recommended to call `exchangeRateStored` when dealing with non-transactional compound calls.

5 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.

5.1 File: Drops4626

ERC4626 adapter for CEther.

Function: `deposit`

Intended behavior

Deposits an amount of WETH into the pool giving the depositor the corresponding amount of shares.

Branches and code coverage

Intended branches

Invokes `_deposit` with the amount of WETH to deposit.

Tests cover the intended functionality.

Negative behavior

- `assets` is zero
  - ☑ Negative test
- user has not granted approval
  - ☑ Negative test
- user WETH balance is insufficient
  - ☑ Negative test
- receiver is zero
  - ☐ Negative test

Preconditions

- Caller has granted approval for the amount to deposit
- Caller has enough balance

Inputs

- `assets`: amount of WETH to deposit
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller

Function call analysis

None directly, see `_deposit`.

Function: `mint`

Intended behavior

Mints an amount of shares by depositing the corresponding amount of WETH into the pool.

Branches and code coverage

Intended branches

- computes the amount of assets required to mint the desired amount of shares
- invokes `_deposit` with the amount of WETH to deposit and the shares

Tests cover the intented functionality.

Negative behavior

- `shares` is zero
  ☑ Negative test

- user has not granted approval
  - ☑ Negative test
- user WETH balance is insufficient
  - ☑ Negative test
- receiver is zero
  - ☐ Negative test

Preconditions

- Caller has granted approval for the amount to deposit
- Caller has enough balance

Inputs

- `shares`: amount of shares to mint
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller

Function call analysis

None directly, see `_deposit`.

Function: `withdraw`

Intended behavior

Withdraws the given amount of WETH in exchange for the corresponding amount of shares.

Branches and code coverage

Intended branches

- computes amount of shares to burn
- invokes `_withdraw` with the requested amount of WETH and the calculated amount of shares

Tests cover the intended functionality.

Negative behavior

- `assets` is zero
  ☑ Negative test
- user was not granted approval for the shares
  ☑ Negative test
- shares balance is insufficient
  ☑ Negative test
- receiver is zero
  ☑ Negative test

Preconditions

- caller has been granted approval for the amount of shares needed
- owner has enough shares

Inputs

- `assets`: amount of assets to withdraw
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, appropriate approval has to be granted
- `receiver`: receiver of the assets
  - **Control**: can't be zero
  - **Authorization**: N/A
  - **Impact**:
- `owner`: owner of the shares to be spent
  - **Control**: None
  - **Authorization**: must have granted approval to msg.sender
  - **Impact**:

Function call analysis

None directly, see `_withdraw`.

Function: `redeem`

Intended behavior

Redeems the given amount of shares for the corresponding amount of WETH.

Branches and code coverage

Intended branches

Invokes `_withdraw` with the requested amount of shares to redeem.

Tests cover the intented functionality.

Negative behavior

- `shares` is zero
  ☑ Negative test
- user was not granted approval for the shares
  ☑ Negative test
- shares balance is insufficient
  ☑ Negative test
- receiver is zero
  ☑ Negative test

Preconditions

- caller has been granted approval for the amount of shares to redeem
- owner has enough shares

Inputs

- `shares`: amount of shares to redeem
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, appropriate approval has to be granted

- `receiver`: receiver of the assets
  - **Control**: can't be zero
  - **Authorization**: N/A
  - **Impact**:
- `owner`: owner of the shares to be spent
  - **Control**: None
  - **Authorization**: must have granted approval to msg.sender
  - **Impact**:

Function call analysis

None directly, see `_withdraw`.

Function: `_deposit`

Intended behavior

Internal function handling depositing and minting.

Branches and code coverage

Intended branches

- transfers WETH from user to contract
- unwraps WETH
- mints cETH using unwrapped ETH
- mints as many shares to `receiver` as many cETH was minted

The function is tested by tests for `deposit` and `mint`.

Negative behavior

The function is tested by tests for `deposit` and `mint`.

Preconditions

- `msg.sender` has granted approval for the amount to deposit
- `msg.sender` has enough WETH balance

Inputs

- `assets`: amount of WETH to deposit
  - **Control**: None directly
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller

Function call analysis

- `weth.transferFrom(msg.sender, address(this), assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable, nor checked. Technically a violation of ERC20 interface, but WETH reverts on failure.
  - **What happens if it reverts, reenters, or does other unusual control flow?** WETH does not allow to reenter
- `weth.withdraw(assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** WETH does not allow to reenter
- `uint256 beforeBalance = IERC20Upgradeable(lpTokenAddress).balanceOf(address(this));`
  - **What is controllable?** None
  - **If return value controllable, how is it used and how can it go wrong?** Represents the token balance of the contract, can be increased by sending more to it, not a threat
  - **What happens if it reverts, reenters, or does other unusual control flow?** `lpTokenAddress` is trusted not to allow reentrancy
- `cEther.mint{value: assets}();`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable nor used
  - **What happens if it reverts, reenters, or does other unusual control flow?**

`cETH` is trusted not to allow reentrancy

- `shares = IERC20Upgradeable(lpTokenAddress).balanceOf(address(this)) - beforeBalance;`
  - **What is controllable?** None
  - **If return value controllable, how is it used and how can it go wrong?** Represents the token balance of the contract, can be increased by sending more to it, not a threat
  - **What happens if it reverts, reenters, or does other unusual control flow?** `lpTokenAddress` is trusted not to allow reentrancy

Function: `_withdraw`

Intended behavior

Internal function handling withdraws and redeems.

Branches and code coverage

Intended branches

- spends allowance if the caller is not the owner of the shares
- burns the shares
- redeems cETH for ETH
- wraps redeemed ETH in WETH
- transfers WETH to the receiver

The function is tested by tests for `withdraw` and `redeem`.

Negative behavior

The function is tested by tests for `withdraw` and `redeem`.

Preconditions

- `msg.sender` was granted approval for the amount of shares to redeem
- `owner` has enough shares

Inputs

- `caller`: caller performing the withdrawal

  - **Control**: None
  - **Authorization**: Must have approval from the owner
  - **Impact**: None
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `owner`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: owner must have approved `msg.sender`
  - **Impact**: None, since there's authorization
- `shares`: amount of shares to burn
  - **Control**: None directly
  - **Authorization**: N/A
  - **Impact**: None, shares are taken from the owner who must have approved `msg.sender`

Function call analysis

- `cEther.redeem(shares)`
  - **What is controllable?** `shares`
  - **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?** The called contract is trusted not to allow reentrancy
- `weth.deposit{value: assets}()`
  - **What is controllable?** `assets`, `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** The called contract is trusted not to allow reentrancy
- `weth.transfer(receiver, assets)`
  - **What is controllable?** `assets`, `receiver`
  - **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?** The called contract is trusted not to allow reentrancy

5.2 File: SpiceFi4626

Function: `_authorizeUpgrade`

Intended behavior

`UUPSUpgradeable` standard function to authorize implementation upgrades.

Branches and code coverage

Intended branches

Reverts if and only if the upgrade is unauthorized, otherwise returns successfully without reverting.

No tests target this function directly.

Negative behavior

- `msg.sender` does not have the right role
  - ☐ Negative test

Preconditions

- `msg.sender` has the `DEFAULT_ADMIN_ROLE` role

Inputs

- `newImplementation`: address of the new implementation contract. Unused.
  - **Control**: None
  - **Authorization**: None
  - **Impact**: This address will become the new implementation address

Function call analysis

None.

Function: `setWithdrawalFees`

Sets the withdrawal fee rate in base points.

Branches and code coverage

Intended branches

- `withdrawalFees` parameter is set to the value given as argument
  ☑ Test coverage

Negative behavior

- `msg.sender` does not have `DEFAULT_ADMIN_ROLE` role
  ☑ Test coverage
- `withdrawalFees_` > 10000
  ☑ Test coverage

Preconditions

- `msg.sender` has `DEFAULT_ADMIN_ROLE` role

Inputs

- `withdrawalFees_`: withdrawal fees in base points
  - **Control**: Checked to be <= 10000
  - **Authorization**: None
  - **Impact**: Withdrawal fees are set to this value

Function call analysis

None.

Function: `setMaxTotalSupply`

Sets the maximum total supply variable in storage.

Branches and code coverage

Intended branches

- `maxTotalSupply` parameter is set to the value given as argument
  ☑ Test coverage

Negative behavior

- `msg.sender` does not have `DEFAULT_ADMIN_ROLE` role
  ☑ Test coverage
- `maxTotalSupply_ < maxTotalSupply`: not handled!
  ☐ Test coverage

Preconditions

- `msg.sender` has `DEFAULT_ADMIN_ROLE` role

Inputs

- `maxTotalSupply_`: max total supply
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Maximum total supply is set to this value

Function call analysis

None.

Function: `setVerified`

Sets the verified flag of the contract.

Branches and code coverage

Intended branches

- `verified` parameter is set to the value given as argument
  ☐ Test coverage: only checks that the flag can be set to true, since the parameter is boolean test coverage could easily be expanded

Negative behavior

- `msg.sender` does not have `DEFAULT_ADMIN_ROLE` role
  ☑ Test coverage

Preconditions

- `msg.sender` has `DEFAULT_ADMIN_ROLE` role

Inputs

- `verified_`: boolean value
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Verified flag is set to this value

Function call analysis

None.

Function: `_mint`

Internal function handling minting.

Branches and code coverage

Intended branches

- Invokes `super.mint()` to mint the required amount

Negative behavior

- Reverts if maxTotalSupply is too low
  - ☐ Test coverage

Preconditions

None. It's an internal function, it assumes the caller is trusted.

Inputs

- `account`: receiver of the shares
  - **Control**: None
  - **Authorization**: None

  - **Impact**: Receives the minted shares
- `amount`: amount of shares
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Amount of shares to be minted

Function call analysis

No external calls.

Function: `totalAssets`

Computes the total AUM managed by the contract.

Branches and code coverage

Intended branches

- gets the amount directly owned by the contract
- sums the amount of assets in the vaults and owned by the contract

Negative behavior

- there's more than 255 vaults and `i` overflows
  ☐ Test coverage
- a `balanceOf` or `previewRedeem` call fails
  ☐ Test coverage

Preconditions

None

Inputs

None

Function call analysis

- `IERC20MetadataUpgradeable(asset()).balanceOf(address(this))`

  - **What is controllable?** None
  - **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?** Revert would bubble up, not an issue.
- `vault.previewRedeem(vault.balanceOf(address(this)))`
  - **What is controllable?** None
  - **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?** Revert would bubble up, not handled but hardly an issue, and a recoverable one (would require contract update).

Function: `_withdraw`

Internal function handling withdrawals.

Branches and code coverage

Intended branches

- gets addresses of fee receivers (first addresses with `ASSET_RECEIVER_ROLE` and `SPICE_ROLE`)
- computes fees as the difference between the assets corresponding to the shares being burnt and the assets being withdrawn
  ☑ Test coverage

Negative behavior

The function is not directly handling any failure case. It is an internal function, the assumption is that the caller has already performed whatever validation is required, and that the called functions will perform further internal validation and revert in case there's a problem.

Preconditions

- only one address has role `ASSET_RECEIVER_ROLE`
- only one address has role `SPICE_ROLE`

Inputs

- `caller`: whoever has called the public function that caused this call to `_withdraw`
  - **Control**: None
  - **Authorization**: None (indirectly, must have allowance)
  - **Impact**: The caller's share allowance is spent
- `receiver`: receiver of the withdrawn assets
  - **Control**: None
  - **Authorization**: None
  - **Impact**: This address receives the withdrawn assets
- `owner`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: the owner of the shares to be burnt
- `assets`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: amount of assets to withdraw (plus fees)
- `shares`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: amount of shares to be burnt

Function call analysis

- `SafeERC20Upgradeable.safeTransfer(IERC20MetadataUpgradeable(asset()), feesAddr1, fees1)`
  - **What is controllable?** Nothing arbitrarily
  - **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 are bubbled up, asset is trusted not to allow reentrancy
- `SafeERC20Upgradeable.safeTransfer(IERC20MetadataUpgradeable(asset()), feesAddr2, fees2)`
  - **What is controllable?** Nothing arbitrarily
  - **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 are bubbled up, asset is trusted not to allow reentrancy

Function: `_deposit`

Internal function handling deposits.

Branches and code coverage

Intended branches

- checks authorization, then calls `super._deposit()`
  ☑ Test coverage: tested indirectly by tests for the public `deposit` function

Negative behavior

- there are no users with role `USER_ROLE`
  ☑ Test coverage
- `caller` does not have `USER_ROLE`
  ☑ Test coverage
- `shares` is zero
  ☐ Test coverage
- `caller` does not have enough assets
  ☑ Test coverage

Preconditions

- there must be at least one account with `USER_ROLE`
- `caller` must have `USER_ROLE`
- `caller` has enough assets
- `caller` has granted approval for the assets to SpiceFi

Inputs

- `caller`: whoever has called the function calling `_deposit`
  - **Control**: None (internal function, it's supposed to be `msg.sender` normally)
  - **Authorization**: Must have allowance
  - **Impact**: The allowance of this address is spent

Function call analysis

No direct external call.

Function: `transfer`

Can be used by strategists to move vault tokens.

Branches and code coverage

Intended branches

- after performing authorization checks invokes transfer on the underlying vault
  ☑ Test coverage

Negative behavior

- contract is paused
  ☑ Test coverage
- `caller` is not a strategist
  ☑ Test coverage
- `vault` does not have vault role
  ☑ Test coverage
- `to` does not have vault receiver role
  ☑ Test coverage
- `amount` is too high
  ☑ Test coverage

Preconditions

- `caller` has strategist role
- `vault` has vault role
- `to` has vault receiver role

Inputs

- `vault`:
  - **Control**: must be an address with vault role
  - **Authorization**: must be an address with vault role
  - **Impact**: vault tokens are transferred from this vault
- `to`:
  - **Control**: must be an address with vault receiver role
  - **Authorization**: must be an address with vault receiver role
  - **Impact**: vault tokens are transferred to this address

- `amount`:
  - **Control**: arbitrary
  - **Authorization**: NA
  - **Impact**: this amount of vault tokens is transferred

Function call analysis

- `IERC4626Upgradeable(vault).transfer(to, amount)`
  - **What is controllable?** `to`, `amount`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert in the callee is bubbled up, the vault is trusted not to allow reentrancy.

Function: `transferFrom`

Can be used by strategists to move vault tokens from other addresses (with approval).

Branches and code coverage

Intended branches

- after performing authorization checks invokes transfer on the underlying vault
  ☑ Test coverage

Negative behavior

- contract is paused
  ☑ Test coverage
- `caller` is not a strategist
  ☑ Test coverage
- `vault` does not have vault role
  ☑ Test coverage
- `to` does not have vault receiver role
  ☑ Test coverage
- `amount` is too high
  ☑ Test coverage
- `caller` does not have approval
  ☑ Test coverage

Preconditions

- `caller` has strategist role
- `caller` has approval from `from`
- `vault` has vault role
- `to` has vault receiver role

Inputs

- `vault`:
  - **Control**: must be an address with vault role
  - **Authorization**: must be an address with vault role
  - **Impact**: vault tokens are transferred from this vault
- `from`:
  - **Control**: arbitrary
  - **Authorization**: must have granted authorization to `to`
  - **Impact**: vault tokens are transferred from this address balance
- `to`:
  - **Control**: must be an address with vault receiver role
  - **Authorization**: must be an address with vault receiver role
  - **Impact**: vault tokens are transferred to this address
- `amount`:
  - **Control**: arbitrary
  - **Authorization**: NA
  - **Impact**: this amount of vault tokens is transferred

Function call analysis

- `IERC4626Upgradeable(vault).transferFrom(from, to, amount)`
  - **What is controllable?** `from`, `to`, `amount`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert in the callee is bubbled up, the vault is trusted not to allow reentrancy.

Function: `approve`

Can be used by strategists to grant approval for vault tokens.

Branches and code coverage

Intended branches

- After having performed checks on the caller and inputs, invokes approve on the vault
  ☑ Test coverage

Negative behavior

- caller is not a strategist
  ☑ Test coverage
- `vault` does not have vault role
  ☑ Test coverage
- `spender` does not have vault receiver role
  ☑ Test coverage

Preconditions

- `caller`, `vault` and `spender` must be authorized

Inputs

- `vault`:
  - **Control**: None
  - **Authorization**: Must have vault role
  - **Impact**: It's the address of the vault on which `approve` is invoked
- `spender`:
  - **Control**: None
  - **Authorization**: Must have vault receiver role
  - **Impact**: It's the address granted approval
- `amount`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: It's the amount being approved

Function call analysis

- `IERC4626Upgradeable(vault).approve(spender, amount)`
  - **What is controllable?** `spender`, `amount`

- **If return value controllable, how is it used and how can it go wrong?** NA
- **What happens if it reverts, reenters, or does other unusual control flow?** A revert would bubble up, the vault is trusted not to allow for reentrancy

Function: `deposit`

Can be used by strategists to deposit assets from the contract to a vault.

Branches and code coverage

Intended branches

- performs authorization checks on the caller and arguments
- increases allowance
- calls `deposit` on the vault
  ☑ Test coverage

Negative behavior

- `caller` does not have `STRATEGIST_ROLE`
  ☑ Test coverage
- `vault` does not have `VAULT_ROLE`
  ☑ Test coverage
- `receiver` does not have `VAULT_RECEIVER_ROLE`
  ☑ Test coverage
- contract does not have enough assets
  ☑ Test coverage

Preconditions

- `caller`, `vault`, `receiver` have the correct roles
- contract has enough assets

Inputs

- `vault`:
  - **Control**:
  - **Authorization**: Must have `VAULT_ROLE`
  - **Impact**: It's the address of the vault to be called
- `assets`:

  - **Control**:
  - **Authorization**: NA
  - **Impact**: Amount of assets to be deposited
- `receiver`:
  - **Control**:
  - **Authorization**: Must have `VAULT_RECEIVER_ROLE`
  - **Impact**: Receiver of the deposited assets

Function call analysis

- `IERC4626Upgradeable(vault).deposit(assets, receiver)`
  - **What is controllable?** `assets`, `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert in the callee bubbles up. The vault is trusted not to revert.

Function: `mint`

Can be used by strategists to mint shares from an underlying vault.

Branches and code coverage

Intended branches

- performs authorization checks on the caller and arguments
- computes the amount of assets to deposit
- increases allowance
- calls `mint` on the vault.
  ☑ Test coverage

Negative behavior

- caller is not a strategist
  ☑ Test coverage
- `vault` does not have `VAULT_ROLE`
  ☑ Test coverage
- `receiver` does not have `VAULT_RECEIVER_ROLE`
  ☑ Test coverage
- contract does not have enough assets
  ☑ Test coverage

Preconditions

- `caller`, `vault`, `receiver` have the correct roles
- contract has enough assets

Inputs

- `vault`:
  - **Control**:
  - **Authorization**: Must have `VAULT_ROLE`
  - **Impact**: It's the address of the vault to be called
- `shares`:
  - **Control**:
  - **Authorization**: NA
  - **Impact**: Amount of shares to be minted
- `receiver`:
  - **Control**:
  - **Authorization**: Must have `VAULT_RECEIVER_ROLE`
  - **Impact**: Receiver of the deposited assets

Function call analysis

- `SafeERC20Upgradeable.safeIncreaseAllowance(IERC20MetadataUpgradeable(asset()), vault, assets_)`
  - **What is controllable?** `vault`, `assets_`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would bubble up. The asset is trusted not to allow for reentrancy.
- `IERC4626Upgradeable(vault).mint(shares, receiver)`
  - **What is controllable?** `shares`, `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would bubble up. The vault is trusted not to allow for reentrancy.

Function: `withdraw`

Can be used by strategists to withdraw assets from an underlying vault.

Branches and code coverage

Intended branches

- performs authorization checks on the caller and arguments
- calls `withdraw` on the vault.
  - ☑ Test coverage

Negative behavior

- caller is not a strategist
  - ☑ Test coverage
- `vault` does not have `VAULT_ROLE`
  - ☑ Test coverage
- `receiver` does not have `VAULT_RECEIVER_ROLE`
  - ☑ Test coverage
- contract does not have enough assets in the vault
  - ☐ Test coverage

Preconditions

- `caller`, `vault`, `receiver` have the correct roles
- contract has enough assets in the vault

Inputs

- `vault`:
  - **Control**:
  - **Authorization**: Must have `VAULT_ROLE`
  - **Impact**: It's the address of the vault to be called
- `assets`:
  - **Control**:
  - **Authorization**: NA
  - **Impact**: Amount of assets to be withdrawn
- `receiver`:
  - **Control**:
  - **Authorization**: Must have `VAULT_RECEIVER_ROLE`
  - **Impact**: Receiver of the withdrawn assets
- `owner`:
  - **Control**: NA

  - **Authorization**: NA
  - **Impact**: Owner of the shares that are burned to withdraw the assets

Function call analysis

- `IERC4626Upgradeable(vault).withdraw(assets, receiver, owner)`
  - **What is controllable?** `assets`, `receiver`, `owner`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would bubble up. The vault is trusted not to allow for reentrancy.

Function: `redeem`

Can be used by strategists to redeem vault shares in exchange for assets from an underlying vault.

Branches and code coverage

Intended branches

- performs authorization checks on the caller and arguments
- calls `redeem` on the vault.
  ☑ Test coverage

Negative behavior

- caller is not a strategist
  ☑ Test coverage
- `vault` does not have `VAULT_ROLE`
  ☑ Test coverage
- `receiver` does not have `VAULT_RECEIVER_ROLE`
  ☑ Test coverage
- contract does not have enough assets in the vault
  ☐ Test coverage

Preconditions

- `caller`, `vault`, `receiver` have the correct roles
- contract has enough assets in the vault

Inputs

- `vault`:
  - **Control**:
  - **Authorization**: Must have `VAULT_ROLE`
  - **Impact**: It's the address of the vault to be called
- `shares`:
  - **Control**:
  - **Authorization**: NA
  - **Impact**: Amount of shares to be redeemed
- `receiver`:
  - **Control**:
  - **Authorization**: Must have `VAULT_RECEIVER_ROLE`
  - **Impact**: Receiver of the withdrawn assets
- `owner`:
  - **Control**: NA
  - **Authorization**: NA
  - **Impact**: Owner of the shares that are burned to withdraw the assets

Function call analysis

- `IERC4626Upgradeable(vault).redeem(shares, receiver, owner)`
  - **What is controllable?** `shares`, `receiver`, `owner`
  - **If return value controllable, how is it used and how can it go wrong?** NA
  - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would bubble up. The vault is trusted not to allow for reentrancy.

5.3 File: SpiceFiFactory.sol

Function: `Constructor`

Intended behavior

verifies implementation gives msg.sender DEFAULT_ADMIN_ROLE

Precondition:

None

Inputs:

- `Implementation`
  - **Control**: Full Control
  - **Authorization**: called by constructor
  - **Impact**: determines implementation

Function call analysis

`_setupRole`:

- **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: ok
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `createVault`

Intended behavior

takes vault params such as (underlying asset, the vault receiver, the investment-vaults and withdrawalfees) and creates a vault

Precondition:

None

Inputs:

- asset
  - **Control**: Full Control
  - **Authorization**: != 0
  - **Impact**: the asset of the vault
- assetReceiver:
  - **Control**: Full Control
  - **Authorization**: != 0
  - **Impact**: the asset of the vault
- vaults:

  - **Control**: Full Control
  - **Authorization**: None
  - **Impact**: possible destination investment vaults.
- withdrawalFees:
  - **Control**: Full Control
  - **Authorization**: None here
  - **Impact**: determines the withdrawal fees.

Function call analysis

- clone:
  - **What is controllable?**: no control (implementation is immutable)
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: determines vault address
- initialize:
  - **What is controllable?**: Control of constructor arguments
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded
- _checkRole:
  - **What is controllable?**: the vaults
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded
- grantRole:
  - **What is controllable?**: the vaults you pass in
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: None

5.4 File: Bend4626

Function: `deposit`

Intended behavior

Deposits an amount of WETH into the pool giving the depositor the corresponding amount of shares.

Branches and code coverage

Intended branches

- computes the amount of shares to mint
- invokes `_deposit` with the amount of WETH to deposit and the corresponding shares

Tests cover the intended functionality.

Negative behavior

- `assets` is zero
  - ☑ Negative test
- user has not granted approval
  - ☑ Negative test
- user WETH balance is insufficient
  - ☑ Negative test
- receiver is zero
  - ☐ Negative test

Preconditions

- Caller has granted approval for the amount to deposit
- Caller has enough balance

Inputs

- `assets`: amount of assets to deposit
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None

- **Authorization**: N/A
- **Impact**: None, funds are taken from the caller

Function call analysis

None directly, see `_deposit`.

Function: `mint`

Intended behavior

Mints an amount of shares by depositing the corresponding amount of WETH into the pool.

Branches and code coverage

Intended branches

- computes the amount of assets required to mint the desired amount of shares
- invokes `_deposit` with the amount of WETH to deposit and the shares

Tests cover the intended functionality.

Negative behavior

- `shares` is zero
  ☑ Negative test
- user has not granted approval
  ☑ Negative test
- user WETH balance is insufficient
  ☑ Negative test
- receiver is zero
  ☐ Negative test

Preconditions

- Caller has granted approval for the amount to deposit
- Caller has enough balance

Inputs

- `shares`: amount of shares to mint
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller

Function call analysis

None directly, see `_deposit`.

Function: `withdraw`

Intended behavior

Withdraws the given amount of WETH in exchange for the corresponding amount of shares.

Branches and code coverage

Intended branches

- computes amount of shares to burn
- invokes `_withdraw` with the requested amount of WETH and the calculated amount of shares

Tests cover the intended functionality.

Negative behavior

- `assets` is zero
  ☑ Negative test
- user was not granted approval for the shares
  ☑ Negative test
- shares balance is insufficient
  ☑ Negative test
- receiver is zero

☑ Negative test

Preconditions

- caller has been granted approval for the amount of shares needed
- owner has enough shares

Inputs

- `assets`: amount of assets to withdraw
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, appropriate approval has to be granted
- `receiver`: receiver of the assets
  - **Control**: can't be zero
  - **Authorization**: N/A
  - **Impact**:
- `owner`: owner of the shares to be spent
  - **Control**: None
  - **Authorization**: must have granted approval to msg.sender
  - **Impact**:

Function call analysis

None directly, see `_withdraw`.

Function: `redeem`

Intended behavior

Redeems the given amount of shares for the corresponding amount of WETH.

Branches and code coverage

Intended branches

- computes amount of WETH corresponding to the shares being burnt

- invokes `_withdraw` with the requested amount of shares and the calculated amount of WETH

Tests cover the intended functionality.

Negative behavior

- `shares` is zero
  ☑ Negative test
- user was not granted approval for the shares
  ☑ Negative test
- shares balance is insufficient
  ☑ Negative test
- receiver is zero
  ☑ Negative test

Preconditions

- caller has been granted approval for the amount of shares to redeem
- owner has enough shares

Inputs

- `shares`: amount of shares to redeem
  - **Control**: Can't be zero
  - **Authorization**: N/A
  - **Impact**: None, appropriate approval has to be granted
- `receiver`: receiver of the assets
  - **Control**: can't be zero
  - **Authorization**: N/A
  - **Impact**:
- `owner`: owner of the shares to be spent
  - **Control**: None
  - **Authorization**: must have granted approval to msg.sender
  - **Impact**:

Function call analysis

None directly, see `_withdraw`.

Function: `_deposit`

Intended behavior

Internal function handling depositing and minting.

Branches and code coverage

Intended branches

- transfers WETH from user to contract
- deposits WETH from contract to BendPool
- mints shares to the user

The function is tested by tests for `deposit` and `mint`.

Negative behavior

The function is tested by tests for `deposit` and `mint`.

Preconditions

- `msg.sender` has granted approval for the amount to deposit
- `msg.sender` has enough WETH balance

Inputs

- `assets`: amount of assets to deposit
  - **Control**: None directly
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `shares`: amount of shares to mint
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller

Function call analysis

- `weth.transferFrom(msg.sender, address(this), assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable, nor checked. Technically a violation of ERC20 interface, but WETH reverts on failure.
  - **What happens if it reverts, reenters, or does other unusual control flow?** WETH does not allow to reenter
- `weth.approve(poolAddress, assets)`
  - **What is controllable?** `assets`
  - **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?** WETH does not allow to reenter
- `IBendLendPool(poolAddress).deposit(WETH, assets, address(this), 0)`
  - **What is controllable?** `assets`
  - **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?** If the callee reverts the whole transaction reverts, deposit doesn't happen

Function: `_withdraw`

Intended behavior

Internal function handling withdraws and redeems.

Branches and code coverage

Intended branches

- spends allowance if the caller is not the owner of the shares
- burns the shares
- approves the pool to manage the required amount of pool tokens
- withdraws WETH from the pool, sending them to the receiver

The function is tested by tests for `withdraw` and `redeem`.

Negative behavior

The function is tested by tests for `withdraw` and `redeem`.

Preconditions

- `msg.sender` was granted approval for the amount of shares to redeem
- `owner` has enough shares

Inputs

- `caller`:
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `receiver`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: N/A
  - **Impact**: None, funds are taken from the caller
- `owner`: receiver of the minted tokens
  - **Control**: None
  - **Authorization**: owner must have approved `msg.sender`
  - **Impact**: None, since there's authorization
- `assets`: amount of assets to withdraw
  - **Control**: None directly
  - **Authorization**: N/A
  - **Impact**: None, shares are taken from the owner who must have approved `msg.sender`
- `shares`: amount of shares to burn
  - **Control**: None directly
  - **Authorization**: N/A
  - **Impact**: None, shares are taken from the owner who must have approved `msg.sender`

Function call analysis

- `bToken.approve(poolAddress, assets)`
  - **What is controllable?** `assets`
  - **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?** The called contract is trusted not to allow reentrancy
- `IBendLendPool(poolAddress).withdraw(WETH, assets, receiver)`
  - **What is controllable?** `assets`, `receiver`
  - **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?** The called contract is trusted not to allow reentrancy

Function: `totalAssets`

Intended behavior

get the total AUM by getting the balance of the LP token.

Preconditions

None

Inputs:

None

Analysis:

None

Function call analysis

- `lpTokenAddress.balanceOf(this)`:
  - **What is controllable?**: only upwards through direct transfer
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: AUM

5.5 File: Vault.sol

Function: `previewDeposit (same as OZ ERC4626)`

Function: `previewMint (same as OZ ERC4626)`

Function: `previewWithdraw (same as OZ ERC4626)`

Function: `previewRedeem() (same as OZ ERC4626)`

Function: `deposit`

Intended behavior

deposits the assets and mints the user shares.

Preconditions

None

Inputs:

- `assets`:
  - **Control**: Full Control
  - **Authorization**: (> 0), should check maxDeposit.
  - **Impact**: determines amount of shares
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: None
  - **Impact**: determines who receives the shares.

Function call analysis

- `previewDeposit`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: shares to min
- `_deposit`:
  - **What is controllable?**: assets, shares, receiver

  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: discarded
- `_asset.safeTransferFrom`:
  - **What is controllable?**: amount of asset
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (you don't have enough asset/allowance.)
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded

Function: `mint`

Intended behavior

users asks for N shares, as has to supply equivalent amount of asset.

Preconditions

None

Inputs:

- `shares`:
  - **Control**: Full Control
  - **Authorization**: (> 0), should check maxShares.
  - **Impact**: determines amount of asset to ask for and num shares to mint.
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: None
  - **Impact**: determines who receives the shares.

Function call analysis

- `previewMint`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok

  - **If return value is controllable, how is it used and how can it go wrong**: asset needed
- `_deposit`:
  - **What is controllable?**: assets, shares, receiver
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: discarded
- `_asset.safeTransferFrom`:
  - **What is controllable?**: amount of asset
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (you don't have enough asset/allowance.)
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded

Function: `withdraw`

Intended behavior

users withdraws N amount of assets.

Preconditions

user has the shares

Inputs:

- `assets`:
  - **Control**: Full Control
  - **Authorization**: (> 0), should check maxWithdraw.
  - **Impact**: determines amount of shares
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: != 0
  - **Impact**: determines who receives the shares.
- `owner`:
  - **Control**: Full Control
  - **Authorization**: if msg.sender is not owner, check allowance.
  - **Impact**: whose allowance to spend

Function call analysis

- `previewWithdraw`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: shares to withdraw (including fees)
- `_convertToShares`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: get shares for assets without fees
- `_convertToAssets`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: convert the the fee portion to asset.
- `_withdraw`:
  - **What is controllable?**: assets, shares, receiver, owner
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: discarded

Function: `Redeem`

Intended behavior

users withdraws N amount of shares of assets.

Preconditions

user has the shares

Inputs:

- `shares`:
  - **Control**: Full Control
  - **Authorization**: (> 0), should check maxWithdraw.
  - **Impact**: determines amount of shares and asset asked for.
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: != 0
  - **Impact**: determines who receives the shares.
- `owner`:
  - **Control**: Full Control
  - **Authorization**: if msg.sender is not owner, check allowance.
  - **Impact**: whose allowance to spend

Function call analysis

- `previewRedeem`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: amount of asset to withdraw with fees
- `_convertToAssets`:
  - **What is controllable?**: Full Control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: calculate the shares into asset without fees, calculate the fee part and calculate the fees
- `_withdraw`:
  - **What is controllable?**: assets, shares, receiver, owner
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: discarded

Function: `_deposit`

Intended behavior

execute the actual deposit, and convert asset to the investment instrument

Preconditions

None

Inputs:

- `caller`:
  - **Control**: msg.sender
  - **Authorization**: None
  - **Impact**: determines if the caller is whitelisted.
- `assets`:
  - **Control**: Full Control depending on call to deposit/mint
  - **Authorization**: result of convertToShares/Assets depending on deposit/mint
  - **Impact**: determines amount to convert.
- `shares`:
  - **Control**: Full Control depending on call to deposit/mint
  - **Authorization**: result of convertToShares/Assets depending on deposit/mint
  - **Impact**: determines amount to mint.
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: None
  - **Impact**: who receives the vault shares.

Function call analysis

- `getRoleMemberCount`:
  - **What is controllable?**: None
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: Are there whitelisted addresses?
- `hasRole`:

  - **What is controllable?**: No control
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: is msg.sender a whitelisted address?
- `_mint`:
  - **What is controllable?**: receiver & shares
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded

Function: `_withdraw`

Intended behavior

the actual withdrawal implementation

Preconditions

you have shares to burn.

Inputs:

- `caller`:
  - **Control**: None (always msg.sender)
  - **Authorization**: None
  - **Impact**: determine if allowance check is needed.
- `receiver`:
  - **Control**: Full Control
  - **Authorization**: None
  - **Impact**: who receives withdrawn assets.
- `owner`:
  - **Control**: Full Control
  - **Authorization**: branch.
  - **Impact**: determines if an allowance check is needed.
- `assets`:
  - **Control**: full/partial control depending on whether call is withdraw/redeem.

  - **Authorization**: token balance
  - **Impact**: amount to withdraw
- `shares`:
  - **Control**: full/partial control depending on whether call is withdraw/redeem.
  - **Authorization**: token balance
  - **Impact**: amount to withdraw
- `fees`:
  - **Control**: % of assets.
  - **Authorization**: None
  - **Impact**: fees to take.

Function call analysis

- `_spendAllowance`:
  - **What is controllable?**: owner and shares
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (fails if you don't have enough allowance.)
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded
- `asset.balanceOf`:
  - **What is controllable?**: can control upwards using direct transfers.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
  - **If return value is controllable, how is it used and how can it go wrong**: used to determine if there's enough liquid asset to withdraw
- `_burn`
  - **What is controllable?**: owner & shares, after allowance check
  - **What happens if it reverts, reenters, or does other unusual control flow?**: don't have enough shares.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded
- `_asset.safeTransfer`:
  - **What is controllable?**: receiver, and asset (checked)
  - **What happens if it reverts, reenters, or does other unusual control flow?**: ok
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded
- `getRoleMember`:

- **What is controllable?**: No Control
- **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
- **If return value is controllable, how is it used and how can it go wrong**: determines the address that fees are sent to.

6 Audit Results

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

During our audit, we discovered six findings. Of these, one was high risk, one was medium risk, two were low risk, and two were suggestions (informational). Spice Finance Inc. acknowledged all findings and implemented fixes.

6.1 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.