Zellic

Vendor Finance

Smart Contract Security Assessment

April 12, 2023

Prepared for:

Vendor Finance

Prepared by:

Katerina Belotskaia and Ulrich Myhre

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 Vendor Finance from February 27th to March 2nd, 2023. During this engagement, Zellic reviewed Vendor Finance's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are users able to bypass the restrictions on borrowing funds?
- Can an on-chain attacker drain the funds from the pool?
- Can an on-chain attacker drain the funds from the strategy?
- Is it possible to roll over into a nonvalid contract?

1.2 Non-goals and Limitations

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

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

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

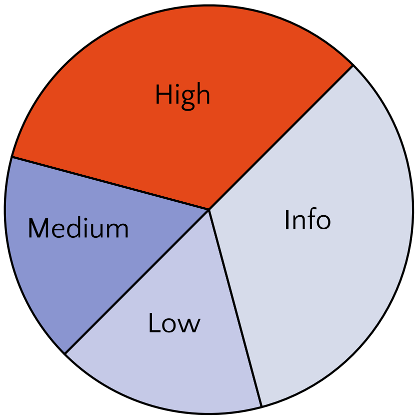
1.3 Results

During our assessment on the scoped Vendor Finance contracts, we discovered six findings. No critical issues were found. Of the six findings, two were high risk, one was medium risk, one was low risk, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Vendor Finance's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	1
Informational	2



2 Introduction

2.1 About Vendor Finance

Vendor Finance is a permissionless lending platform where anyone can create a loan on their own terms without the risk of liquidation. Users can configure loan duration, rate, and tokens accepted and even allow only specific borrowers. This version introduces strategies. With strategies, all idle funds can now be deployed into any ERC4626 vault.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Vendor Finance Contracts

Repository	https://github.com/VendorFinance/contractsv2
Versions	`6a97b73b36ae3057eee60e722129685f13aa5a58`
Programs	• FeesManager.sol • Oracle.sol • PoolFactory.sol • PositionTracker.sol • IERC4626.sol • IFeesManager.sol • IGenericPool.sol • ILicenseEngine.sol • IOracle.sol • IPoolFactory.sol • IPositionTracker.sol • IStrategy.sol • IVendorOracle.sol • IgOHM.sol • ILendingPool.sol • LendingPool.sol • LendingPoolUtils.sol • Vendor4626Strategy.sol • AaveV3ERC4626.sol • AaveV3ERC4626Factory.sol • IPool.sol • IRewardsController.sol • ERC4626Factory.sol • GenericUtils.sol • Types.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

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:

Katerina Belotskaia, Engineer
kate@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

February 24, 2023	Kick-off call
February 27, 2023	Start of primary review period
March 2, 2023	End of primary review period
March 8, 2023	Closing call

3 Detailed Findings

3.1 Denial of service on behalf of borrower

- **Target**: LendingPool, PositionTracker
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

In LendingPool, the function `borrowOnBehalfOf()` enables a user to deposit collateral to the lending pool and sends the borrowed tokens to another user. Any user can do this, and any user can also pay back the loan on behalf of another user. The result is that a borrow position is opened in the PositionTracker contract on behalf of the borrower when a loan is taken, and this position is supposed to be closed when the loan and fees are paid back.

When an attacker borrows zero tokens on behalf of a victim user, a borrow position is created for the victim user with debt equal to zero. Attempting to repay a loan when there is no debt will result in a reverting `NoDebt()` error. A similar situation can arise if a very small loan is taken on behalf of a victim user, but then they at least get some tokens for it, and they are able to unlock their account by paying it back with some fees.

Impact

The borrow position becomes impossible to close, and the victim cannot borrow anything from that pool instance forever. A test case that reproduces the scenario has been implemented within the existing test framework. This test should not pass after remediation.

```
it(“[Bug1] – DoS on behalf of user”, async function () {
  await usdc.mint(pool.address, USDC_1000);
  await weth.mint(userA.address, WETH_5);
  await weth.connect(userA).approve(pool.address, WETH_5);
  let feeRate = await feesManager.getCurrentRate(pool.address);

  await pool.connect(userA).borrowOnBehalfOf(userB.address, 0, feeRate);
  await expect(
      pool.connect(userA).repayOnBehalfOf(userB.address, 0)
```

```
    ).to.be.revertedWithCustomError(LendingPoolImplementation, “NoDebt”);

  await expect(
      pool.connect(userA).borrowOnBehalfOf(userB.address, WETH_5,
    feeRate)
  ).to.be.revertedWithCustomError(positionTracker,
    “PositionIsAlreadyOpen”);
  expect((await pool.debts(userB.address)).debt).to.equal(0);
});
```

Recommendations

To prevent errors and potential abuse, we recommend disallowing loans of zero tokens. Users may accidentally input zero-valued parameters or attempt to exploit the system by depositing an insignificant amount and blocking a victim’s account until they pay it back with interest. To address this, consider setting a minimum loan amount that users must borrow or adding other safeguards to ensure the integrity of the lending system.

Remediation

Vendor Finance acknowledged this finding and implemented a fix in commit c5331198.

3.2 Hardcoded `expiry` and `protocolFee`

- **Target**: LendingPool, FeesManager
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Low

Description

The `LendingPool.setPoolRates()` is a wrapper function for `FeesManager.setPoolRates()` that sends `expiry` and `protocolFee` to the FeesManager contract by passing the global `poolSettings` struct. This makes the parameters uncontrollable by the pool owner, but the values are not verified in any meaningful way in the receiver (FeesManager). A risk is that an upgrade to LendingPool changes or forgets this calling convention, which makes FeesManager's `expiry` or `protocolFee` go out of sync with the LendingPool `expiry` time.

```
function setPoolRates(bytes32 _ratesAndType) external {
    onlyOwner();
    onlyNotPaused();
    feesManager.setPoolRates(address(this), _ratesAndType,
    poolSettings.expiry, poolSettings.protocolFee);
}
```

The scenario is highly unlikely to happen in practice, as this is a central part of the contract functionality, and we expect it will be tested. So while it would be impactful if it happens, the impact has been adjusted to Low to reflect its unlikeliness.

Impact

If the `expiry` suddenly changes to an invalid value (e.g., in the past or distant future), the calculation `FeesManager.getCurrentRate(pool)` will misbehave. This can lead to the rate suddenly becoming 0 or the wrong decay being calculated, or it can have no discernable effect.

```
function getCurrentRate(address _pool) external view returns (uint48) {
    RateData memory rateData = poolFeesData[_pool];
    if (rateData.rateType == FeeType.NOT_SET) revert NotAPool();
    if (block.timestamp > 2**48 - 1) revert InvalidExpiry(); // Per
    auditors suggestion if timestamp will overflow.
    if (rateData.poolExpiry <= block.timestamp) return 0; // Expired pool.
    if (rateData.rateType == FeeType.LINEAR_DECAY_WITH_AUCTION) {
        return computeDecayWithAuction(rateData, rateData.poolExpiry);
    }else if(rateData.rateType == FeeType.FIXED){
        return rateData.startRate;
    }
    revert InvalidType();
}
```

Recommendations

Looking at the contracts in isolation can be a good way to avoid upgradable mistakes in the future. We recommend that FeesManager implement basic sanity checks on the `protocolFee` and `expiry` time before accepting them. These checks can help prevent potential errors from occurring down the line.

Remediation

This issue has been acknowledged by Vendor Finance.

3.3 Lack of approved borrower check during rollover to the private pool

- **Target**: LendingPool
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The PoolFactory contract allows any caller to create private and public pools. A private pool contains a list of approved borrowers managed by the pool owner. If users lend funds using the `borrowOnBehalfOf()` function from the private pool, the transactions from the nonapproved users will be rejected.

Public pool borrowers can roll over to a private pool using the `rollInFrom()` function if the pool meets the following conditions:

- The private pool uses the same `lendToken` and `colToken` as the public pool.
- The private pool has the same `owner` as the public pool.
- The `expiry` time of the new pool is more than the `expiry` time of the current pool.

Impact

The `rollInFrom()` function does not check the new borrowers if a pool is private, so any existing borrowers from public pools can roll over to the private pools. This allows bypassing the prohibition on getting a loan by nonwhitelisted users.

Recommendations

We recommend adding a check that `msg.sender` is an allowed borrower to the `rollInFrom` function.

```
function rollInFrom(
        address _originPool,
        uint256 _originDebt,
        uint48 _rate
    ) external nonReentrant {
        [ ... ]
        if ((settings.borrowers.length > 0)
    && (!allowedBorrowers[msg.sender])) revert PrivatePool(); // @audit
    add this check
        if (settings.pauseTime ≤ block.timestamp)
    revert BorrowingPaused();
        if (effectiveBorrowRate > _rate) revert FeeTooHigh();
        onlyNotPaused();
        if (block.timestamp > settings.expiry) revert PoolExpired(); //
    Can not roll into an expired pool
        LendingPoolUtils.validatePoolForRollover(
            originSettings,
            settings,
            _originPool,
            factory
        );
        [...]
    }
```

Remediation

Vendor Finance acknowledged this finding and implemented a fix in commit afb48cc6.

3.4 The lenderTotalFees can be mistakenly reset

- **Target**: LendingPool
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The withdraw function allows the pool owner to withdraw the lendToken to their address. If the pool uses a strategy, funds will first be withdrawn from the strategy contract before being transferred to the pool owner.

The caller can specify the amount to withdraw by passing the _withdrawAmount parameter. If the _withdrawAmount value is equal to type(uint256).max, then all available tokens in the strategy will be withdrawn, and the actual withdrawn amount will be reflected in the balanceChange value. If the _withdrawAmount value is less than the maximum, then the balanceChange value will equal the _withdrawAmount value passed by the caller.

If the owner attempts to withdraw tokens greater than the lenderTotalFees, then the lenderTotalFees will be reset to zero. Otherwise, the lenderTotalFees will be decreased by the _withdrawAmount value.

```
function withdraw(
        uint256 _withdrawAmount
    ) external nonReentrant {
        GeneralPoolSettings memory settings = poolSettings;
        onlyOwner();
        onlyNotPaused();
        if (block.timestamp > settings.expiry) revert PoolExpired(); //
    Use collect after expiry of the pool

        uint256 initLendTokenBalance
    = settings.lendToken.balanceOf(address(this));
        uint256 balanceChange;
        if (address(strategy) != address(0)) {
            strategy.beforeLendTokensSent(_withdrawAmount); // Taxable
    tokens should not work with strategy.
            balanceChange = settings.lendToken.balanceOf(address(this))
    - initLendTokenBalance;
            if (_withdrawAmount != type(uint256).max && balanceChange
    < _withdrawAmount) revert FailedStrategyWithdraw();
    } else {
      balanceChange = _withdrawAmount;
    }

    lenderTotalFees = _withdrawAmount < lenderTotalFees
  ? lenderTotalFees - _withdrawAmount : 0;
    GenericUtils.safeTransfer(settings.lendToken, settings.owner,
  balanceChange);
    emit Withdraw(msg.sender, _withdrawAmount);
  }
```

Impact

The `lenderTotalFees` can be mistakenly reset in certain situations.

1. If the `strategy` is not zero and the caller passes the `withdrawAmount` equal to `type(uint256).max`, then the `_withdrawAmount` will be greater than the `lenderTotalFees`, regardless of how many tokens were actually withdrawn.
2. If the `strategy` is zero and the caller passes the `withdrawAmount` equal to `type(uint256).max`, the transaction will not be reverted inside the `GenericUtils.safeTransfer` function. This is because, in this case, the `GenericUtils.safeTransfer` function will transfer the current balance, regardless of whether it is less than the `lenderTotalFees`.

```
function safeTransfer(
    IERC20 _token,
    address _account,
    uint256 _amount
  ) external{
    uint256 bal = _token.balanceOf(address(this));
    if (bal < _amount) {
      _token.safeTransfer(_account, bal);
      emit BalanceChange(address(_token), _account, false, bal);
    } else {
      _token.safeTransfer(_account, _amount);
      emit BalanceChange(address(_token), _account, false, _amount);
    }
  }
```

3. If the strategy is zero and the caller passes a withdrawAmount that is greater than the lendToken balance of contract. This could result in the lenderTotalFees being mistakenly reset, just as described in the second point.

If the lenderTotalFees is mistakenly reset, more lend funds will be available to borrowers than expected by the owner.

Recommendations

When the pool uses a strategy, it is important to compare the lenderTotalFees with the balanceChange value before making any updates. If the balanceChange value is greater than the lenderTotalFees, then the lenderTotalFees should be reset to zero. On the other hand, if the balanceChange value is less than or equal to the lenderTotalFees, then the lenderTotalFees should be decreased by the balanceChange value.

If the pool does not use a strategy, then the current lendToken balance of the contract should be obtained and compared with the _withdrawAmount. If the _withdrawAmount is more than the current balance, then _withdrawAmount should be updated to the balance value. Only then can the lenderTotalFees be compared and reduced if the _withdrawAmount is less than lenderTotalFees, or reset if it is not.

Remediation

Vendor Finance acknowledged this finding and implemented a fix in commit b85e552f.

3.5 Missing test coverage

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

Description

While the overall test coverage of the project is very good, there are some critical functionalities that have not been fully tested. Specifically, there is a lack of negative test cases, which are essential for ensuring the platform's resilience to unexpected inputs and edge cases.

As such, it is recommended that the development team focus on writing negative test cases for critical functionalities that have not been fully tested. These test cases can be relatively short and quick to write and execute, but they are essential for identifying potential vulnerabilities and ensuring that the platform is able to handle unexpected situations.

The threat model section of this report will mention some missing test coverage, but here are some highlights for critical functionality:

- FeesManager-setPoolRates()->`validateParams()`
- PoolFactory->`grantOwnership()` and `claimOwnership()`
- PoolFactory->`deployPool()`, check that all reverts work as expected
- Multiple functions that have the `onlyOwnerORFirstResponder()` modifier are only verified with the `owner`. Testing with the least amount of privileges is better here.

Impact

Even minor missing test cases may lead to large mistakes during future code changes. Comprehensive test coverage is essential to minimize the risk of errors and vulnerabilities. It helps to identify potential issues early, reduce debugging, and increase the reliability of the platform. Prioritizing test coverage for major functionalities and edge cases is crucial for ensuring a robust and reliable platform.

Recommendations

Implement missing negative test cases for the most critical business logic (ownership transfer, special privilege functions, pausing and unpausing, etc.).

Remediation

Vendor Finance expanded the test suite by writing additional tests for critical functionalities in commits 1f81d121, 6b1b59e3 and ff44da5c.

3.6 Contracts provide a function to renounce ownership

- **Target**: FeesManager, PositionTracker, Oracle, Vendor4626Strategy
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The Oracle, Vendor4626Strategy, FeesManager, and PositionTracker contracts implement Ownable and OwnableUpgradeable functionality, which provides a method `renounceOwnership` to remove the current owner. This is likely not a desired feature.

Impact

Calling `renounceOwnership` would leave the contract without an owner.

Recommendations

Override the `renounceOwnership` function:

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

Remediation

This issue has been acknowledged by Vendor Finance.

4 Discussion

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

4.1 Lack of check on amount of tokens sent with GenericUtils.safeTransfer function

The GenericUtils.safeTransfer function does not guarantee that the specified _amount of tokens will be transferred to the designated _account. This function is utilized within the repayOnBehalfOf function in instances where a borrower is repaying their loan as well as in the case of a rollover loan to a new pool. If the current colToken balance of the pool is less than the specified colAmount, then only the remaining available funds will be transferred to the receiver.

```
function safeTransfer(
        IERC20 _token,
        address _account,
        uint256 _amount
    ) external{
        uint256 bal = _token.balanceOf(address(this));
        if (bal < _amount) {
            _token.safeTransfer(_account, bal);
            emit BalanceChange(address(_token), _account, false, bal);
        } else {
            _token.safeTransfer(_account, _amount);
            emit BalanceChange(address(_token), _account, false, _amount);
        }
    }

    function repayOnBehalfOf(
        address _borrower,
        uint256 _repayAmount
    ) external nonReentrant returns (uint256 lendTokenReceived,
    uint256 colReturnAmount){
        ...
        if (factory.pools(msg.sender)) { // If rollover
            if (_repayAmount != report.debt)
    revert RolloverPartialAmountNotSupported();
```

```
            if (!allowedRollovers[msg.sender])
    revert PoolNotWhitelisted();
            GenericUtils.safeTransfer(settings.colToken, msg.sender,
    report.colAmount);
            delete debts[_borrower];
        } else {
            ...
            GenericUtils.safeTransfer(settings.colToken, _borrower,
    colReturnAmount);
            ...
        }
    }
```

Additionally, the `rollInFrom` function does not include a check to ensure that the `colReturned` value matches the `colAmount` value of the borrower from the `_originPool` contract. As a result, if there is a discrepancy between these values, the `rollInFrom` function call will not be reverted.

```
function rollInFrom(
        address _originPool,
        uint256 _originDebt,
        uint48 _rate
    ) external nonReentrant {
        ILendingPool originPool = ILendingPool(_originPool);
        GeneralPoolSettings memory settings = poolSettings;
        GeneralPoolSettings memory originSettings
    = originPool.getPoolSettings();
        ...
        uint256 colReturned;
        { // Saving some stack space
            uint256 initColBalance
    = settings.colToken.balanceOf(address(this));
            originPool.repayOnBehalfOf(msg.sender, _originDebt);
            colReturned = settings.colToken.balanceOf(address(this))
    - initColBalance;
        }
        ...
    }
```

During the expected operation of the protocol, the pool balance cannot be less than the number of tokens deposited by borrowers. But at the same time, the successful execution of the function does not guarantee the withdrawal of the expected amount of tokens.

4.2 Centralization risks

Overview

In the given contract setup, the owners of the LendingPool implementation have very strong control of the activities in the pool. Some of these control mechanics are necessary in order to intervene during a catastrophic event, when user funds are at risk. In other places, the owners have deliberately distanced themselves away from being able to make choices on behalf of lenders (e.g., when it comes to upgrading the implementation contract for a deployed pool). If an administrator role is compromised, it is important to be aware of the potential consequences and ways to take action. This can happen if access to the wallet of an owner, or first responder, is obtained through direct hacks or social engineering.

Administrative powers and responsibilities

Pausing a pool, a token or the protocol (a full stop) also blocks repayment in addition to borrowing, but the time keeps advancing. During the time between pausing and unpausing, pools can expire, and the borrower(s) will all default on their loan and lose their collateral as soon as the pool owner gets access to call `collect()`. In the current audited code, there is no possibility to remediate this situation after a pause. The developers do plan to let lenders extend the expiry time after a vendor-initiated pause, or to allow repayment during a pause, but these are currently not implemented. The pausing functionality does have some granularity in that the vendor can pause the protocol, just a token, or just a specific lending pool.

Vendor security

The vendor plans to use Gnosis multisig wallets combined with OpenZeppelin timelock to minimize the risk of both general mistakes and losing control of the owner's wallet(s).

4.3 The `poolSettings.borrowers` can be out of sync with `allowedBorrowers`

Inside the LendingPool contract, the `poolSettings.borrowers` variable contains the borrower's list, and initially, the mapping `allowedBorrowers` includes the same addresses. But the owner can update the `allowedBorrowers`, so the `poolSettings.borr` owers can be out of sync with allowedBorrowers. It is not a problem inside the current implementation of the pool because the borrowers list is mostly unused (only the length of this list is used), but because LendingPool is an upgradable contract, it can be used in a future implementation and it must be taken into account that the addresses inside the lists may not match.

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.

> Please be advised that our threat model was conducted on commit 6a97b73b, which reflects the state of the codebase at a specific point in time. Thus, the absence of a particular test in our report may not necessarily reflect the current state of the test suite. Please see the remediation for finding 3.5 to see additional tests added by Vendor Finance during the remediation period.

5.1 File: FeesManager.sol

Function: `setPoolRates(address _lendingPool, byte[32] _ratesAndType, uint48 _expiry, uint48 _protocolFee)`

Sets the fees and rates for a given lending pool and a given time frame. Can be called by the pool factory or the pool itself through `pool.setPoolRates`. The lender can call the pool function, which indirectly calls this function.

Inputs

- `_lendingPool`
  - **Control**: Not controlled.
  - **Constraints**: Has to be a valid, registered pool checked with `factory.pools(addr)`. Only the pool factory or the pool itself can change its stored params.
  - **Impact**: Decides which pool to change the rates, expiry, and type for. Note that type cannot be changed once set.
- `_ratesAndType`
  - **Control**: Controlled, but custom parsing in `parseRates()`.
  - **Constraints**: The rateType cannot be NOT_SET or be changed if it was already set. Checked in `validateParams()` together with _expiry and _protocolFee.The rateType must be either LINEAR_DECAY_WITH_AUCTION or

FIXED. If rateType is LINEAR_DECAY_WITH_AUCTION, then endRate must be smaller or equal to startRate and auctionEndDate must be larger than auctionStartDate. Otherwise, these are not checked nor used. If rateType is LINEAR_DECAY_WITH_AUCTION, fee + rate cannot be 100% or more. If rateType is FIXED, only the startRate field is used as the rate, and fee + rate cannot be 100% or more.
    - **Impact**: Decides the type of the pool (forever) and the fee rates in use.
- `_expiry`
    - **Control**: Not controlled - taken from poolSettings during the call.
    - **Constraints**: None, except if rateType is LINEAR_DECAY_WITH_AUCTION.
    - **Impact**: Only used in getCurrentRate and will make it return 0 if block timestamp > _expiry
- `_protocolFee`.
    - **Control**: Not controlled - taken from poolSettings during the call.
    - **Constraints**: Together with the pool rate, it cannot exceed 100% rate.
    - **Impact**: Decides the vendor fee in percentage.

Branches and code coverage (including function calls)

Intended branches

- Rates changed successfully.
  ☑ Positive test case

Negative behavior

- Reverts when `_lendingPool` is not whitelisted.
  ☐ Negative test case
- Reverts when not called by the factory or `_lendingPool`.
  ☐ Negative test case
- Reverts when `validateParams()` fails (multiple ways).
  ☐ Negative test case

Function call analysis

- `setPoolRates` → `factory.pools(_lendingPool)`
    - **What is controllable?**: Only controllable if called directly on the contract, which will make it revert unless caller is whitelisted.
    - **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?**: N/A.

- `setPoolRates` → `parseRates(_ratesAndType)`
  - **What is controllable?** Controllable.
  - **If return value controllable, how is it used and how can it go wrong?** It can set strange rates, which will fail later validations. Expiry is not affected.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `setPoolRates` → `validateParams(_parsedRateData, _expiry, _protocolFee)`
  - **What is controllable?**Oonly `_parsedRateData` is indirectly controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Invalid values will lead to a revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverting is an intended functionality.

5.2 File: Oracle.sol

Function: `addFeed(address _token, address _feed, bool _isNative)`

Adds a price feed for a given token without changing existing feeds. To actually change a feed, the Oracle contract would need to be redeployed. The function has the `onlyOwner` modifier and can only be called by owners of the Oracle contract (i.e., Vendor Finance).

Inputs

- `_token`
  - **Control**: Only controllable by owner.
  - **Constraints**: Cannot be zero and cannot be present in either the feedsNATIVE or feedsUSD mappings.
  - **Impact**: Decides which token contract to set an oracle for.
- `_feed`
  - **Control**: Only controllable by owner.
  - **Constraints**: Cannot be zero.
  - **Impact**: Sets the oracle address for a given token. It can be against the native token or USD.
- `_isNative`
  - **Control**: Only controllable by owner.
  - **Constraints**: N/A.
  - **Impact**: Decides which mapping to place the oracle in, either feedsNATIVE or feedsUSD. Only one of them can be set.

Branches and code coverage (including function calls)

Intended branches

- Native feed is added successfully.
  - □ Positive test case
- USD feed is added successfully.
  - □ Positive test case

Negative behavior

- Revert if token or feed is not set.
  - □ Negative test case
- Revert if feed is already set for some token.
  - □ Negative test case

This function has no test coverage.

5.3 File: PoolFactory.sol

Function: `claimOwnership()`

Part two of `grantOwnership()`, where someone must prove ownership of the `newOwner` address. If accepted, the owner set in `grantOwnership` becomes the new owner.

Branches and code coverage (including function calls)

Intended branches

- The suggested owner accepts the ownership.
  - ☑ Positive test case

Negative behavior

- Someone else tries to accept ownership.
  - □ Negative test case

Function: `deployPool(DeploymentParameters _params, byte[] additionalData)`

As the initial entry point for the PoolFactory, this function creates, whitelists, and deploys new pools on behalf of the caller (a lender). The lender becomes the owner of the deployed contract, which is a ERC1967Proxy of a vendor-whitelisted LendingPool implementation.

Inputs

- `_params`
  - **Control**: Fully controlled.
  - **Constraints**: `_params` is a `DeploymentParameters` struct with various constraints: `mintRatio` must be nonzero. This is the amount of lent tokens per unit of collateral (18 decimals). `colToken` must be in tokenAllowList and not equal to lendToken (collateral token). `lendToken` must also be in tokenAllowList and not equal to colToken (token to lend out). `feeRatesAndType` - no constraints here, but it is further checked in FeesManager. `poolType` has to match an entry in the implementation whitelist. `strategy` can be zero, otherwise it must pass the `_validateStrategy(strategy, lendToken)` check, which checks if the strategy is whitelisted for the token. `borrowers` - no constraints, can be empty to make a public pool. `initialDeposit` - no constraints. `expiry` must be at least 24 hours from current block timestamp. `maxLTV` - no constraints here; it is used to decide if borrowing should pause when oracle says the loan-to-value hits this ratio. `pauseTime` must be in the future, but not after the expiry time.
  - **Impact**: `_params` is a `DeploymentParameters` struct with various impacts: `mintRatio` decides how many tokens a borrower can lend for every unit of collateral they provide. Cannot be changed later. `colToken` decides which token to accept as collateral. `lendToken` decides which token to lend out and expects the lender to provide this to the pool. `feeRatesAndType` decides the type of rates (fixed, decay) and its parameters. The type cannot be changed, but the rates can be changed later. `poolType` decides which implementation contract to use for the lender-owned proxy contract. `strategy` optionally enables a whitelisted strategy to be used for the pool. This allows a strategy contract to hook withdraw/deposit functions in the LendingPool. `borrowers` - if this has nonzero length, the pool becomes private and only the addresses in this list can borrow. `initialDeposit` - the contract will try to withdraw this many tokens after deploying the contract. More can be added. `expiry` sets the expiration time of the pool and at which point all debts must be paid off, or they are assumed to be defaulted on.
- `additionalData`
  - **Control**: Fully controlled.
  - **Constraints**: None yet.
  - **Impact**: This field is reserved for future upgrades of the contract and is not in use.

Branches and code coverage (including function calls)

Intended branches

- Deploy pool.
  ☑ Test coverage

Negative behavior

- Revert when `fullStop` is enabled.
  ☑ Negative test
- Revert when lend token is not whitelisted.
  ☑ Negative test
- Revert when collateral token is not whitelisted.
  □ Negative test
- Revert when mint ratio is 0.
  □ Negative test
- Revert if expiry is too early.
  □ Negative test
- Revert if the `poolType` does not have a whitelisted implementation contract.
  □ Negative test
- Revert if `pauseTime` is invalid.
  □ Negative test

Function call analysis

- `deployPool` → `_initializePool(args)` → `new ERC1967Proxy(impl, sig)`
  - **What is controllable?**: `additionalData` and `poolSettings`, plus the strategy field of `factorySettings`. All params are checked earlier.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Pool is not deployed, everything reverts.
- `deployPool` → `openLendPosition(msg.sender, poolAddress)`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `deployPool` → `setPoolRates(poolAddress, _params.feeRatesAndType, _params.expiry, protocolFee)`
  - **What is controllable?**: `_params.feeRatesAndType` and `_params.expiry are controllable`, but `feeRatesAndType` will be validated inside the function.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.

    - **What happens if it reverts, reenters, or does other unusual control flow?**: Expected to revert if `feeRatesAndType` is incorrectly configured.
- `deployPool → _setupPool(_params, poolAddress) → IERC20(_params.lendToken).approve(_pool, deposit)`
    - **What is controllable?**: `deposit` amount is controlled
    - **If return value controllable, how is it used and how can it go wrong?**: N/A
    - **What happens if it reverts, reenters, or does other unusual control flow?**: no problem
- `deployPool → _setupPool(_params, poolAddress) → IGenericPool(_pool).deposit(deposit)`
    - **What is controllable?**: `deposit` amount is controlled
    - **If return value controllable, how is it used and how can it go wrong?**: N/A
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Only called if `_params.initialDeposit` > 0, and it will take the whitelisted `lendToken` and approve an initial deposit. If a strategy is selected, this will also call the `afterLendTokensReceived` hook inside the strategy. If the strategy is faulty, a pool will not be deployed.

Function: `grantOwnership(address newOwner)`

Sets up for a multistage owner change process by letting a given address accept the ownership in the next stage.

Inputs

- `newOwner`
    - **Control**: Only existing owner.
    - **Constraints**: None.
    - **Impact**: This address can become the new owner if `claimOwnership()` is called later.

Branches and code coverage (including function calls)

Intended branches

- The owner of the contract offers the ownership to `newOwner`.
  ☑ Positive test case

Negative behavior

- Someone else tries to offer the ownership.
  □ Negative test case

Function: `initialize(address _oracle, uint48 _protocolFee, address _treasury, address _firstResponder, address[] _tokenAllowList)`

Initializes the PoolFactory contract, setting the owner, oracle, treasury, and `firstResponder` address in addition to the protocol fee. A list of token addresses that are allowed to trade in the pool is also initialized. This function has the `initializer` modifier, and constructor disables initializers, so this function cannot be called after the deployment is done.

Branches and code coverage (including function calls)

Intended branches

- Pool is deployed and initialized successfully, with a `tokenAllowList`.
  ☑ Positive test cases
- Pool is deployed and initialized successfully, without a `tokenAllowList`.
  □ Positive test cases

Negative behavior

- Reverts if oracle, treasury, or firstResponder is address(0).
  ☑ Negative test case

Function: `setAllPoolsStop(bool _paused)`

This sets a full stop flag that prohibits the following functions in the lending pool: `borrow`, `repay`, `collect`, `deposit`, `rollover`, `withdraw`, and `withdrawStrategyTokens`, as well as updating borrowing whitelist, setting pool rates, and lender setting borrowing pause for owned pool. All functions have the `onlyNotPaused` modifier. It is a function that supposedly is intended for catastrophic events where customer funds are at great risk. Note that while the pools are paused, the time still ticks, so if `fullStop` is enabled for a long time, every borrower will default on their loans.

Inputs

- `_paused`
  - **Control**: Controlled by owner or first responder.
  - **Constraints**: `bool`.
  - **Impact**: Stops every state-modifying function in every pool.

Function: `setAllowedLendTokensForStrategy(byte[32] _strategy, address[] _tokens, bool[] _enabled)`

A simple setter that sets up whether a given strategy is allowed for use with a given token. This is done through a nested mapping `mapping(bytes32 ⇒ mapping(address ⇒ bool))`.

Inputs

- `_strategy`
  - **Control**: Controllable by owner.
  - **Constraints**: None.
  - **Impact**: Decides which strategy to update token permissions for.
- `_tokens`
  - **Control**: Controllable by owner.
  - **Constraints**: Length must be equal to the length of `_enabled`.
  - **Impact**: Decides what token to update permission for on a given strategy.
- `_enabled`
  - **Control**: Controllable by owner.
  - **Constraints**: Length must be equal to the length of `_tokens`.
  - **Impact**: Decides which permission to set for a given token for a given permission.

Function: `setPauseFactory(bool _pauseEnable)`

Pauses the factory so that new pools cannot be deployed and the contract cannot be upgraded.

Inputs

- `_pauseEnable`
  - **Control**: Only controlled by owner or first responder.
  - **Constraints**: boolean.
  - **Impact**: Pauses all functions with the `whenNotPaused` modifier.

Branches and code coverage (including function calls)

Intended branches

- (Un)pause the contract as owner.
  - ☐ Positive test case
- (Un)pause the contract as first responder.

- □ Positive test case

Negative behaviur

- Called by nonowner and non–first responder.
  - □ Negative test case

Function: `setPoolStop(address _pool, bool _paused)`

Adds a single pool to the list of manually paused pools. Like for setAllPoolsStop, this affects the `onlyNotPaused` modifier, but only for a single contract.

Inputs

- `_pool`
  - **Control**: Controllable only by owner or first responder.
  - **Constraints**: None.
  - **Impact**: The target pool is paused.
- `_paused`
  - **Control**: Controllable only by owner or first responder.
  - **Constraints**: boolean.
  - **Impact**: Decides whether to pause or unpause.

Branches and code coverage (including function calls)

Intended branches

- Owner (un)pauses a pool, and the pool is blocked.
  - □ Positive test cases

Negative behavior

- Pool does not exist.
  - □ Negative test case
  - □ Coded check
- Called by nonowner and non–first responder.
  - □ Negative test case

Function: `setTokenStop(address _token, bool _paused)`

Pauses all deployed pools that accept or offer a given token as collateral/loan, blocking all functions that have the `onlyNotPaused` modifier. See `setPoolStop`.

5.4 File: PositionTracker.sol

Function: `closeBorrowPosition(address _user)`

Deletes a borrow position for a user in the pool. This is called via `LendingPool→repayOnBehalfOf(borrower, payment)`, but only if the reported debt is 0 after the repayment. The linked list of positions is updated to remove a single entry.

Inputs

- `_user`
  - **Control**: Only controllable if the sender is a whitelisted pool.
  - **Constraints**: Must be an open position in the `borrowPositions` mapping.
  - **Impact**: The given address is used in a hash calculation, and the resulting mapping entry is deleted.

Branches and code coverage (including function calls)

Intended branches

- The first borrow position (tail) in a chain is closed.
  ☑ Positive test case
- The last borrow position in a chain is closed.
  ☑ Positive test case
- A borrow position in the middle of a chain is closed.
  ☑ Positive test case

Negative behavior

- Reverts if position is not found
  □ Negative test case
- Reverts if the sender is not a pool
  □ Negative test case

Function call analysis

- `closeBorrowPosition → factory.pools(msg.sender)`
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Reverts.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Expected to revert if `msg.sender` is not a pool.

Function: `closeLendPosition(address _user)`

Deletes a lending position for a user in the pool. This is called indirectly by a lender via `LendingPool→collect()` after the pool has expired.

Inputs

- `_user`
  - **Control**: Controlled through the call to the pure function `getEntry`. Afterwards, it is only controlled if the sender is a whitelisted pool.
  - **Constraints**: Only whitelisted pool address can set, and it has to map to a valid lending position to not revert.
  - **Impact**: Deletes the current lending position for the given `_user` and updates the mappings and linked lists.

Branches and code coverage (including function calls)

Intended branches

- The first lend position (tail) in a chain is closed.
  ☑ Positive test case
- The last lend position in a chain is closed.
  ☑ Positive test case
- A lend position in the middle of a chain is closed.
  ☑ Positive test case

Negative behavior

- Reverts if position is not found.
  □ Negative test case
- Reverts if the sender is not a pool.
  □ Negative test case

Function call analysis

- `closeLendPosition → factory.pools(msg.sender)`
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Reverts.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Expected to revert if `msg.sender` is not a pool.

Function: `openBorrowPosition(address _user)`

Opens up a borrow position on behalf of a user. This function is called from LendingPool as part of the `borrowOnBehalfOf` and `rollInFrom` functions.

Inputs

- `_user`
  - **Control**: Controllable if called via `borrowOnBehalfOf`; otherwise, it is the address that initiated the call in to `rollInFrom`.
  - **Constraints**: From `borrowOnBehalfOf`, it has to be in the `allowedBorrowers` whitelist if a whitelist is set.
  - **Impact**: Opens up a new borrowing position for the given user, provided none already exists. The `_user` is part of an entry hash used for multiple mappings.

Branches and code coverage (including function calls)

Intended branches

- The first borrow position for a user is successfully opened.
  ☑ Positive test case
- Consecutive borrow positions for a user are successfully opened.
  ☑ Positive test case

Negative behavior

- Reverts if sender is not a whitelisted pool.
  □ Negative test case

Function call analysis

- `openBorrowPosition → factory.pools(msg.sender)`
  - **What is controllable?**: Nothing, it is analogous to an access modifier.
  - **If return value controllable, how is it used and how can it go wrong?**: there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?**: no problem
- `openBorrowPosition → getEntry(_user, msg.sender)`
  - **What is controllable?**: `_user` is controllable via `borrowOnBehalfOf`; otherwise, it is the address that initiated the call in to `rollInFrom`.
  - **If return value controllable, how is it used and how can it go wrong?**: Opens up a new borrow position for some address but does not deal with

debts.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.

Function: `openLendPosition(address _user, address _pool)`

Called as part of PoolFactory.sol->`deployPool`, when a lending pool is deployed.

Inputs

- `_user`
  - **Control**: Not controlled; it is always the address of the lender that created a new pool.
  - **Constraints**: Original `msg.sender`.
  - **Impact**: Opens up a new lend position for the given user, provided none already exists. `_user` is part of an entry hash used for multiple mappings.
- `_pool`
  - **Control**: Not controlled.
  - **Constraints**: N/A.
  - **Impact**: The address of the newly deployed lending pool. It is part of the entry hash used for multiple mappings.

Branches and code coverage (including function calls)

Intended branches

- First lend position for a user is successfully opened.
  ☑ Positive test case
- Consecutive lend positions for a user are successfully opened.
  ☑ Positive test case

Negative behavior

- Reverts if a position is already opened.
  ☑ Negative test case
- Reverts if the caller is not a factory.
  □ Negative test case
- Reverts if a the lending pool is not whitelisted.
  □ Negative test case

Function call analysis

Refer to `openBorrowPosition()`. This function is very similar to that function, only using the lend mappings instead of borrow.

5.5 File: AaveV3ERC4626.sol

Function: `redeem(uint256 shares, address receiver, address owner)`

Allows anyone who has shares tokens or has approval to withdraw the asset tokens.

Inputs

- `shares`
  - **Control**: Full control.
  - **Constraints**: The owner should have more or an equal amount of shares, otherwise the `_burn` will revert.
  - **Impact**: The corresponding number of assets will be calculated using the shares amount. And the owner will be able to withdraw this number of assets.
- `receiver`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: The receiver of asset tokens from the lendingPool.
- `owner`
  - **Control**: Full control.
  - **Constraints**: If `msg.sender` is not an owner, `msg.sender` should have approval from the `owner`.
  - **Impact**: The owner of shares.

Function call analysis

- `lendingPool.withdraw(address(asset), assets, receiver);`
  - **What is controllable?**: `receiver`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Will revert if lendingPool does not have enough asset tokens.
- `previewRedeem(shares) → convertToAssets(uint256 shares) → totalAssets() → aToken.balanceOf(address(this))`
  - **What is controllable?**: `shares`.

    - **If return value controllable, how is it used and how can it go wrong?**: This value is using during the assets amount calculation `shares.mulDivDown(totalAssets(), supply)`, the greater the value of `totalAssets`, the more asset tokens will be redeemed by the caller.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: No problems.

Function: `withdraw(uint256 assets, address receiver, address owner)`

Allows anyone who has corresponding amount of shares tokens or has approval to withdraw the asset tokens.

Inputs

- `assets`
    - **Control**: Full control.
    - **Constraints**: The caller should have this `shares` amount, which is calculated using `assets` amount.
    - **Impact**: The caller will receive this amount of asset tokens.
- `receiver`
    - **Control**: Full control.
    - **Constraints**: No checks.
    - **Impact**: The receiver of asset tokens from the lendingPool.
- `owner`
    - **Control**: Full control.
    - **Constraints**: If `msg.sender` is not an owner, `msg.sender` should have approval from the `owner`.
    - **Impact**: The owner of shares.

Function call analysis

- `lendingPool.withdraw(address(asset), assets, receiver);`
    - **What is controllable?**: `receiver`.
    - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Will revert if lendingPool does not have enough asset tokens.
- `previewWithdraw(assets) → totalAssets()→ aToken.balanceOf(address(this))`
    - **What is controllable?**: `assets`.
    - **If return value controllable, how is it used and how can it go wrong?**: This

value is using inside the shares number calculation `assets.mulDivUp(supply, totalAssets())`, the greater the value of `totalAssets`, the less share tokens will be burned to withdraw this `assets` amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problems.

5.6 File: AaveV3ERC4626Factory.sol

Function: `createERC4626(ERC20 asset)`

Allows anyone to deploy the new AaveV3ERC4626 contract.

Function call analysis

- `lendingPool.getReserveData(address(asset));`
  - **What is controllable?**: `asset`.
  - **If return value controllable, how is it used and how can it go wrong?**: Return the `aToken` address from the lendingPool contract. The lendingPool should be a trusted contract to be sure that `aToken` is also trusted.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problems.

5.7 File: LendingPool.sol

Function: `borrowOnBehalfOf(address _borrower, uint256 _colDepositAmount, uint48 _rate)`

Allows any caller to borrow funds from lender. The pool can be private and nonprivate. In case of a private pool, the `_borrower` address should be an allowed borrower, but `msg.sender` can be any address.

If `settings.maxLTV == type(uint48).max`, the price will not checked.

Inputs

- `_borrower`
  - **Control**: The caller controls this value, so it can be any address, for example, `msg.sender` or any other user.
  - **Constraints**: If pool is private, the `allowedBorrowers` should contain the `_borrower` address.
  - **Impact**: `msg.sender` will transfer `colToken` to the contract, but the `_borrower`

will receive the borrowed assets.

- `_colDepositAmount`
  - **Control**: Full control.
  - **Constraints**: `mgs.sender` should have more or an equal amount of `colToken`.
  - **Impact**: This amount of `colToken` will be transfered from the `msg.sender` to this contract.
- `_rate`
  - **Control**: Full control.
  - **Constraints**: `_rate` should be more or equal to `effectiveBorrowRate`.
  - **Impact**: The expected rate value. If `effectiveBorrowRate` is more than `_rate`, the transaction will revert, othewise the `effectiveBorrowRate` will be used for fee calculations.
- `msg.sender`
  - **Control**: N/A.
  - **Constraints**: No checks.
  - **Impact**: `msg.sender` transfers the `_colDepositAmount` amount of the `colToken` to this contract.

Branches and code coverage (including function calls)

Intended branches

- `msg.sender` is not an approved borrower if pool is private.
  - □ Test coverage
- The `_borrower` address is an approved borrower if pool is private.
  - □ Test coverage
- The `_borrower` address is not an approved borrower if pool is not private.
  - □ Test coverage

Negative behavior

- The `_borrower` address is not an approved borrower if pool is private.
  - □ Negative test
- `msg.sender` balance of `colToken` is less than `_colDepositAmount`.
  - □ Negative test
- `_colDepositAmount` is zero.
  - □ Negative test

Function call analysis

- `feesManager.getCurrentRate(address(this));`
  - **What is controllable?**: nothing is controlled, the feesManager address is

passed from the PoolFactory during initialization
  - **If return value controllable, how is it used and how can it go wrong?**: if return value is more than `_rate` transaction will reverted
  - **What happens if it reverts, reenters, or does other unusual control flow?**: no problems
- `positionTracker.openBorrowPosition(_borrower);`
  - **What is controllable?**: nothing is controlled, the feesManager address is passed from the PoolFactory during initialization
  - **If return value controllable, how is it used and how can it go wrong?**: if return value is more than `_rate` transaction will reverted
  - **What happens if it reverts, reenters, or does other unusual control flow?**: no problems

Function: `claimOwnership()`

Allows new owner to claim the ownership. The `_grantedOwner` should be set by the owner using the `grantOwnership` function.

Branches and code coverage (including function calls)

Intended branches

- `_grantedOwner` is new owner.
  □ Test coverage
- `_grantedOwner` is zero address after call.
  □ Test coverage

Negative behavior

- `_grantedOwner` is zero address.
  □ Negative test
- Caller is not `_grantedOwner`.
  □ Negative test

Function: `collect()`

Anyone can trigger this function only after the `expiry` time has come, but tokens will be transferred to the current owner contract. The funds will be withdrawn from the strategy if it is used.

Function: `deposit(uint256 _depositAmount)`

Allows anyone to deposit lending funds, which will be used for issuing borrowing. If strategy is using, these funds will be deposited to the vault.

Function: `grantOwnership(address _newOwner)`

Allows owner of contract to set the `_grantedOwner` address. The `_grantedOwner` should claim the ownership using the `claimOwnership` function.

Function: `initialize(byte[] _factoryParametersBytes, byte[] _poolSettingsBytes, byte[] None)`

The function is supposed to be used from the PoolFactory contract, and it does not affect the system in any way if it was deployed separately; therefore, this function is considered in the context of a call from the `PoolFactory:deployPool` function. The caller of `deployPool` does not control feesManager, oracle, treasury, and posTracker addresses.

Inputs

- `_factoryParametersBytes`
  - **Control**: Full control.
  - **Constraints**: Check nonzero addresses.
  - **Impact**: -
- `_poolSettingsBytes`
  - **Control**: Full control.
  - **Constraints**: Check nonzero addresses, expiry is more than current time, and the type is correct.
  - **Impact**: N/A.

Function call analysis

- `initiateStrategy(_factoryParameters.strategy, _poolSettings.lendToken)` → `IERC20(IStrategy(strategyAddress).getDestination()).approve(strategyAddress, type(uint256).max);`
  - **What is controllable?**: `strategy` and `lendToken` - but both these addresses are validated then whitelisted inside the `deployPool` function, which allows to deploy this pool. `IStrategy(strategyAddress).getDestination()` also a controlled address.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.

  - **What happens if it reverts, reenters, or does other unusual control flow?**: In case of unproper check during whitelisting of strategy contract, the caller can have full control over strategy contract, so it is possible to get an approve for any token address.
- `initiateStrategy(_factoryParameters.strategy, _poolSettings.lendToken)` → `_lendToken.approve(strategyAddress, type(uint256).max)`
  - **What is controllable?**: `strategy` and `lendToken` - but both these addresses are validated then whitelisted inside the `deployPool` function, which allows to deploy this pool.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The `_lendToken` address should be approved by the owner of factory to be used with the strategy, so the caller does not have full control over the lendToken contract.

Function: `repayOnBehalfOf(address _borrower, uint256 _repayAmount)`

The function can be called in two cases. The first is the rollover case, when the loan will be transferfed to the other pool. And the second case is when the caller repays the loan. In case of rollover, only the full amount of debt can be transfefred to the new pool. But in the second case, part repay is possible.

Inputs

- `_borrower`
  - **Control**: Full control.
  - **Constraints**: Should be actual borrower.
  - **Impact**: The colTokens will be transferred to the `_borrower` address.
- `_repayAmount`
  - **Control**: Full control.
  - **Constraints**: If `_repayAmount > report.debt`, will revert.
  - **Impact**: The amount of lenTokens for repay.

Branches and code coverage (including function calls)

Intended branches

- Rollover test from accepted pool and `_repayAmount == report.debt`.
  □ Test coverage
- The borrower repays the full loan.
  □ Test coverage

- The borrower repays part of loan.
  - □ Test coverage

Negative behavior

- The `_borrower` is fake.
  - □ Negative test

Function call analysis

- `GenericUtils.safeTransfer(settings.colToken, msg.sender, report.colAmount);`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the `colToken` balance of the contract is less than `report.colAmount`, will transfer only the remaining part of the expected amount.
- `GenericUtils.safeTransferFrom(settings.lendToken, msg.sender, address(this), _repayAmount);`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the actual number of tokens received.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problem.
- `strategy.afterLendTokensReceived(lendTokenReceived)`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Deposit the repaying amount of `lendTokens` to the strategy.

Function: `rollInFrom(address _originPool, uint256 _originDebt, uint48 _rate)`

Allows borrower to roll over loan into this pool from the pool of the same owner. The pools should have the same owner, `lendToken` and `colToken`, and be deployed over the factory. Also both pools should be active and the expiry of current tokens should be more than the `_originPool` expiry. At first, the `colToken` will be transfered from the `_originPool` to the current pool. If it is necessary, the part of `colTokens` will be reimbursed to the caller - or if the current pool allows to borrow less amount of `lendTokens`, the corresponding amount of tokens will be withdrawn from msg.sender.

Inputs

- `_originPool`
  - **Control**: Full control.
  - **Constraints**: `_originPool` should also be deployed by the same factory contract and should have the same owner, `lendToken` and `colToken`.
  - **Impact**: The `colTokens` from `_originPool` will be transfered to the current pool.
- `_originDebt`
  - **Control**: Full control.
  - **Constraints**: Should be equal to the original debt amount.
  - **Impact**: Using this value, the current debt value will be calculated. If the current `mintRatio` and original `mintRatio` are the same, the new debt will be the same as original.
- `_rate`
  - **Control**: Full control.
  - **Constraints**: `_rate` should be more or equal to `effectiveBorrowRate`.
  - **Impact**: The expected rate value. If `effectiveBorrowRate` is more than `_rate`, the transaction will revert, otherwise the `effectiveBorrowRate` will be used for fee calculations.

Branches and code coverage (including function calls)

Intended branches

- `_originPool` is trusted.
  - ☐ Test coverage
- The caller is a real borrower from the `_originPool`.
  - ☐ Test coverage

Negative behavior

- `_originPool` is not trusted.
  - ☐ Negative test
- `_originDebt` is less than actual.
  - ☐ Negative test
- `_originDebt` is more than actual.
  - ☐ Negative test
- The caller is not an actual borrower inside the `_originPool`.
  - ☐ Negative test

Function call analysis

- `feesManager.getCurrentRate(address(this));`
  - **What is controllable?**: Nothing is controlled; the `feesManager` address is passed from the PoolFactory during initialization.
  - **If return value controllable, how is it used and how can it go wrong?**: If return value is more than `_rate`. transaction will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problems.
- `originPool.repayOnBehalfOf(msg.sender, _originDebt)`
  - **What is controllable?**: `_originDebt`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the `_originDebt` is the same as `debts[_borrower].debt`, the `originPool` will transfer the `colTokens` to the current pool. But if the `originalToken` does not have enought `colTokens`, only a part of funds will be transferred.
- `strategy.afterLendTokensReceived(lendToRepay)`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Deposit the repaying amount of `lendTokens` to the strategy.

Function: `setPauseBorrowing(uint48 _timestamp)`

Allows the contract owner to set the `pauseTime` after that is not possible borrowing. The functions `repayOnBehalfOf` and `rollInFrom` will revert if the current time is more or equal to the `settings.pauseTime` value.

Inputs

- `_timestamp`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: If `_timestamp` is less than the `expiry` time, user will not be able to borrow.

Branches and code coverage (including function calls)

Negative behavior

- Caller is not an owner.
  - ☐ Negative test
- The owner of the factory paused this contract.
  - ☐ Negative test

Function: `setPoolRates(byte[32] _ratesAndType)`

Allows owner of contract to change the fees rate of this pool

Inputs

- `_ratesAndType`
  - **Control**: full controll
  - **Constraints**: no checks
  - **Impact**: the fee rate to be used in calculating the fee for lender taking a loan.

Branches and code coverage (including function calls)

Intended branches

- The new rate is set properly.
  - ☐ Test coverage

Negative behavior

- Caller is not an owner.
  - ☐ Negative test
- The owner of the factory paused this contract.
  - ☐ Negative test

Function call analysis

- `feesManager.setPoolRates(address(this), _ratesAndType, poolSettings.expiry, poolSettings.protocolFee);`
  - **What is controllable?**: `_ratesAndType` -> `rateType`, `auctionStartDate`, `auctionEndDate`, `startRate`, and `endRate`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Can revert if input data is invalid: if `endRate > startRate`, `auctionEndDate ≤ auctionStartDate`, if `_protocolFee` + rate >= 100%.

Function: `setRolloverPool(address _pool, bool _enabled)`

Allows owner of contract to set the approved rollover pool address.

Inputs

- `_pool`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The `_pool` address will be able to transfer the `colToken` to its address from the current contract.
- `_enabled`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: If true, the pool is approved, if false, it is not approved.

Branches and code coverage (including function calls)

Intended branches

- Enable the allowance.
  - □ Test coverage
- Disable the allowance.
  - □ Test coverage

Negative behavior

- Caller is not an owner.
  - □ Negative test

Function: `updateBorrower(address _borrower, bool _allowed)`

Allows the owner of the contract to add new approved `_borrower` addresses or cancel existing addresses. But it is possible only for the private pools. The private pool is a pool that initially has a non-empty list of approved borrowers.

Inputs

- `_borrower`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The address that will receive confirmation for the function call `borrowOnBehalfOf` or the confirmation for this address will be canceled.

- `_allowed`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: True — is allowed to borrow, false — is forbidden to borrow.

Branches and code coverage (including function calls)

Intended branches

- Enable the allowance.
  - □ Test coverage
- Disable the allowance.
  - □ Test coverage

Negative behavior

- Caller is not an owner.
  - □ Negative test
- Not a private pool.
  - □ Negative test

Function: `withdrawStrategyTokens()`

Allows owner to transfer full shares balance from the vault in an emergency situation.

Function: `withdraw(uint256 _withdrawAmount)`

Allows owner of pool ti withdraw `lendToken`. If pool supports strategy, tokens will be withdrawn from the vault. If `_withdrawAmount` is `type(uint256).max`, the full balance will be withdrawn from strategy.

Inputs

- `_withdrawAmount`
  - **Control**: Full control by owner.
  - **Constraints**: no checks
  - **Impact**: If `_withdrawAmount == type(uint256).max`, then the current balance will be withdrawn from strategy.

Branches and code coverage (including function calls)

Intended branches

- Withdraw full balance from stratagy.

    - □ Test coverage
- Withdraw `lendToken` if strategy is not used.
    - □ Test coverage

Negative behavior

- Timestamp is more than expiry time.
    - □ Negative test
- `_withdrawAmount` is more than the current balance.
    - □ Negative test

Function call analysis

- `strategy.beforeLendTokensSent(_withdrawAmount);`
    - **What is controllable?**: `_withdrawAmount`.
    - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: if `_withdrawAmount == type(uint256).max`, will withdraw full current balance.
- `GenericUtils.safeTransfer(settings.lendToken, settings.owner, balanceChange);`
    - **What is controllable?**: `balanceChange` in case strategy == address(0).
    - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: If current balance is less than `balanceChange`, will transfer only the remaining part of the expected amount.

5.8 File: Vendor4626Strategy.sol

Function: `afterLendTokensReceived(uint256 _amount)`

Allows trusted pool contract to deposit the `asset` tokens into the vault.

Inputs

- `_amount`
    - **Control**: Full control by the trusted pool.
    - **Constraints**: The pool should have enough asset tokens.
    - **Impact**: This amount of asset tokens will be deposited to the vault from the pool.

Function call analysis

- `IERC20 asset = IERC20(vault.asset());`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Return the `asset` contract address. The vault should be a trusted contract. Otherwise, it will be possible to withdraw any tokens from the pool, but it is not expected that the pool will have any different tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problem.
- `asset.safeTransferFrom(msg.sender, address(this), _amount);`
  - **What is controllable?**: `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problem.
- `asset.approve(address(vault), _amount);`
  - **What is controllable?**: `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No problem.
- `vault.deposit(_amount, msg.sender);`
  - **What is controllable?**: `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Asset tokens will be transferfed to the vault contract from the strategy, and the corresponding share amount will be minted for the pool.
- `vault.afterDeposit → lendingPool.supply(address(asset), assets, address(this), 0);`
  - **What is controllable?**: Assets - this amount of asset tokens will be transferd from the vault to the aToken contract.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Will revert if `validateSupply` has failed (if reserve is active or if supply capacity exceeded).

Function: `beforeLendTokensSent(uint256 _amount)`

Available only for a trusted pool address. Allows to withdraw asset tokens from the pool. In this case, the `asset` token address is the `lendToken` token. In case of a redeem call, the `vault.balanceOf(msg.sender)` amount of shares will be burned and the corresponding amount of assets will be redeemed from the vault. In case of withdraw call, the passed `_amount` of assets will be redeemed from the vault and the corresponding amount of shares will be burned. To withdraw the tokens from the vault, the corresponding aTokens amount of tokens will be burned (the `assets` amount of tokens) and the `asset` tokens will be transferred to the pool address. After that, this amount of `lendToken` will be transferred to the borrower.

Inputs

- `_amount`
  - **Control**: Full control by the trusted pool.
  - **Constraints**: If the caller does not have the corresponding amount of shares, the transaction will be reverted.
  - **Impact**: The number of asset tokens that withdraw from the vault.

Function call analysis

- `vault.redeem(vault.balanceOf(msg.sender), msg.sender, msg.sender);`
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The full balance of shares will be burned and corresponding amount of assets will be transferred to the `msg.sender`. Will revert if lendingPool does not have enough funds for withdraw.
- `vault.withdraw(_amount, msg.sender, msg.sender);`
  - **What is controllable?**: `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: There is no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Will revert if `msg.sender` does not have enought shares. Will revert if lendingPool does not have enough funds for withdraw.

Function: `setPause(bool _pauseEnable)`

If pause is enabled, then calling `afterLendTokensReceived` and `beforeLendTokensSent` functions will not be possible.

6 Audit Results

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

During our audit, we discovered six findings. Of these, two were high risk, one was medium risk, one was low risk, and two were suggestions (informational). Vendor Finance 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.