Ambit Finance Audit Report

Version 1.0

Conducted by: Kiki

January 2025

Table of Contents

1 Introduction

1.1 About Kiki

Kiki is a Security Researcher who has conducted dozens of security reviews with the top security firm Guardian Audits, as well as through private engagements. View their previous work here, or reach out via Twitter or Telegram.

1.2 Disclaimer

Security Reviews are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

1.3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

1.3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

1.3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

2 Executive Summary

2.1 Ambit Finance

Ambit is cutting-edge, cross-chain DeFi protocol offering users simple yields on stablecoin deposits, sustainable money market lending, and risk-defined portfolio investment strategies - all within a user-friendly environment.

More on the protocol can be found on the official documentation page.

2.2 Overview

Project Name	Ambit Finance
Codebase	ambitfi-contracts
Operating platform	Arbitrum, BNB Chain
Language	Solidity

Audited commits	`[1] fec2466cfa8d21a598be1ebf3dc455be1612c7e8`
	`[2] 7569a8a1998de916816347a434aa9afee2ead79e`

Remediation commit	Pending
Audit methodology	Manual Review

2.3 Scope

Files and folders in scope

`- packages/finance/protocol/market/Market.sol`
`- packages/finance/protocol/market/MarketLib.sol`
`- packages/finance/protocol/market/MarketOracle.sol`
`- packages/finance/protocol/market/DynamicInterestRateModel.sol`
`- packages/finance/protocol/market/MarketRouter.sol`
`- packages/finance/protocol/zaps/Zapper.sol`

2.4 Issues Found

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	0	0	0	0
High risk	3	3	0	0
Medium risk	5	5	0	0
Low risk	7	6	0	1

2.5 Findings & Resolutions

ID	Title	Severity	Status
H-01	Kink Portion of previewRate is Miscalculated	High	Resolved
H-02	Attacker can Profitably Manipulate Share Price	High	Resolved
H-03	Interest Rate can be Artificially Inflated	High	Resolved
M-01	Errantly High Utilization Rate After deposit	Medium	Resolved
M-02	Slippage Check Wont Protect Users	Medium	Resolved
M-03	Attacker can use Leftover Funds to Repay Liabilities	Medium	Resolved
M-04	Allowlist Overlap Allows Unexpected Delegate Calls	Medium	Resolved
M-05	Portfolio Assets are not Utilized in a Write Off	Medium	Resolved
L-01	Reserves can be Temporarily Depleted	Low	Resolved
L-02	Debt Erased When Socialization is Insufficient	Low	Resolved
L-03	Borrowers May Not Pay All Interest	Low	Resolved
L-04	Arbitrary Token can be used in FlashLoan	Low	Resolved
L-05	Users can Escape Debt Socialization	Low	Acknowledged
L-06	Flash Loan Missing Balance Check	Low	Resolved
L-07	Reserve Fee Change can Cause Loss of Yield	Low	Resolved

3 Findings

3.1 High Risk

3.1.1 Kink Portion of previewRate is Miscalculated

Severity: *High risk (Resolved)*

Context: [1] : ***DynamicInterestRateModel.sol****:103*

Description:

The `previewRate` function is intended to calculate the interest rate based on the current utilization. A kink factor is implemented to increase the velocity that the borrowing rate will grow at once utilization exceeds a certain threshold. Once the utilization is greater than or equal to the `_kink` it will use the steeper of the two slopes for this excess. As seen here:

```
    return _baseRate + _slope0 + _slope1.mulDiv(utilization - _kink, 1e18).toUint128
        ();
```

The issue however is that `_slope1` will never be fully utilized because the denominator is 1e18 while what is being multiplied by `_slope1` is capped at the difference between `utilization` and `_kink`. Take the following example where only 30% of `_slope1` will be used despite there being a 100% utilization:

```
_kink: 0.7e18
utilization: 1e18
_slope1: 30%

_slope1.mulDiv(utilization - _kink, 1e18)
30% * (1e18 - 0.7e18) / 1e18
30% * 0.3e18 / 1e18
30% * 0.3
9%
```

From a LP point of view this would result in a loss of yield since interest will accrue at a lower rate than intended.

Recommendation:

Calculate `_slope1` with `1e18 - kink` as the denominator. This will allow the entirety of `slope1` to be utilized.

```
- return _baseRate + _slope0 + _slope1.mulDiv(utilization - _kink, 1e18).toUint128()
    ;
+ return _baseRate + _slope0 + _slope1.mulDiv(utilization - _kink, 1e18 - _kink).
    toUint128();
```

Resolution: Resolved. The recommended fix was implemented in **8859a38**.

3.1.2 Attacker can Profitably Manipulate Share Price

Severity: *High risk (Resolved)*

Context: [-○- 1] : ***Market.sol****:305*

Description

When a user calls the mint function, it will calculate the amount of assets required to receive the requested amount of shares. The calculation uses ERC4626's `_convertToAssets` function, which will utilize the overridden `totalAssets` function.

```
function totalAssets() public view override(ERC4626Upgradeable, IERC4626) returns (
    uint256) {
  MarketStorage storage $ = getMarketStorage();

  (, uint128 accrued) = previewAccrueLiabilities();

  return $.totalAssets + (accrued - $.reserveFee.percentOf(accrued));
}
```

This will return the sum of the total amount of assets in the market and the LP's pending yield. To calculate the LP's pending yield, it must first determine the rate. This is accomplished by taking 3 of the 4 most recent snapshots and computing the average utilization. Based on this utilization, it will determine the appropriate rate and use that to accrue the yield.

The issue is that the accrued value returned from the `previewAccrueLiabilities` function can be different than what the actual calculation will return in the `accrueLiabilities` function. This is because the `takeSnapshot` function is called after `previewAccrueLiabilities`, but before `accrueLiabilities`. With a new snapshot introduced between these two functions the sets used to calculate utilization will be different.

An attacker can leverage this to mint at a discount, then immediately withdraw with a profit. This comes directly at the expense of the LP's.

All an attacker would need to do is order the utilization of the four most recent snapshots to be 2 low utilizations followed by 2 high. This can easily be done by performing large deposits or borrows right before the end of a snapshot, ensuring the utilization is the desired value.

Once the 4 snapshots are in place and the next snapshot period is open, an attacker will call `mint`, which will use the 3 oldest snapshots available, consisting of 2 low utilizations and 1 high. This results in a lower share price. This will be the share price that the attacker is entering at.

Then, when `_deposit` is called, a snapshot will be taken, and liabilities will be accrued based on the new set of snapshots, which now consist of 1 low utilization and 2 high. This results in a higher share price than what the attacker entered at. This difference is profit for the attacker that they can immediately withdraw from the market.

Recommendation

Call `takeSnapshot` in the `totalAssets` function before `previewAccrueLiabilities` is called. This will ensure the same snapshots are being used throughout the deposit process.

Resolution: Resolved. A fix was implemented in **b6b1ca5**.

3.1.3 Interest Rate can be Artificially Inflated

Severity: *High risk (Resolved)*

Context: [-o-1] : ***Market.sol****:311*

Description:

Because rate is based on the average utilization of 3 of the 4 most recent snapshots. And the snapshots themself are a representation of the state of the market at a single point in time. A malicious LP can inflate utilization of the snapshots to ensure maximum interest is paid.

This can be accomplished by any user but an LP would have the most to gain in this situation. The attacker would just need to wait until the last block of the current snapshot window and borrow up to the maximum and then make a dust deposit to trigger an update of the snapshot.

At the very next block the a new snapshot period will be open at which point the attacker can repay everything borrowed + a few seconds worth of interest. Withdraw the funds from their portfolio and deposit it into the vault.

Despite the utilization being low for all but a few seconds of the snapshot period. The interest rate will be based off an extremely high utilization since that is the last state the market was in from the perspective of that snapshot.

The attacker can repeat this at the end of every snapshot period to force borrowers into paying a much higher interest rate. All while collecting the yield as the LP.

Recommendation:

Consider implementing a time weighted average within each snapshot. This way if the snapshot period has a utilization of 10% for 99% of the snapshot period. The utilization cant be increased to 100% just because at the very last second the amount borrowed changed.

Resolution: Resolved. A fix was implemented in **b6b1ca5**.

3.2 Medium Risk

3.2.1 Errantly High Utilization Rate After deposit

Severity: *Medium risk (Resolved)*

Context: [-o-1] : ***Market.sol****:358*

Description:

Due to where the current snapshot is taken in the `_deposit` function snapshots wont represent the actual rate after the deposit. More specifically it is possible for the snapshot to state that during its time there was a high utilization where in fact the utilization was low.

When a user calls the `_deposit` function the utilization should decrease but it wont since there is no snapshot taken after the `totalAssets` is increased. Instead when rates are calculated the snapshot will state that the utilization is as high as it was prior to the deposit since this is the last snapshot that was taken. Because of this borrowers will wrongly pay a high interest rate.

Recommendation:

call `takeSnapShot` at the end of `_deposit` to ensure the rate is accurate. It will also be important to call `takeSnapshot` at the end of `borrow` and `repay`

Resolution: Resolved. A fix was implemented in **b6b1ca5**.

3.2.2 Slippage Check Wont Protect Users

Severity: *Medium risk (Resolved)*

Context: [-o-1] : ***Zapper.sol****:143*

Description:

The `zapOut` function will initially withdraw an amount from the caller's portfolio. When withdrawing assets, a proportional amount of shares will be burned for that user. To protect the user from burning excessive shares and sudden share value changes, a slippage check is implemented:

```
uint256 shares = withdraw(params.asset, params.amount);
if (shares < params.minSharesOut) {
    Errors.Zapper_SlippageExceeded(shares, params.minSharesOut);
}
```

The issue with this check is that it is designed to revert in cases where the amount of shares burned is less than the specified amount. However, if fewer shares are required than expected, that is actually beneficial since the user obtains their desired assets for less. On the other hand, if the share price suddenly drops and the withdrawal consumes many more shares than expected, there will be no slippage protection because the shares will be much greater than `params.minSharesOut`, failing the reverting condition:

```
if (shares < params.minSharesOut) {
    // revert
}
```

In summary, the current check will revert when the withdrawal is advantageous for the user but will not revert when the withdrawal is disadvantageous.

Recommendation:

Change the slippage check to ensure that shares are within a passed-in maximum amount:

```
- if (shares < params.minSharesOut) { revert Errors.Zapper_SlippageExceeded(shares,
    params.minSharesOut); }
+ require(shares <= params.maxSharesOut, Errors.Zapper_SlippageExceeded({ expected:
    params.maxSharesOut, actual: shares }));
```

Resolution: Resolved. The recommended fix was implemented in **7569a8a**.

3.2.3 Attacker can use Leftover Funds to Repay Liabilities

Severity: *Medium risk (Resolved)*

Context: [-o-1] : ***Zapper.sol****:150*

Description:

When the `zapOut` function is called, it allows users to specify a `repaymentAmount`. If non-zero, it will take the Zapper contract's balance of the market asset and use that to repay the user's liabilities. Any funds that will be used in the transaction are expected to be swept and sent to the treasury prior to any zaps or other actions occurring.

However, because the `repayLiabilities` function will always use the market's `asset`, a savvy user can exclude `asset` from their `tokensOut` array, which in turn will lead to the `asset` being ignored by the `sweep` function. With the funds being ignored, the `repayLiabilities` function will be able to use those funds to repay the attacker's debt.

Recommendation:

In the `zapOut` function, sweep the market's `asset` before executing any withdrawals, zaps, or repayments.

Resolution: Resolved. The recommended fix was implemented in **9cbd9ec**.

3.2.4 Allowlist Overlap Allows Unexpected Delegate Calls

Severity: *Medium risk (Resolved)*

Context: [-o-1] : ***Zapper.sol***:*286*

Description:

The `executeZaps` function will execute an array of zaps via delegate call as long as those zaps are a part of the `allowList`. Additionally, the `ERC4626Zapper` contract allows users to deposit to any market as long as it is on the `allowList`. The issue with the current design is that the zap targets and the markets share the same `allowList`.

Even though it is intended that `executeZaps` only calls zappers such as `ERC4626Zapper` and `WETHZapper`, users actually have access to all approved markets as well. This means they can use the market functions in the context of the Zapper contract. Given that the Zapper likely has large approvals from some of its users, leftover funds in the Zapper, and potential storage collisions, this unintended access to the market contracts should be eliminated.

Recommendation:

Maintain separate `allowList` entries for the zapper targets and the approved markets.

Resolution: Resolved. The recommended fix was implemented in **2bf90f3**.

3.2.5 Portfolio Assets are not Utilized in a Write Off

Severity: *Medium risk (Resolved)*

Context: [-o-2] : ***Market.sol***:*630*

Description:

The `writeOff` function is designed to eliminate bad debt from the system by utilizing reserves and, if necessary, `totalAssets`. Currently, the function includes a check to ensure the account is unhealthy before writing off debt.

However, there is a distinction between having an unhealthy account and having no assets in an account. An unhealthy account may still have assets in its portfolio to pay off some or all of its debts. In such cases, a liquidation or pulling from the portfolio first would be more appropriate.

Because there is no validation to confirm that a user's portfolio cannot cover any of the debt. A `writeOff` may occur where the unhealthy user retains funds in their portfolio while allowing the market's reserves and assets to cover their debts.

Recommendation:

Ensure the user's portfolio is empty and unable to pay off any debt before proceeding with a `writeOff`. If the portfolio has funds, use those to cover the debt before utilizing the market's reserves.

Resolution: Resolved.

Liquidator contract will be responsible for using portfolio assets first.

3.3 Low Risk

3.3.1 Reserves can be Temporarily Depleted

Severity: *Low risk (Resolved)*

Context: [-o-1] : ***Market.sol***:*415*

Description:

Any user can deplete the reserves of a market by performing a sequence of deposits, borrows, and withdrawals. This would be advantageous for users that want to borrow from a market that is already at a high utilization but has reserves available.

Take, for example, the following market setup:

- **Assets**: 100 USDT
- **Liabilities**: 100 USDT
- **Reserve**: 200 USDT
- **Actual Token Balance**: 200 ((100 - 100) + 200)

Alice takes a flash loan and deposits 200 USDT into the market.

- **Assets**: 300 USDT
- **Liabilities**: 100 USDT
- **Reserve**: 200 USDT
- **Actual Token Balance**: 400 ((300 - 100) + 200)

Alice then uses her own funds to supply and borrow 200 USDT:

- **Assets**: 300 USDT
- **Liabilities**: 300 USDT
- **Reserve**: 200 USDT
- **Actual Token Balance**: 200 ((300 - 300) + 200)

Alice withdraws the 200 USDT that was deposited:

- **Assets**: 100 USDT
- **Liabilities**: 300 USDT

- **Reserve**: 200 USDT
- **Actual Token Balance**: 0 ((100 - 300) + 200)

Finally, Alice repays the flash loan and is able to keep the 200 USDT she borrowed until she decides to repay it. During this time, the market's reserves are depleted, and LPs will not be able to withdraw.

Recommendation:

To partially mitigate this, consider replacing the current reserve method with a borrowable factor so that only a percentage of `totalAssets` can be borrowed at any point in time.

```
- require(required <= $.totalAssets, Errors.Market_InsufficentLiquidity(required, $.
    totalAssets));
+ require(required <= ($.totalAssets * borrowableFactor) / 1e18, Errors.
    Market_InsufficentLiquidity(required, $.totalAssets));
```

This will also help with capital efficiency since the current method of taking a portion of the LPs' yield and assigning it to the reserves will initially not be helpful as the amount will be too small. And eventually, it will become excessive as the amount grows too large.

Resolution: Resolved. A fix was implemented in **223c911**.

3.3.2 Debt Erased When Socialization is Insufficient

Severity: *Low risk (Resolved)*

Context: [-o-2] : ***Market.sol***:*650*

Description:

When the `writeOff` function is called, it uses a combination of `totalReserveAssets` and `totalAssets`, prioritizing `totalReserveAssets` since socializing debt among LPs is not desirable.

However, there is an unhandled edge case where, if it occurs, it could erase some or all of the user's debt without offsetting it with a corresponding decrease in reserves or total assets.

The reason this is possible is that the `openLiabilities` function assumes that the `amount` passed into the `writeOff` function was the amount actually written off. However, as shown below, there are two places where the `amount` to be paid is truncated:

```
uint256 provisions = Math.min(remaining, $.totalReserveAssets);

//...
//...

if (remaining > 0) {
      $.totalAssets = $.totalAssets - Math.min(remaining, $.totalAssets);
    }
```

If `remaining` is truncated in both places, the amount of debt written off will be less than the amount by which `liabilities` is reduced when reopening the user's position.

Example:

- Alice's liabilities: 100

- Write-off amount: 90
- `totalReserveAssets`: 10
- `totalAssets`: 20

First, all of `totalReserveAssets` would be used, lowering her liabilities to 90. Then, all of `totalAssets` would be used, lowering her liabilities further to 70.

At this point, 30 of Alice's `liabilities` have been written off. However, when the position is reopened, it will open with debt equal to `liabilities` minus `amount`, even though only a fraction of the `amount` was actually covered through reserves and LPs' assets.

Recommendation:

Only reduce liabilities by the amount that was actually written off. For example:

```
if (remaining > 0) {
+ uint256 socialized = Math.min(remaining, $.totalAssets);
+ $.totalAssets = $.totalAssets - socialized;
- $.totalAssets = $.totalAssets - Math.min(remaining, $.totalAssets);
}

if (liabilities > amount) {
    market.openLiabilities(
        user,
-   liabilities - amount.toUint128(),
+   liabilities - (provisions + socialized).toUint128(),
-   discountOf(account, liabilities - amount.toUint128())
+   discountOf(account, liabilities - (provisions + socialized).toUint128())
    );
}
```

Resolution: Resolved. A fix was implemented in **e24d869**.

3.3.3 Borrowers May Not Pay All Interest

Severity: *Low risk (Resolved)*

Context: [-o-2] : ***DynamicInterestRateModel.sol****:34*

Description:

When determining the utilization rate, the system first checks if there are any `totalAssets`. If so, it will return a utilization rate of zero, resulting in no interest being paid during that period.

The issue with this approach is that it is possible for a market to have zero `totalAssets` and a non-zero amount of `totalLiabilities`.

One scenario where this can occur is when the `writeOff` function is called. When debt is socialized, it reduces `totalAssets` by the amount needed to write off the debt. In rare cases, this can result in `totalAssets` being reduced to zero while `totalLiabilities` is non-zero. , leading to a situation where borrowers pay no interest on their outstanding debt.

Recommendation:

When calculating the utilization rate, if `totalAssets` is zero and `totalLiabilities` is non-zero, set the utilization rate to `1e18`.

Resolution: Resolved. The recommended fix was implemented in **48352c8**.

3.3.4 Arbitrary Token can be used in FlashLoan

Severity: *Low risk (Resolved)*

Context: [1] : ***Market.sol****:587*

Description:

The `flashLoan` function allows users to take an arbitrary amount of tokens from the `market` contract. When this function is called via the `zapper`, it only provides the option of taking the market's `asset`. This makes sense, as the `market` is intended to have a single asset used for both LPs and borrowers. However, the `flashLoan` function itself has no requirement on what token can be flash loaned, which means any arbitrary token that happens to be in the market can be utilized.

Recommendation:

Consider removing the `token` parameter and only use the market's `asset` for the `flashLoan` function.

Resolution: Resolved. The recommended fix was implemented in **052ea01**.

3.3.5 Users can Escape Debt Socialization

Severity: *Low risk (Acknowledged)*

Context: [2] : ***Market.sol****:619*

Description:

When debt is written off via the `writeOff` function, there is a possibility that the `totalReserveAssets` will not be sufficient. In such cases, `totalAssets` will be used. When this happens, it causes an immediate change in the LPs' share price. With assets decreasing while the total number of shares remains the same, the share price will drop, negatively impacting all LPs.

However, any LP that is aware of this situation can monitor for such transactions. If they identify this scenario, the LP can front-run the `writeOff` call and withdraw their shares. As long as there are sufficient assets to withdraw, the LP can escape the debt socialization. By doing so, they transfer their share of the burden to the remaining LPs.

The user can then immediately deposit again, ultimately ending up with more shares than they started with.

Recommendation:

To prevent this exploit, consider executing the `writeOff` function through a private mempool. This would obscure the transaction from being publicly visible in advance, mitigating the ability for LPs to front-run the call.

Resolution: Acknowledged by the team.

3.3.6 Flash Loan Missing Balance Check

Severity: *Low risk (Resolved)*

Context: [1] : ***Market.sol***:*584*

Description:

The `flashloan` function allows users to take an arbitrary amount of tokens from the `market` contract, and as long as the user can transfer exactly the amount borrowed back to the contract at the end of the function, the user can freely use these funds however they wish during the call.

For most tokens, this will work as expected. However, tokens that have fees or other mechanics that alter the amount received can lead to the `market` contract only receiving a percentage of what was lent out. If this were to be repeated, it would eventually deplete the `market` of its assets.

Recommendation:

Consider adding a balance check to the `flashloan` function where the balance after must be greater than or equal to the balance before.

Resolution: Resolved. The recommended fix was implemented in **052ea01**.

3.3.7 Reserve Fee Change can Cause Loss of Yield

Severity: *Low risk (Resolved)*

Context: [2] : ***Market.sol***:*220*

Description:

The `setReserveFee` function is used to change the percentage of the LPs' yield allocated to reserves.

Currently, the `accrue` function is not called before the reserve fee is updated. As a result, any pending yield that has not yet been accrued will be distributed based on the new rate. When the reserve fee is increased, this results in a loss of yield for LPs, as a larger portion of the previously expected yield will be redirected to reserves.

Recommendation:

Call the `accrue` function before updating the reserve fee in `setReserveFee` to ensure that pending yield is distributed using the old rate, preserving fairness for LPs.

Resolution: Resolved. The recommended fix was implemented in **64e2061**.