SHERLOCK

Security Review For Neverland

Collaborative Audit Prepared For: **Neverland**
Lead Security Expert(s): **0x37**
pkqs90
Date Audited: **June 23 - June 27, 2026**

Introduction

Neverland is a next-generation, Monad-native lending protocol fusing the most secure, battle-tested lending technology with proprietary veTokenomics that allow governance to direct protocol revenue and steer the protocol. Featuring self-repaying loans and automation, flexible and tradable governance locks, and multiple lending modes, each designed to maximize capital efficiency and long-term alignment.

Scope

Repository: Neverland-Money/neverland-lending

Audited Commit: ec11951ee0957d134f523cde85b31a19c4285378

Final Commit: df940b453157037148fc5101fc75c1708fb03011

Files:

- contracts/dependencies/openzeppelin/contracts/ECDSA.sol
- contracts/protocol/libraries/helpers/TokenMath.sol
- contracts/protocol/libraries/logic/BorrowLogic.sol
- contracts/protocol/libraries/logic/BridgeLogic.sol
- contracts/protocol/libraries/logic/ConfiguratorLogic.sol
- contracts/protocol/libraries/logic/EModeLogic.sol
- contracts/protocol/libraries/logic/FlashLoanLogic.sol
- contracts/protocol/libraries/logic/GenericLogic.sol
- contracts/protocol/libraries/logic/LiquidationLogic.sol
- contracts/protocol/libraries/logic/PoolLogic.sol
- contracts/protocol/libraries/logic/ReserveLogic.sol
- contracts/protocol/libraries/logic/SupplyLogic.sol
- contracts/protocol/libraries/logic/ValidationLogic.sol
- contracts/protocol/libraries/math/PercentageMath.sol
- contracts/protocol/libraries/math/WadRayMath.sol
- contracts/protocol/pool/PoolConfigurator.sol
- contracts/protocol/pool/Pool.sol
- contracts/protocol/tokenization/AToken.sol
- contracts/protocol/tokenization/base/DebtTokenBase.sol
- contracts/protocol/tokenization/base/PriceEmitter.sol

- contracts/protocol/tokenization/VariableDebtToken.sol

Final Commit Hash

df940b4531570371 48fc5101fc75c1708fb03011

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	0	6

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue L-1: Flash loan premiums can over-credit suppliers [RESOLVED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/22

Vulnerability Detail

`FlashLoanLogic._handleFlashLoanRepayment()` distributes the LP share of a flash loan fee by increasing the reserve `liquidityIndex` through `ReserveLogic.cumulateToLiquidityIndex()`.

In `FlashLoanLogic.sol`, the denominator is built from the visible `aToken` supply plus pending treasury liquidity:

```
reserveCache.nextLiquidityIndex = reserve.cumulateToLiquidityIndex(
  IERC20(reserveCache.aTokenAddress).totalSupply() +
    uint256(reserve.accruedToTreasury).getATokenBalance(reserveCache.nextLiquidityI
    ↪  ndex),
  premiumToLP
);
```

Both terms are rounded down. `AToken.totalSupply()` returns the scaled supply converted with `getATokenBalance()`, and `TokenMath.getATokenBalance()` uses `rayMulFloor()`:

```
function getATokenBalance(
  uint256 scaledAmount,
  uint256 liquidityIndex
) internal pure returns (uint256) {
  return scaledAmount.rayMulFloor(liquidityIndex);
}
```

`ReserveLogic.cumulateToLiquidityIndex()` then divides `premiumToLP` by this rounded-down denominator:

```
uint256 result = (amount.wadToRay().rayDivFloor(totalLiquidity.wadToRay()) +
↪  WadRayMath.RAY)
  .rayMulFloor(reserve.liquidityIndex);
```

This is the wrong rounding direction for a denominator. If `totalLiquidity` is rounded down, the premium per unit of liquidity is rounded up, causing `liquidityIndex` to increase more than it should.

Impact

Suppliers can be credited more underlying through `liquidityIndex` than the `premiumToLP` actually paid by the flash loan borrower.

Recommendation

We should round up when calculating the denominator for total liquidity here.

Issue L-2: Errors and reverts can be improved [RESOLVED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/23

Vulnerability Detail

Several newly introduced validation paths use either misleading protocol errors or implicit Solidity underflow panics.

In `SupplyLogic.executeWithdraw()`, the recipient address check reverts with `Errors.INVALID_AMOUNT`, even though the invalid value is `params.to`:

```
require(params.to != reserveCache.aTokenAddress, Errors.INVALID_AMOUNT);
```

In `ValidationLogic.validateSupply()`, the `onBehalfOf` address check also reverts with `Errors.INVALID_AMOUNT`:

```
require(onBehalfOf != reserveCache.aTokenAddress, Errors.INVALID_AMOUNT);
```

In `AToken.transferFrom()`, insufficient allowance is checked through an unused subtraction expression, causing a generic arithmetic panic instead of a clear protocol error:

```
uint256 currentAllowance = this.allowance(sender, _msgSender());
currentAllowance - amount;
```

In `VariableDebtToken._decreaseBorrowAllowanceCapped()`, insufficient borrow delegation is checked the same way:

```
uint256 oldBorrowAllowance = _borrowAllowances[delegator][delegatee];
oldBorrowAllowance - amount;
```

Impact

No real impact, recommend to use best dev practice to revert with better errors.

Issue L-3: Flash loan price action codes are defined but never emitted [RESOLVED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/24

Vulnerability Detail

`PriceEmitter` defines action codes for flash loan observations:

```
uint8 internal constant ACTION_FLASHLOAN = 4;
uint8 internal constant ACTION_FLASHLOAN_SIMPLE = 5;
```

However, neither code is used anywhere in the protocol. `AToken` and `VariableDebtToken` use `emitPrice()` for supply, withdraw, transfer, liquidation, borrow, and repay flows, but flash loan execution does not emit `PriceObserved`.

In `FlashLoanLogic.executeFlashLoan()` and `FlashLoanLogic.executeFlashLoanSimple()`, the protocol only emits the regular `FlashLoan` event. No `emitPrice()` call or equivalent `PriceObserved` emission is performed for either flash loan path.

Recommendation

Either emit `PriceObserved` during `executeFlashLoan()` and `executeFlashLoanSimple()` using the existing flash loan action codes, or remove the unused constants if flash loan price observations are intentionally unsupported.

Issue L-4: permit can be frontrun to cause temporary dos [RESOLVED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/25

Summary

permit can be frontrun to cause temporary dos

Vulnerability Detail

In Pool contract, users can merge their permit and supply into one transaction via the function `supplyWithPermit`.

The problem here is that if this permit operation is frontrun by someone, this whole transaction will be reverted.

And the same issue exists in the function `repayWithPermit`.

Impact

The impact is low.

1. Cause the transaction reverted.
2. If users want to provide more collateral to increase his position's healthy. This temporary dos may cause that this position is liquidated because of the delayed increasing collateral value.

Code Snippet

https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/blob/6548c812075fd64f6af98bdc0040fd1382d83145/neverland-lending/contracts/protocol/pool/Pool.sol#L130-L160

Tool Used

Manual Review

Recommendation

Suggest to add try/catch for the permit operation. Then even if this permit is frontrun by someone, this operation can always succeed.

Issue L-5: Missing bad debt process mechanism [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/26

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Missing bad debt process mechanism

Vulnerability Detail

In liquidation, if some collateral price drops very quickly, it may cause that liquidators seize all collateral and some remaining debt left in this borrower's position. These debt will keep accrue interest.

When bad debt occurs, early lenders can withdraw their assets as expected, while the remaining (last) users may fail to complete their withdrawals as anticipated.

```
function liquidationCall(
  address collateralAsset,
  address debtAsset,
  address user,
  uint256 debtToCover,
  bool receiveAToken
) public virtual override {
  LiquidationLogic.executeLiquidationCall(
    _reserves,
    _reservesList,
    _usersConfig,
    _eModeCategories,
    DataTypes.ExecuteLiquidationCallParams({
      reservesCount: _reservesCount,
      debtToCover: debtToCover,
      collateralAsset: collateralAsset,
      debtAsset: debtAsset,
      user: user,
      receiveAToken: receiveAToken,
      priceOracle: ADDRESSES_PROVIDER.getPriceOracle(),
      userEModeCategory: _usersEModeCategory[user],
      priceOracleSentinel: ADDRESSES_PROVIDER.getPriceOracleSentinel()
    })
  );
}
```

Impact

The last lender may take all the loss from the bad debt.

Code Snippet

https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/blob/6548c812075fd64f6af98bdc0040fd1382d83145/neverland-lending/contracts/protocol/pool/Pool.sol#L326-L350

Tool Used

Manual Review

Recommendation

That depends on the design. If the team plans to cover or absorb this debt, can refer to the aave's latest implementation.

Issue L-6: Missing accrue interest before the setReserveFactor [RESOLVED]

Source: https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/issues/27

Summary

Missing accrue interest before the setReserveFactor

Vulnerability Detail

The function `setReserveFactor` is used to update the reserve factor by the pool admin. The reserve factor is used to calculate the protocol fee from the accrued interest.

The problem here is that in function `setReserveFactor`, the system misses to update and accrue the previous time slot's interest. When this reserve factor is updated, the previous time slot's interest and protocol fee will be calculated with the updated reserve factor.

```
function setReserveFactor(
  address asset,
  uint256 newReserveFactor
) external override onlyRiskOrPoolAdmins {
  require(newReserveFactor <= PercentageMath.PERCENTAGE_FACTOR,
  ↪  Errors.INVALID_RESERVE_FACTOR);
  DataTypes.ReserveConfigurationMap memory currentConfig =
  ↪  _pool.getConfiguration(asset);
  uint256 oldReserveFactor = currentConfig.getReserveFactor();
  currentConfig.setReserveFactor(newReserveFactor);
  _pool.setConfiguration(asset, currentConfig);
  emit ReserveFactorChanged(asset, oldReserveFactor, newReserveFactor);
}
```

Impact

The previous time slot's protocol fee may be calculated with the updated reserve factor.

Code Snippet

https://github.com/sherlock-audit/2026-06-neverland-lending-pool-audit-june-23rd-2026/blob/6548c812075fd64f6af98bdc0040fd1382d83145/neverland-lending/contracts/protocol/pool/PoolConfigurator.sol#L234-L244

Tool Used

Manual Review

Recommendation

Accrue the previous time slot's interest before the reserve factor is updated.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.