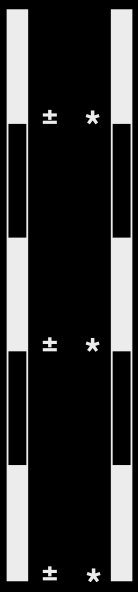
VERILOG X FUZZLAND
JOINT AUDIT

2024

BENDDAO V2: SECURITY REVIEW REPORT



Jun 4, 2024

Contents

1 | Introduction

Figure 1.1: Bend V2 Report Cover

This report presents our engineering engagement with the BendDAO team on their new version of the decentralized non-custodial NFT liquidity and lending protocol, the BendDAO V2 Protocol.

Project Name	**Bend V2 Protocol**
Repository Link	https://github.com/BendDAO/bend-v2
First Commit Hash	First: 0688b99;
Final Commit Hash	Final: 819255b;
Language	Solidity
Chain	Merlin

2 | About Verilog Solutions

Founded by a group of cryptography researchers and smart contract engineers in North America, Verilog Solutions elevates the security standards for Web3 ecosystems by being a full-stack Web3 security firm covering smart contract security, consensus security, and operational security for Web3 projects.
Verilog Solutions team works closely with major ecosystems and Web3 projects and applies a quality above quantity approach with a continuous security model. Verilog Solutions onboards the best and most innovative projects and provides the best-in-class advisory services on security needs, including on-chain and off-chain components.

3 | Service Scope

3.1 | Service Stages

Our auditing service includes the following two stages:

- Smart Contract Auditing Service

3.1.1 | Smart Contract Auditing Service

The Verilog Solutions team analyzed the entire project using a detailed-oriented approach to capture the fundamental logic and suggested improvements to the existing code. Details can be found under Findings And Improvement Suggestions.

3.2 | Methodology

- **Code Assessment**
  - We evaluate the overall quality of the code and comments as well as the architecture of the repository.
  - We help the project dev team improve the overall quality of the repository by providing suggestions on refactorization to follow the best practices of Web3 software engineering.
- **Code Logic Analysis**
  - We dive into the data structures and algorithms in the repository and provide suggestions to improve the data structures and algorithms for the lower time and space complexities.
  - We analyze the hierarchy among multiple modules and the relations among the source code files in the repository and provide suggestions to improve the code architecture with better readability, reusability, and extensibility.

3.3 | Audit Scope

Our auditing for Bend V2 covered the Solidity smart contracts under the folder 'src' in the repository (https://github.com/BendDAO/bend-v2) with commit hash **0688b99**.

4 | Project Summary

The BendDAO V2 protocol is a decentralized peer-to-pool-based NFT liquidity protocol. Borrowers can borrow tokens instantly through the lending pool using NFTs and other ERC20 tokens as collateral. Depositors provide ERC20 tokens as liquidity to the lending pool to earn interest. As opposed to the first version of the protocol where users could only deposit NFTs as collateral, the BendDAO V2 protocol offers more flexibility with the assets that can be deposited and borrowed.
Below is the protocol architecture:

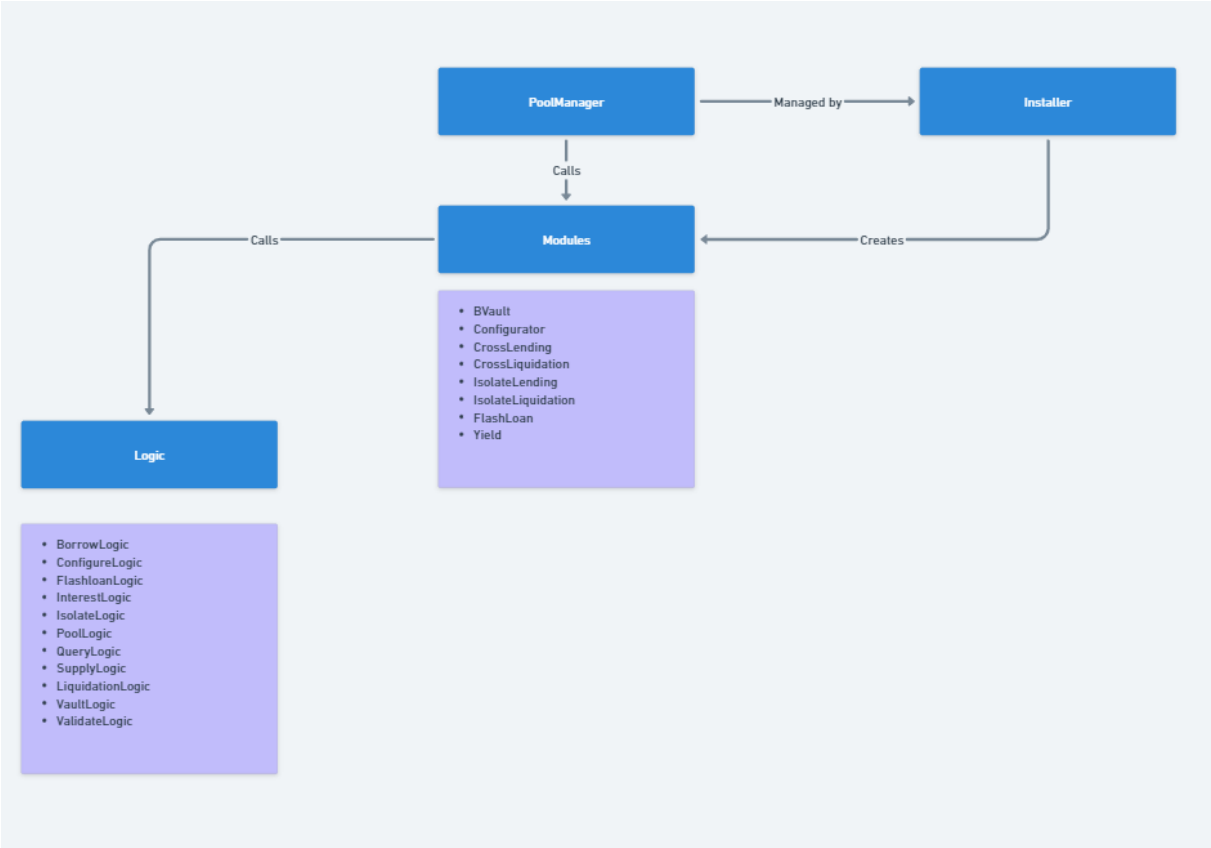


Figure 4.1: BendDAO V2 Architecture

The BendDAO V2 protocol is a collection of smart contracts connected together with a module system. Each module handles specific areas of the protocol, the main modules are described below:

- Configurator

  Allows privileged roles to create and delete pools, groups, assets and set pauses to certain actions.

- BVault

  Contains the logic to deposit and withdraw ERC20 and ERC721 assets.

- Cross Lending

  Contains the logic to borrow when the users deposit ERC20 and ERC721 tokens as collateral.

- Isolate Lending

  Contains the logic to borrow when the users deposit ERC721 tokens as collateral.

- Cross Liquidation

  Contains the logic for liquidation when the users deposit ERC20 and ERC721 tokens as collateral.

- Isolate Liquidation

  Contains the logic for liquidation when the users deposit ERC721 tokens as collateral.

- Flash Loan

  Contains the logic to execute flash loans.

- Pool Lens

  Contains the getter functions.

- Yield

  Contains the logic to allow users who borrowed to stake those tokens in Lido, Etherfi and the DAI savings pool.

5 | Findings and Improvement Suggestions

Severity	Total	Acknowledged	Resolved
High	3	3	3
Medium	5	5	5
Low	6	6	4
Informational	6	6	6

5.1 | High

5.1.1 | Self-liquidation leads to incorrect calculation of `totalCrossSupply`

Severity	High
Source	src/PoolManager.sol#L331;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  The `executeCrossLiquidateERC20()` function handles the liquidation logic. Through the analysis of the test case `test_Should_LiquidateUSDTself`, it was found that when the liquidator and the borrower are the same entity, there is a discrepancy between the actual and expected values of `totalCrossSupply` after `executeCrossLiquidateERC20()` liquidation.

- **Exploit Scenario**

  - Before liquidation, the expected `totalCrossSupply` is 0, but it actually is 10,000,000,000,000,000,000;
  - Following the initiation of liquidation, there is a significant difference between the expected and actual values of `totalCrossBorrow`, with the expected being 2,866,274,286 and the actual being 18,558,267,614;
  - After liquidation, the expected and actual values of `walletBalance` also do not match, with one checkpoint showing a discrepancy of 10,000,000,000,000,000,000s.

- **Recommendations**
  Check the liquidation logic to ensure accurate handling of asset status updates when the liquidator and borrower are the same address.

- **Results**
  Resolved in commit 7d6f7be.

  The liquidator can not be the borrower.

5.1.2 | The NFT on auction cannot be redeemed if the borrower is the first bidder.

Severity	High
Source	src/PoolManager.sol#L450;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

■ **Description**

There is a logical vulnerability in the `executeIsolateAuction()` function that prevents the redemption of NFTs when the borrower is the `firstBidder`.

In the `executeIsolateAuction()` function, if the borrower is the `firstBidder`, the code transfers the bid fine from the borrower to the `firstBidder` using the `erc20TransferBetweenWallets()` function. However, the `erc20TransferBetweenWallets()` function does not properly handle the transfer of tokens from the borrower to the `firstBidder`. As a result, the bid fine is not transferred correctly, and the redemption of NFTs is blocked.

```
   // src\libraries\logic\IsolateLogic.sol
if (loanData.firstBidder != address(0)) {
  // transfer bid fine from borrower to the first bidder
  VaultLogic.erc20TransferBetweenWallets(
    params.asset,
    msg.sender,
    loanData.firstBidder,
    vars.bidFines[vars.nidx]
  );
}
//  src\libraries\logic\VaultLogic.sol
function erc20TransferBetweenWallets(address asset, address from,
address to, uint amount) internal {
  require(to != address(0), Errors.INVALID_TO_ADDRESS);

  uint256 userSizeBefore = IERC20Upgradeable(asset).balanceOf(to);

  IERC20Upgradeable(asset).safeTransferFrom(from, to, amount);

  uint userSizeAfter = IERC20Upgradeable(asset).balanceOf(to);
  require(userSizeAfter == (userSizeBefore + amount), Errors.INVALID_TRANSFER_AMOUNT);
}
```

■ **Exploit Scenario**

The following POC can be taken into consideration:

```
// test\integration\TestIntIsolateRedeem.t.sol
function test_Should_Auction_Redeem_Same_WETH() public {
  TestCaseLocalVars memory testVars;

  // deposit
  prepareWETH(tsDepositor1);
  uint256[] memory tokenIds = prepareIsolateBAYC(tsBorrower1);

  // borrow
  prepareBorrow(tsBorrower1, address(tsBAYC), tokenIds, address(tsWETH));

  // make some interest
  advanceTimes(365 days);

  // drop down nft price
```

```
    actionSetNftPrice(address(tsBAYC), 5000);

    // auction
    testVars.loanDataBefore1 = getIsolateLoanData(tsCommonPoolId,
    address(tsBAYC), tokenIds);
    testVars.bidAmounts1 = new uint256[](tokenIds.length);
    for (uint256 i = 0; i < tokenIds.length; i++) {
      testVars.bidAmounts1[i] = testVars.loanDataBefore1[i].borrowAmount;
      testVars.totalBidAmount1 += testVars.bidAmounts1[i];
    }
    tsBorrower1.approveERC20(address(tsWETH), type(uint256).max);
    tsBorrower1.isolateAuction(tsCommonPoolId, address(tsBAYC), tokenIds,
    address(tsWETH), testVars.bidAmounts1);
    testVars.txAuctionTimestamp1 = block.timestamp;

    // auction2
    testVars.loanDataAfter1 = getIsolateLoanData(tsCommonPoolId, address(tsBAYC),
    tokenIds);
    testVars.bidAmounts2 = new uint256[](tokenIds.length);
    for (uint256 i = 0; i < tokenIds.length; i++) {
      testVars.bidAmounts2[i] = (testVars.loanDataAfter1[i].bidAmount * 1011) / 1000;
      // plus 1.1%
      testVars.totalBidAmount2 += testVars.bidAmounts2[i];
    }
    tsLiquidator2.approveERC20(address(tsWETH), type(uint256).max);
    tsLiquidator2.isolateAuction(tsCommonPoolId, address(tsBAYC), tokenIds,
    address(tsWETH), testVars.bidAmounts2);

    // redeem
    // tsBorrower1.approveERC20(address(tsWETH), type(uint256).max);
    tsHEVM.expectRevert(bytes(Errors.INVALID_TRANSFER_AMOUNT));
    tsBorrower1.isolateRedeem(tsCommonPoolId, address(tsBAYC), tokenIds, address(tsWETH));
    testVars.txAuctionTimestamp = block.timestamp;
  }
```

- **Recommendations**
  Ensure that the liquidator can not be the borrower in the `executeIsolateAuction()` function.

- **Results**
  Resolved in commit 7d6f7be.

  The liquidator can not be the borrower.

5.1.3 | `createYieldAccount()` in contract `YieldEthStakingEtherfi` and `YieldEthStakingLido` should be using `msg.sender` directly

Severity	High
Source	src/yield/YieldStakingBase.sol#L162;
Commit	5c7bbaa;
Status	Resolved in commit 8ca4301;

■ **Description**

The `createYieldAccount()` function in the contracts `YieldEthStakingEtherfi` and `YieldEthStakingLido` should using `msg.sender` directly instead of passing in a variable `user`.

If the user variable is not the same as the `msg.sender` address, the yield account will be created for `user` rather than `msg.sender`. The `yieldAccount.safeApprove(address(eETH)` will fail as the yield account is not created for `msg.sender` yet.

```
// YieldEthStakingEtherfi.sol and YieldEthStakingLido.sol
  function createYieldAccount(address user) public virtual override
  returns (address) {
    super.createYieldAccount(user);

    IYieldAccount yieldAccount = IYieldAccount(yieldAccounts[msg.sender]);
    yieldAccount.safeApprove(address(stETH), address(unstETH), type(uint256).max);

    return address(yieldAccount);
  }
```

■ **Exploit Scenario**

- □ The `createYieldAccount()` function gets called with the incorrect `user` parameter;
- □ A user starts staking with that account;
- □ The functionality won't work as expected as the input address for the creation was incorrect.

■ **Recommendations**

Remove the `user` arguments and use `msg.sender` directly.

```
// YieldEthStakingEtherfi.sol and YieldEthStakingLido.sol
  function createYieldAccount() public virtual override returns (address) {
    super.createYieldAccount(msg.sender);

    IYieldAccount yieldAccount = IYieldAccount(yieldAccounts[msg.sender]);
    yieldAccount.safeApprove(address(stETH), address(unstETH), type(uint256).max);

    return address(yieldAccount);
  }
```

■ **Results**

Resolved in commit 8ca4301.

Kept the `user` parameter but fixed the `IYieldAccount(yieldAccounts[msg.sender]) => IYieldAccount(yieldAccounts[user])`

5.2 | Medium

5.2.1 | Use of deprecated Chainlink function `latestAnswer`

Severity	Medium
Source	src/PriceOracle.sol#L107;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  According to Chainlink's documentation, the `latestAnswer()` function is deprecated. This function does not error if no answer has been reached but returns 0:

```
function getAssetPriceFromChainlink(address asset) public view returns (uint256) {
    uint256 price;

    AggregatorV2V3Interface sourceAgg = assetChainlinkAggregators[asset];
    require(address(sourceAgg) != address(0), Errors.ASSET_AGGREGATOR_NOT_EXIST);

    int256 priceFromAgg = sourceAgg.latestAnswer();
    //@audit Use of deprecated Chainlink function latestAnswer
    price = uint256(priceFromAgg);

    require(price > 0, Errors.ASSET_PRICE_IS_ZERO);
    return price;
}
```

- **Exploit Scenario**
  - The `getAssetPrice()` function gets called to get the price of a token;
  - The answer is not reached;
  - The return value is zero since the `latestAnswer()` function doesn't perform a check on the stale price.
- **Recommendations**
  Use the `latestRoundData()` function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is incomplete, for example.
- **Results**
  Resolved in commit 7d6f7be.

  The `latestRoundData` is used and the time is checked.

5.2.2 | Native tokens could get stuck in the pool

Severity	Medium
Source	src/PoolManager.sol#L220;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**

  Since the `depositERC20()` function is payable, when users deposit non-native tokens and set the `msg.value` to be non-zero, the native tokens sent to the pool will be trapped.

```
function depositERC20(uint32 poolId, address asset, uint256 amount)
public payable whenNotPaused nonReentrant {
  DataTypes.PoolStorage storage ps = StorageSlot.getPoolStorage();
  if (asset == Constants.NATIVE_TOKEN_ADDRESS) {
    asset = ps.wrappedNativeToken;
    amount = msg.value;
    VaultLogic.wrapNativeTokenInWallet(asset, msg.sender, amount);
  }

  SupplyLogic.executeDepositERC20(
    InputTypes.ExecuteDepositERC20Params({poolId: poolId, asset: asset, amount: amount})
  );
}
```

- **Exploit Scenario**

  - Alice deposits non-native tokens using the `depositERC20()` function;
  - Alice mistakenly sets the `msg.value` to a value greater than zero;
  - The native token gets transferred into the contract and is stuck.

- **Recommendations**

  Require the the `msg.value` to be zero when accepting deposits of non-native tokens

- **Results**

  Resolved in commit 7d6f7be.

  A check on the `msg.value` has been added.

5.2.3 | Unchecked return values

Severity	Medium
Source	src/libraries/logic/ConfigureLogic.sol#L545; src/libraries/logic/ConfigureLogic.sol#L552;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  The `executeSetAssetYieldEnable()` function adds or removes a yield group using the `add()` and `remove()` functions from the `EnumerableSetUpgradeable` contract. Since these functions return a boolean value, this value should be checked before performing any additional change.
- **Exploit Scenario**
  - The `executeSetAssetYieldEnable()` function gets called to remove a new asset;
  - The return value when removing is false but the transaction goes through;
  - Admin thinks he has removed the asset but users can still make transactions with the asset.
- **Recommendations**
  Check the boolean return value the way it is performed in other functions such as the `executeSetPoolYieldEnable()` function.
- **Results**
  Resolved in commit 7d6f7be.

  A check has been added on the return values.

5.2.4 | Flashloan is not disabled for NFT on auction

Severity	Medium
Source	src/libraries/logic/FlashLoanLogic.sol#L20;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  The NFT assets that are on auction can still be flash loaned. This means that users can momentarily get access to their NFTs while other users are already making bids on them.

- **Exploit Scenario**

  - Alice has her NFT on auction;
  - Alice calls the `executeFlashLoanERC721()` function to get her NFT
  - The transaction will go through even though the flash loan should not be available.

- **Recommendations**
  Check that the loan status is active in the `executeFlashLoanERC721()` function.

```
if (loanData.loanStatus != 0) {
  require(loanData.loanStatus == Constants.LOAN_STATUS_ACTIVE,
  Errors.INVALID_LOAN_STATUS);
}
```

- **Results**
  Resolved in commit 7d6f7be.

  Checks have been added to the isolate loan status.

5.2.5 | Missing sender address check in `receive()` may lead to locked ETH

Severity	Medium
Source	src/PoolManager.sol; src/base/Proxy.sol; src/yield/YieldAccount.sol; src/yield/etherfi/YieldEthStakingEtherfi.sol; src/yield/lido/YieldEthStakingLido.sol;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  The following contracts have `receive()` functions without performing any additional checks. That means that anyone can transfer native tokens into the contract.

```
src/PoolManager.sol:
77
78: receive() external payable {}
79 }
src/base/Proxy.sol:
12
13: receive() external payable {}
14 src/yield/YieldAccount.sol:
78
79: receive() external payable {}
80 } src/yield/etherfi/YieldEthStakingEtherfi.sol:
134
135: receive() external payable {}
136 } src/yield/lido/YieldEthStakingLido.sol:
 134
135: receive() external payable {}
136 }
```

  We recommend adding checks to the addresses that can transfer native tokens to the contract.

- **Exploit Scenario**

  - Alice mistakenly transfers the native token to the `PoolManager` contract;
  - She is not able to recover the tokens even by contacting the BendDAO team.

- **Recommendations**
  Add an address check in the `receive()` function.

- **Results**
  Resolved in commit be2b2e9.

  An `emergencyEtherTransfer ()` function was added.

5.3 | Low

5.3.1 | Lack of event emission for critical operations

Severity	Low
Source	src/libraries/logic/ConfigureLogic.sol#L612;
Commit	0688b99;
Status	Resolved in commit 7d6f7be;

- **Description**
  Events are vital aids in monitoring contracts and detecting suspicious behavior. The `executeSetStakerYieldCap()` function performs state changes, therefore they should consider the emission of an event.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Add an event to the `executeSetStakerYieldCap()` function.
- **Results**
  Resolved in commit 7d6f7be.

  The event was added.

5.3.2 | Unchecked transfers

Severity	Low
Source	src/libraries/logic/VaultLogic.sol#L758; src/libraries/logic/VaultLogic.sol#L764;
Commit	2360625;
Status	Resolved in commit 7d6f7be;

- **Description**
  Calls to the `transferFrom()` function of the `WETH` contract don't check the return values. They should check it to make sure that the transfers indeed take place.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Add a `require()` statement to check that the transfers actually go through.
- **Results**
  Resolved in commit 7d6f7be.

  The checks were added.

5.3.3 | Unused functions

Severity	Low
Source	src/libraries/logic/PoolLogic.sol#36; src/libraries/logic/VaultLogic.sol#753; src/libraries/logic/VaultLogic.sol#518; src/libraries/logic/VaultLogic.sol#523;
Commit	7d9cd57;
Status	Resolved in commit 7d6f7be;

- **Description**
  The `checkCallerIsOracleAdmin()` checks that the sender is the Oracle admin. Just like the `checkCallerIsPoolAdmin()` and `checkCallerIsEmergencyAdmin()` functions are used within the `PoolManager` contract, this function should be used in the `PriceOracle` contract to verify that the sender is the admin.

  Additionally, the `safeTransferNativeToken()`, `erc721ResetTokenLockFlag()`, and `erc721SetTokenLockFlag()` functions are internal functions that are not used.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Use the `checkCallerIsOracleAdmin()` function in the `PriceOracle` contract.

- **Results**
  Resolved in commit 7d6f7be.

  The functions were removed.

5.3.4 | Upgradeable contract is missing a `/_gap[50]` storage variable to allow for new storage variables in later versions

Severity	Low
Source	Global;
Commit	7d9cd57;
Status	Resolved in commit 7d6f7be;

- **Description**
  While some contracts may not currently be sub-classed, adding the `/_gap[50]` variable now protects against forgetting to add it in the future.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Add a `/_gap[50]` storage variable.
- **Results**
  Resolved in commit 7d6f7be.

  The suggestion was implemented.

5.3.5 | The `getAccountTotalYield()` function has potential manipulation

Severity	Low
Source	src/yield/etherfi/YieldEthStakingEtherfi.sol#120;
Commit	7d9cd57;
Status	Acknowledged;

■ **Description**

Taking `YieldEthStakingEtherfi` as an example, the `getAccountTotalYield()` function is used to query the `eETH` token balance of the specified account. This function calls the `balanceOf` function of the `eETH` contract and returns the `eETH` balance of the target account.

```
function getAccountTotalYield(address account) public
view override returns (uint256) {
    return eETH.balanceOf(account);
}
```

This function directly relies on the `balanceOf` function of the `eETH` contract to obtain the balance of the specified account. Therefore, the return value of this function depends entirely on the balance data in the `eETH` contract.

POC

```
 test/yield/YieldEthStakingEtherfi.t.sol
nction test_Should_unstake1() public {
  YieldTestVars memory testVars;

  prepareWETH(tsDepositor1);

  uint256[] memory tokenIds = prepareIsolateBAYC(tsBorrower1);

  uint256 stakeAmount =
  tsYieldEthStakingEtherfi.getNftValueInUnderlyingAsset(address(tsBAYC));
  stakeAmount = (stakeAmount * 80) / 100;

  tsHEVM.prank(address(tsBorrower1));
  address yieldAccount =
  tsYieldEthStakingEtherfi.createYieldAccount(address(tsBorrower1));

  tsHEVM.prank(address(tsBorrower1));
  tsYieldEthStakingEtherfi.stake(tsCommonPoolId, address(tsBAYC), tokenIds[0],
  stakeAmount);

  uint256 deltaAmount = (stakeAmount * 35) / 1000;
  tsEtherfiLiquidityPool.rebase{value: deltaAmount}(yieldAccount);


  tsHEVM.prank(address(tsFaucet));
  tsEETH.mint(address(yieldAccount),11100);


  (uint256 yieldAmount, ) =
  tsYieldEthStakingEtherfi.getNftYieldInUnderlyingAsset(address(tsBAYC), tokenIds[0]);
  testEquality(yieldAmount, (stakeAmount + deltaAmount), 'yieldAmount not eq');

  tsHEVM.prank(address(tsBorrower1));
  tsYieldEthStakingEtherfi.unstake(tsCommonPoolId, address(tsBAYC), tokenIds[0], 0);
```

```
    (testVars.poolId, testVars.state, testVars.debtAmount, testVars.yieldAmount) =
    tsYieldEthStakingEtherfi
      .getNftStakeData(address(tsBAYC), tokenIds[0]);
    assertEq(testVars.state, Constants.YIELD_STATUS_UNSTAKE, 'state not eq');

    (testVars.unstakeFine, testVars.withdrawAmount, testVars.withdrawReqId) =
    tsYieldEthStakingEtherfi
      .getNftUnstakeData(address(tsBAYC), tokenIds[0]);
    assertEq(testVars.unstakeFine, 0, 'state not eq');
    assertEq(testVars.withdrawAmount, yieldAmount, 'withdrawAmount not eq');
    assertGt(testVars.withdrawReqId, 0, 'withdrawReqId not gt');
  }
  // output
  emit log_named_string(key: "Error", val: "yieldAmount not eq")
     emit log(val: "Error: a ~= b not satisfied [uint]")
     emit log_named_uint(key: "      Left", val: 23040944067867819724 [2.304e19])
     emit log_named_uint(key: "     Right", val: 23040944067867808625 [2.304e19])
     emit log_named_uint(key: " Max Delta", val: 1)
     emit log_named_uint(key: "     Delta", val: 11099 [1.109e4])
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Do not use `eETH` balances directly for calculations.

- **Results**
  Acknowledged.

  Response from the BendDAO team:

  "No need to fix this, cos our yield must depend on the eETH balance which is a rebase model like stETH. If eETH can be manipulated or attacked, we can not integrate Etherfi."

5.3.6 | Auction duration not checked to be set

Severity	Low
Source	src/libraries/logic/IsolateLogic.sol#187;
Commit	7d9cd57;
Status	Acknowledged;

■ **Description**

In the `executeIsolateAuction()` function, there is no check to see if `nftAssetData.auctionDuration()` is set. If `auctionDuration()` is not set, it defaults to 0, which can cause issues with auctions where you can only bid once because the auction end time will be the same as the start time. Not setting `auctionDuration` will cause the auction to end immediately, and no subsequent bids will be allowed.

This only affects the second bidder, not the first bidder. This way whoever gets the first bid will be the winner.

POC

```
// test/setup/TestWithSetup.sol
  tsConfigurator.addAssetERC721(tsCommonPoolId, address(tsBAYC));
  tsConfigurator.setAssetCollateralParams(tsCommonPoolId,
  address(tsBAYC), 6000, 8000, 1000);
  // tsConfigurator.setAssetAuctionParams(tsCommonPoolId,
  address(tsBAYC), 5000, 500, 2000, 1 days);
  tsConfigurator.setAssetClassGroup(tsCommonPoolId, address(tsBAYC), tsLowRateGroupId);
  tsConfigurator.setAssetActive(tsCommonPoolId, address(tsBAYC), true);
  tsConfigurator.setAssetSupplyCap(tsCommonPoolId, address(tsBAYC), 10_000);


  run test_Should_AuctionUSDT()
  Failing tests:
Encountered 1 failing test in
test/integration/TestIntIsolateAuction.t.sol:TestIntIsolateAuction
[FAIL. Reason: revert: 607] test_Should_AuctionUSDT() (gas: 1893697)
```

■ **Exploit Scenario**

N/A.

■ **Recommendations**

Add a check in the `executeIsolateAuction()` function to ensure that `nftAssetData.auctionDuration` is set and greater than 0.

■ **Results**

Acknowledged.

Response from the BendDAO team:

"`auctionDuration` is a configurable parameter which can be 0 optional, which means there's no need auction, just the first bidder will win and liquidate the nft"

5.4 | Informational

5.4.1 | Typo in Error message

Severity	Informational
Source	src/libraries/helpers/Errors.sol#L57;
Commit	7d9cd57;
Status	Resolved in commit 7d6f7be;

- **Description**
  The `GROUP _USDED _BY _ASSET` has a typo in the word used.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Correct the error message.
- **Results**
  Resolved in commit 7d6f7be.

  The error message was corrected.

5.4.2 | Incorrect comments

Severity	Informational
Source	src/libraries/logic/ConfigureLogic.sol#L138; src/libraries/types/DataTypes.sol#L67;
Commit	7d9cd57;
Status	Resolved in commit 7d6f7be;

- **Description**
  The first comment states that the checks are performed to see that the group is not used by any asset. However, the checks are actually to see whether or not the yield is enabled for each of the assets.

  The second comment states that the asset type for ERC20 is 0 and for ERC721 is 1. However, according to the `Constants` library, the asset type for ERC20 is 1 and for ERC721 is 2.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Correct the comments in the code.

- **Results**
  Resolved in commit 7d6f7be.

  The comment was fixed.

5.4.3 | Floating solidity pragma version

Severity	Informational
Source	Global;
Commit	7d9cd57;
Status	Acknowledged;

- **Description**
  Current smart contracts use ^0.8.0. And compilers within versions ≥ 0.8.0 and <0.9.0 can be used to compile those contracts. Therefore, the contract may be deployed with a newer or latest compiler version which generally has higher risks of undiscovered bugs.

  It is a good practice to fix the solidity pragma version if the contract is not designed as a package or library that will be used by other projects or developers.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Use the fixed solidity pragma version.

- **Results**
  Acknowledged.

  Response from the BendDAO team:

  "The compiler version is specified in the config file."

5.4.4 | Unused import

Severity	Informational
Source	src/yield/YieldAccount.sol#L15;
Commit	57023f4;
Status	Resolved in commit 13e6d3b;

- **Description**
  The `console2` import in the `YieldAccount` contract is not used.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Removed unused imports.
- **Results**
  Resolved in commit 13e6d3b.

  The suggestion was implemented.

5.4.5 | Unused variable

Severity	Informational
Source	src/libraries/helpers/Constants.sol.sol#L49;
Commit	7d9cd57;
Status	Resolved in commit 7d6f7be;

- **Description**
  The `MINIMUM_HEALTH_FACTOR_LIQUIDATION_THRESHOLD` constant in the `Constants` library is defined but not used.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Use the `TransferHelper` library consistently.
- **Results**
  Resolved in commit 7d6f7be.

  The suggestion was implemented.

5.4.6 | Unintuitive processes when handling deposits and withdrawals of native tokens

Severity	Informational
Source	contracts/PoolManager.sol;
Commit	7d9cd57;
Status	Resolved;

- **Description**
  The procedure for depositing and withdrawing native tokens currently encompasses several unintuitive steps. To enhance user experience, it is advised to create distinct functions specifically for the deposit and withdrawal of native tokens,

  (The following examples assume ETH is the native token)

  Deposit ETH

  When a user deposits ETH:

  - User's ETH is deposited into the pool.
  - Wrap the ETH into WETH and transfer the weth back to the user.
  - `executeDepositERC20()` transfers the WETH from user to the pool.

  For Step 3 to proceed without issues, users must grant WETH allowance to the pool. Typically, ETH transactions do not require such approvals, leading to the expectation that no approval should be necessary.

  Proposed Improvement: This can be improved if the `executeDepositERC20()` can deposit for users. The WETH can be just sent to the pool and the pool deposits the WETH for users.

  Withdraw ETH

  When a user withdraws ETH:

  - `executeWithdrawERC20()` sends the WETH back to the user.
  - `unwrapNativeTokenInWallet()` transfers the WETH back to the pool and unwraps the ETH to the pool.
  - The pool sends the ETH back to the user.

  The need for users to authorize WETH allowance for the pool before withdrawal makes withdrawals unintuitive. Users generally do not expect the necessity of prior approval for withdrawals. And there are many back-and-forth transfers during the withdrawal.

  Proposed Improvement: Implementing a separate function for the withdrawal of native tokens could directly unwrap and return the ETH to the user, simplifying the process and making it more intuitive.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  We would like to discuss this with the team.

- **Results**
  Resolved.

  The BendDAO team added checks on the `msg.value` when wrapping and unwrapping tokens.

6 | Use Case Scenarios

The BendDAO V2 protocol allows users to perform the following actions:

- Provide liquidity in the form of ERC20 and ERC721 tokens to earn interest.
- Add an ERC721 token as collateral to borrow ERC20 tokens. This is known internally as isolate mode and it works exactly as the model that was implemented in V1
- Add ERC721 and ERC20 tokens as collateral to borrow ERC20 tokens. This is known internally as cross mdoe.
- Provide users with a chance to perform flash loans under certain conditions.
- Allow users to stake borrowed funds in Lido, Etherfi, and the DAI savings pool.

Compared with BendDAO V1, V2 changed the model to allow users to deposit ERC20 and ERC721 tokens as collateral. Additionally, it integrated a staking mechanism to allow users to stake borrowed funds into well-known staking platforms (Lido, Etherfi, and DAI savings pool)

7 | Access Control Analysis

There are different privileged roles in the BendDAO V2 protocol. A description of each of the actions for the roles can be found below:

7.1 | PoolAdmin

- Can create new modules that will be connected together through the `PoolManager` contract.
- Can create and remove pools.
- Can collect the fees.
- Can create and remove pools.
- Can set pause to pools and assets.
- Can define critical addresses like the `botAdmin` address.
- Can define critical parameters like the staking and yield parameters.
- Can add and remove assets and groups.

7.2 | EmergencyAdmin

- Can set global pause.

7.3 | OracleAdmin

- Can set chainlink aggregators.
- Can set the Bend NFT oracle address.

8 | Appendix

8.1 | Appendix I: Severity Categories

Severity	Description
High	Issues that are highly exploitable security vulnerabilities. It may cause direct loss of funds / permanent freezing of funds. All high severity issues should be resolved.
Medium	Issues that are only exploitable under some conditions or with some privileged access to the system. Users' yields/rewards/information is at risk. All medium severity issues should be resolved unless there is a clear reason not to.
Low	Issues that are low risk. Not fixing those issues will not result in the failure of the system. A fix on low severity issues is recommended but subject to the clients' decisions.
Information	Issues that pose no risk to the system and are related to the security best practices. Not fixing those issues will not result in the failure of the system. A fix on informational issues or adoption of those security best practices-related suggestions is recommended but subject to clients' decision.

8.2 | Appendix II: Status Categories

Severity	Description
Unresolved	The issue is not acknowledged and not resolved.
Partially Resolved	The issue has been partially resolved
Acknowledged	The Finding / Suggestion is acknowledged but not fixed / not implemented.
Resolved	The issue has been sufficiently resolved

9 | Disclaimer

Verilog Solutions receives compensation from one or more clients for performing the smart contract and auditing analysis contained in these reports. The report created is solely for Clients and published with their consent. As such, the scope of our audit is limited to a review of code, and only the code we note as being within the scope of our audit is detailed in this report. It is important to note that the Solidity code itself presents unique and unquantifiable risks since the Solidity language itself remains under current development and is subject to unknown risks and flaws. Our sole goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies. Thus, Verilog Solutions in no way claims any guarantee of security or functionality of the technology we agree to analyze.
In addition, Verilog Solutions reports do not provide any indication of the technology's proprietors, business, business model, or legal compliance. As such, reports do not provide investment advice and should not be used to make decisions about investment or involvement with any particular project. Verilog Solutions has the right to distribute the Report through other means, including via Verilog Solutions publications and other distributions. Verilog Solutions makes the reports available to parties other than the Clients (i.e., "third parties") – on its website in hopes that it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.