BIFROST

BiFi-X Security Audit

revision 1.0

Prepared for
Bifrost

Prepared by
Andrew Wesie / Theori
Tim Becker / Theori
Tyler Nighswander / Theori
Brian Pak / Theori

June 15th, 2021

Theori

Table of Contents

Executive Summary

Starting on May 31, 2021, Theori assessed updates to the Bifrost lending contract, called BiFi, and its new companion product, BiFi-X. We focused on identifying issues that result in a loss of funds for either users or the contracts' reserves. We started with a review of the flash loan feature added to BiFi. We then reviewed BiFi-X, which uses the flash loan feature to create leveraged positions. The critical issues that we identified were promptly fixed by the Bifrost team and we reviewed the revised code.

Scope

Updates to Bifrost’s lending system, BiFi, a smart contract on the Ethereum blockchain that implements a lending service for ether and ERC20 tokens, are audited. These updates implement support for flash loans. BiFi-X, which uses the flash loan feature, is also audited.

The code was received on May 27, 2021. Commit hash was not given. Additional changes were received on June 9, 2021.

Overview

In this section we describe how we approached the assessment and the issues that a flash loan feature may introduce. We did not find any vulnerabilities in the BiFi contract related to these issues, but they deserve mention due to their importance. The BiFi-X contract, however, did fail to check whether it was expecting a flash loan callback which we discuss in more detail in the findings.

Approach

We approached this assessment from two angles: smart contract vulnerabilities and economic vulnerabilities. For finding smart contract vulnerabilities, we relied on static analysis tools and manual source code audits. We considered whether the source code is currently vulnerable and whether the source code was written securely. For example, the BiFi contracts use safe math functions that prevent integer overflow and underflow. On the other hand, there are no reentrancy guards to prevent reentrancy attacks, so we need to review these manually.

We identify in the appendix the external functions for each new smart contract. We check that the functions are correctly permissioned or annotated with view. For each external function that is not permissioned, we audited the source code for vulnerabilities. We do not consider vulnerabilities in functions that can only be accessed by the owner, since the owner can usually change the contract's code via a proxy and is trusted.

Flash Loans

Since 2020, flash loans have become a staple of decentralized finance protocols. Several examples include Aave, bZx, dYdX, Uniswap, and now BiFi. While flash loans are a useful building block for DeFi products, as demonstrated by BiFi-X, they also introduced a new set of risks to the decentralized finance ecosystem. For this review, we limit our scope to BiFi and BiFi-X.

There are several potential issues that we considered when reviewing the BiFi flash loan feature. The most important is validating that the flash loan is repaid. Next, there are issues inherent to calling out to a potentially malicious contract: ensure BiFi state is consistent and ensure re-entrancy will not break assumptions. Lastly, validate that the flash loan receiver, e.g. BiFi-X, correctly authenticates `msg.sender` and expects the callback.

Ensuring that the BiFi state is consistent during the flash loan required a review of what state is modified by the flash loan and what state is used internally by BiFi. The only state that the flash loan modifies before the call to the receiver is transferring the borrowed tokens out. Since BiFi keeps track of its balances internally, without relying on the tokens' `balanceOf` function, this has no effect on any state that is used internally by BiFi. The worst case is that a BiFi call tries to do a transfer which reverts due to lack of liquidity. If any future update changes this behavior, the security of flash loans should be revisited.

Related to a consistent state is support for re-entrancy. Some protocols choose to prevent re-entrancy altogether which eliminates the issues at the cost of flexibility for users. We did not identify any security vulnerabilities in BiFi due to re-entrancy, though Bifrost should consider if it is worth preventing re-entrancy as a preemptive measure. There is a possible correctness issue with the calculation of the expected balance in flashloan, and whether it makes sense to transfer that entire amount to the token handler. We chose not to mark this as an issue because the current behavior is safe, and a hasty fix might introduce more issues.

Bug Fixes

During the assessment, the Bifrost team quickly fixed the issues we found that could result in a loss of assets. These fixes were provided to us in an updated version of the code on June 9, 2021. We encourage the Bifrost team to continue to carefully review and track all changes to the smart contract code. In the findings below, we identify each issue that was fixed by the Bifrost team.

Findings

These are the potential issues that may have correctness and/or security impacts. We advise Bifrost team to remediate found issues quickly to ensure the safety of the contract.

Summary

#	ID	Title	Severity
1	THE-BIFIX-001	Do not use `tx.origin` for authorization	Fixed (Critical)
2	THE-BIFIX-002	Insufficient authorization in `rewardClaimAll`	Fixed (Critical)
3	THE-BIFIX-003	Vulnerable to sandwich attack when using Uniswap	Fixed (Critical)
4	THE-BIFIX-004	Wrong selector in proxy for `ownerRewardTransfer`	Medium
5	THE-BIFIX-005	Incorrect calculation of transfer amount in `_flashloan`	Minor
6	THE-BIFIX-006	`feeTotal` is not updated in `_flashloan`	Minor
7	THE-BIFIX-007	Incomplete withdraw in `endStrategy`	Medium

Issue #1: Do not use tx.origin for authorization

ID	Summary	Severity
THE-BIFIX-001	Authorization using `tx.origin` can be bypassed by an attacker tricking a user into sending a transaction to an attacker-controlled contract.	Fixed (Critical)

Any contract that uses a flash loan API must have an external receiver function that executes the contract logic with the borrowed coins or tokens. BiFi-X uses the flash loan API provided by BiFi, and must implement `executeOperation`:

```
function executeOperation(
    address reserve,
    uint256 amount,
    uint256 fee,
    bytes calldata params
) external onlyManager payable returns (bool) {
  // Check whether receiver origin caller is owner in callback function
  require(tx.origin == owner);
...
```

While BiFi-X correctly verifies that the caller is the expected manager contract, it does not correctly validate who initiated the flash loan. An attacker that tricks a user into sending ether to an attacker-controlled contract with enough gas would be able to initiate a flash loan with attacker-controlled params. This may allow the attacker to steal from the BiFi-X contract.

In BiFi-X, the only times that `executeOperation` should be called is from `startStrategy` and from `endStrategy`. We proposed two ways this could be enforced:

1. Add an additional parameter to `executeOperation` called initiator which indicates the sender that called `flashloan`
2. Use a flag in the BiFi-X storage to indicate that a call to `executeOperation` is expected

Fix

The Bifrost team chose to fix this issue by using a flag in BiFi-X: `startedByOwner`. This flag is set to true at the beginning of `startStrategy` and the beginning of `endStrategy`. The receiver function, `executeOperation`, validates that `startedByOwner` is set and then unsets it before returning.

This fix is sufficient; however, we suggest keeping the flag set for the least amount of time possible. It would be better to set the flag immediately before the call to `flashloan` and then unset the flag immediately after the require statement in `executeOperation`.

```
function startStrategy(bytes memory params) external payable override returns (bool)
{
  // check strategy already executed
  require(!strategyExecuted);
  address _this = address(this);

  strategyExecuted = true;
  startedByOwner = true;
...
}

function endStrategy(bytes memory params) external onlyOwner payable override
returns (bool) {
  startedByOwner = true;
...
}

function executeOperation(
    address reserve,
    uint256 amount,
    uint256 fee,
    bytes calldata params
) external onlyManager returns (bool) {
  // check executeOperation entry point is start or end Strategy
  require(startedByOwner, "onlyOwner");
...
  startedByOwner = false;
  return true;
}
```

We also suggest that any user documentation for the flash loan feature includes information about this type of vulnerability and how it can be prevented by the user.

Issue #2: Insufficient authorization in rewardClaimAll

ID	Summary	Severity
THE-BIFIX-002	The external `rewardClaimAll` function in `StrategyLogic` was missing access controls.	Fixed (Critical)

One of the benefits for users of BiFi is the BIFI incentive token. In BiFi-X, the external `rewardClaimAll` function was missing access controls and anybody could withdraw the BIFI tokens that were earned by the BiFi-X contract.

```
function rewardClaimAll(address userAddr, address _managerAddr) external {
  _rewardClaimAll(userAddr, _managerAddr);
}

function _rewardClaimAll(address userAddr, address _managerAddr) internal {
  IMarketManager manager = IMarketManager(_managerAddr);
  IERC20 bifi = IERC20(BIFI_ADDRESS);

  manager.rewardClaimAll(payable(userAddr));
  bifi.transfer(msg.sender, bifi.balanceOf(address(this)));
}
```

Fix

The Bifrost team replaced the `rewardClaimAll` function with a new function, `rewardClaim`. It does not take any parameters and will transfers rewards to the owner instead of `msg.sender`. As such, this new function can be safely called by anyone and does not require any access controls.

```
function rewardClaim() external  {
  _rewardClaim();
}

function _rewardClaim() internal {
  address userAddr = address(this);

  address bifiAddr = factory.getBifiAddr();
  IMarketManager manager = IMarketManager(factory.getManagerAddr());
  IERC20 bifi = IERC20(bifiAddr);

  uint256 handlerLength = handlerIDs.length;
  for (uint256 i=0; i < handlerLength; i++) {
    manager.claimHandlerReward(handlerIDs[i], payable(userAddr));
  }

  address owner = NFT.ownerOf(productID);
  bifi.safeTransfer(owner, bifi.balanceOf(userAddr));
}
```

Issue #3: Vulnerable to sandwich attack when using Uniswap

ID	Summary	Severity
THE-BIFIX-003	`_lockPositionSwapPath` and `_unlockPositionSwapPath` do not limit slippage on the Uniswap trades, making them potentially vulnerable to sandwich attacks.	Fixed (Critical)

`_lockPositionSwapPath` uses `swapExactTokensForTokens` and passes 0 for `amountOutMin`:

```
function _lockPositionSwapPath(address _this, uint256 count, address[] memory path)
internal {
...
  uniswap.swapExactTokensForTokens(tmpBalance, 0, path, _this, _timestamp);
...
   uniswap.swapExactTokensForTokens(tmpBalance, 0, path, _this, _timestamp);
...
   uniswap.swapExactTokensForTokens(tmpBalance, 0, secondPath, _this, _timestamp);
...
}
```

The Uniswap documentation for trading from a smart contract[1] explicitly discusses the risk of passing too low of a value for `amountOutMin`. An attacker could sandwich the transaction between two attacker transactions and modify the price on Uniswap. This can result in a "considerable loss."

`_unlockPositionSwapPath` is similarly vulnerable as it uses `getAmountsIn` to calculate the `amountInMax` parameter of `swapTokensForExactTokens`:

```
function _unlockPositionSwapPath(address _this, uint256 count, uint256 totalDebt,
address[] memory path) internal {
...
  uint256 amountsIn = uniswap.getAmountsIn(totalDebt, path)[0];
  swapToken.approve(address(uniswap), amountsIn);
  uniswap.swapTokensForExactTokens(totalDebt, amountsIn, path, _this, _timestamp);
...
}
```

[1] https://uniswap.org/docs/v2/smart-contract-integration/trading-from-a-smart-contract/

The amount of assets that can be stolen during `_lockPositionSwapPath` by an attacker depends on the parameters sent as part of the transaction. The parameters can be chosen such that there is no opportunity for an attacker.

In the first version of BiFi-X:

collateralAmount = (1 + maxSlippage) * flashloanAmount * strategyRate

In the second version of BiFi-X:

lendingAmount = (1 + maxSlippage) * flashloanAmount * collateralPrice / lendingPrice

The max slippage is a trade-off in many decentralized finance systems. This parameter should be high enough that the transaction will execute, but low enough to prevent front running. Some systems allow their users to determine this parameter.

Fix

The Bifrost team fixed the issue in `_unlockPositionSwapPath` before we reported it to them. The parameters for `endStrategy` now includes `collateralOutMin`:

```
function _unlockPositionSwapPath(address _this, StrategyParams memory vars) internal
{
...
  uniswap.swapExactTokensForTokens(swapAmount, vars.collateralOutMin, vars.path,
_this, params.timestamp);
...
   uniswap.swapExactTokensForTokens(swapAmount, 0, firstPath, _this,
params.timestamp);


...
   uniswap.swapExactTokensForTokens(wethBalance, vars.collateralOutMin, secondPath,
_this, params.timestamp);
...
}
```

Whether a front runner can make money from a sandwich attack will depend on the parameters that are chosen by the BiFi-X frontend. It is important that `lendingAmount` (for `startStrategy`) and `collateralOutMin` (for `endStrategy`) are chosen carefully.

Issue #4: Wrong selector in proxy for ownerRewardTransfer

ID	Summary	Severity
THE-BIFIX-004	The `TokenManager` contract proxies the `ownerRewardTransfer` to the wrong function in `HandlerManager`.	Medium

One of the changes to BiFi was the introduction of the `TokenManager` proxy contract and the separation of the logic into multiple contracts. However, the `ownerRewardTransfer` was proxied to the wrong selector (`claimHandlerReward`).

```
function ownerRewardTransfer(uint256 _amount) onlyOwner external returns (bool
result)
{
  bytes memory callData = abi.encodeWithSelector(
    IHandlerManager.claimHandlerReward.selector,
    _amount
  );

  (result, ) = handlerManagerAddr.delegatecall(callData);
  assert(result);
}
```

The fix is to specify the correct selector, `IHandlerManager.ownerRewardTransfer.selector`. There is no security impact in this case, but we recommend that more care is taken when manually writing proxy contracts to avoid this type of bug.

Fix

The Bifrost team fixed the issue after reviewing the report.

Issue #5: Incorrect calculation of transfer amount in _flashloan

ID	Summary	Severity
THE-BIFIX-005	When expected balance is less than the after balance, the wrong amount to send back to the user is calculated.	Minor

For flash loan of ether, when the expected balance is less than the after balance, the calculation for the amount to transfer is incorrect. This is a correctness bug that does not have any security impact. Since the subtraction would underflow, the `SafeMath` library will revert the transaction.

```
function _flashloan(...) {
...
  // payback: over repay amount
  if(handlerInfo.expectedBalance < handlerInfo.afterBalance){
    msg.sender.transfer(handlerInfo.expectedBalance.sub(handlerInfo.afterBalance));
  }
...
}
```

The correct code is already present in the case for handling ERC20 tokens:

```
function _flashloan(...) {
...
  // payback: over repay amount
  if(handlerInfo.afterBalance > handlerInfo.expectedBalance){
    token.transfer(msg.sender,
        handlerInfo.afterBalance.sub(handlerInfo.expectedBalance));
  }
...
}
```

Fix

The Bifrost team fixed the issue after reviewing the report.

Issue #6: feeTotal is not updated in _flashloan

ID	Summary	Severity
THE-BIFIX-006	`_flashloan` accumulates the fee with the `feeTotal` field but does not store the result.	Minor

The `feeTotal` field of the `HandlerFlashloan` structure is used to accumulate the fees earned from flash loans for a coin or token. This is later used in `withdrawFlashloanFee` for the owner to withdraw only the fees.

```
function _flashloan(...) {
...
  // fee store in manager slot
  handlerFlashloan[handlerID].feeTotal.add(fee);
...
}
```

While the code above correctly adds the fee to the old `feeTotal`, it does not store the result and the `feeTotal` is not updated. This is a correctness bug that does not have any security impact.

Fix

The Bifrost team fixed the issue after reviewing the report.

Issue #7: Incomplete withdraw in endStrategy

ID	Summary	Severity
THE-BIFIX-007	If the BiFi contract has insufficient liquidity, `endStrategy` in BiFi-X may not withdraw all of the collateral before it calls `selfdestruct`.	Medium

The purpose of `endStrategy` in BiFi-X is to close a position, withdraw all of the assets to the owner, and then self-destruct. This requires repaying the loan from BiFi and then withdrawing the collateral:

```
function _unlockPosition(StrategyParams memory vars) internal returns (bool) {
  uint256 index = handlersAddress.length.sub(1);
  _repay(vars.tokenAddr[index], handlersAddress[index], vars.decimal);
  _withdraw(handlersAddress[0], EXECUTE_AMOUNT);
  return true;
}
```

The correctness of this code relies on BiFi transferring the correct amount of collateral back to the BiFi-X contract. In normal circumstances, the behavior is correct, but if the BiFi contract lacks sufficient liquidity it will succeed but withdraw less collateral than expected:

```
function _getUserActionMaxWithdrawAmount(address payable userAddr, uint256
requestedAmount, uint256 collateralableAmount) internal view returns (uint256)
{
  uint256 depositAmount = handlerDataStorage.getUserIntraDepositAmount(userAddr);
  uint256 handlerLiquidityAmount = _getTokenLiquidityAmount();
  /* the minimum of request, deposit, collateral and collateralable*/
  uint256 minAmount = depositAmount;
  if (minAmount > requestedAmount)
    minAmount = requestedAmount;
  if (minAmount > collateralableAmount)
    minAmount = collateralableAmount;
  if (minAmount > handlerLiquidityAmount)
    minAmount = handlerLiquidityAmount;
  return minAmount;
}
```

One way to mitigate this issue in BiFi-X is to require that the contract has no remaining assets:

```
function _unlockPosition(StrategyParams memory vars) internal returns (bool) {
  uint256 index = handlersAddress.length.sub(1);
  _repay(vars.tokenAddr[index], handlersAddress[index], vars.decimal);
  _withdraw(handlersAddress[0], EXECUTE_AMOUNT);

  require( IProxy(handlersAddress[0]).getUserAmountWithInterest(this)[0] == 0 );

  return true;
}
```

Fix

The Bifrost team fixed the issue after reviewing the report. They added require statements to the `_withdraw` function to validate that the balance after the withdraw is within an accepted range (`gap`).

Code Quality Recommendations

These are the recommendations to improve the code quality for better readability and optimization. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	Update comments to reflect updates to the code	Code quality	Minor
2	Review and fix typos	Code quality	Minor
3	Use WETH instead of native ether	Code quality	Minor
4	Document or fix require in `executeOperation`	Code quality	Minor
5	Use latest Uniswap router contract (UniswapV2Router02)	Code quality	Minor

Recommendation #1: Update comments to reflect updates to the code

The parameters for some functions were updated but their comments have not been updated. For instance, the `TokenManager` constructor documentation does not match its signature.

The comments within InterestModel `_getSIRandBIR` do not match the updated code for `_getUtilizationRate`.

Fix

The Bifrost team fixed the issue after reviewing the report.

Recommendation #2: Review and fix typos

During the review, we found some likely typos that should be fixed to improve readability of the code:

TokenManager	_getHandlerAmountWithAmount
Updater	rewerder

Fix

The Bifrost team fixed the issue after reviewing the report.

Recommendation #3: Use WETH instead of native ether

In BiFi and BiFi-X, additional code is needed to handle both ether and ERC20 tokens. BiFi-X also needs to deposit ether into the WETH contract in order to interact with Uniswap. The code for both BiFi and BiFi-X could be simplified by using WETH internally.

Recommendation #4: Document or fix require in executeOperation

In `executeOperation` in `StategyLogic`, there is a require statement to ensure that the amounts and fees are reasonable:

```
function executeOperation(...) {
```

```
...
    // calculate flashloan total debt for repay flashloan
    vars.totalDebt = amount.add(fee);
    require(_convertUnifiedToUnderlying(vars.amounts[2], vars.decimal) == amount ||
vars.amounts[2].add(vars.fees[0]) >= vars.totalDebt);
...
}
```

However, the left half of the condition implies that `vars.amounts[2]` is a unified quantity and amount is an underlying quantity, while the right half of the condition implies that `vars.amounts[2]` and amount are the same type of quantity. This is either a bug that needs fixed or a comment is necessary to explain why this is correct.

Fix

The Bifrost team fixed the issue after reviewing the report by removing the require statement.

Recommendation #5: Use latest Uniswap router contract (UniswapV2Router02)

Quoting the Uniswap documentation: "UniswapV2Router01 should not be used any longer, because of the discovery of a low severity bug and the fact that some methods do not work with tokens that take fees on transfer. The current recommendation is to use UniswapV2Router02."

While the issues with UniswapV2Router01 likely don't affect the current version of BIFI-X, we suggest switching to UniswapV2Router02 to prevent these issues from manifesting in the future.

Fix

The Bifrost team fixed the issue after reviewing the report.

Appendix: External APIs – Flash Loan

ManagerFlashLoan

flashloan	**external**	
withdrawFlashloanFee	external	onlyOwner
getFee	external	view
getFeeTotal	external	view

Appendix: External APIs – BiFi-X

BiFiNFT

mint	**external**	
safeTransferFrom	**external**	
setApprovalForAll	**external**	
approve	external	canOperate(_tokenId)
transferFrom	external	canTransfer(_tokenId)
balanceOf	external	view
getApproved	external	view
isApprovedForAll	external	view
ownerOf	external	view
tokenByIndex	external	view
tokenOfOwnerByIndex	external	view
totalSupply	external	view

XFactory

create	**external**	
setBifiAddr	external	onlyOwner
setDiscountBase	external	onlyOwner
setFee	external	onlyOwner
setManagerAddr	external	onlyOwner
setNFT	external	onlyOwner
setStrategy	external	onlyOwner
setUniswapV2Addr	external	onlyOwner
setWethAddr	external	onlyOwner
getBifiAddr	external	view
getManagerAddr	external	view
getNFT	external	view
getStrategy	external	view
getUniswapAddr	external	view
getWETHAddr	external	view

payFee	external	view

PositionStorage

newUserProduct	external	onlyFactory
setProductToNFTID	external	onlyFactory
setFactory	external	onlyOwner
setStrategy	external	onlyOwner
getNFTID	external	view
getStrategy	external	view
getUserProducts	external	view

ProductProxy

setStrategyID	external	onlyOwner
getStrategyID	external	view

StrategyLogic

rewardClaim	**external**	
startStrategy	**external**	
endStrategy	external	onlyOwner
executeOperation	external	onlyManager
getProductID	external	view
getSeedAsset	external	view