BIFROST

BiFi
Lending Contract
Security Audit

revision 1.0

Prepared for
Bifrost

Prepared by
Andrew Wesie / Theori
Brian Pak / Theori
Joohyun Park / Theori
Taeyang Lee / Theori

February 8th, 2021

Theori

Table of Contents

Executive Summary

Starting on January 18, 2021, Theori assessed the Bifrost lending contract, called BiFi. We focused on identifying issues that result in a loss of funds for either users or the contract's reserves. We started with a review of the external functions exposed by the BiFi contracts. We then considered the economic model of the BiFi contract and whether the contracts allow an attacker to violate the model. We identified two critical vulnerabilities that could result in a loss of funds. One of these vulnerabilities was concurrently found by the Bifrost team, and the Bifrost team promptly developed fixes for both vulnerabilities. We reviewed the proposed fixes and confirmed that they fixed the vulnerabilities.

Scope

Bifrost's lending system, BiFi, a smart contract on the Ethereum blockchain that implements a lending service for ether and ERC20 tokens is audited. The following is the list of the files reviewed by Theori for this assessment.

- bifi-code-for-audit (2021-01-18)
- BiFi Doc (2021-01-18)
- bifi-solidity-contracts_2021-01-21 (2021-01-21)

The code was received on January 18th, 2021. Commit hash was not given.

Overview

In this section we describe how we approached the assessment and our assumptions about the thread model. We also discuss two acute issues for lending contracts: oracle price manipulation and liquidation. We did not find any vulnerabilities in BiFi contract related to these issues, but they deserve mention due to their importance.

Threat Model

While conducting an assessment, it is important to define who is a possible threat actor and what does it mean for an attack to be successful. For this assessment, we assume that the BiFi contract owner(s) are trusted, as it would be trivial for them to change the source code by changing which implementation contracts the proxy contracts are using. We assume that all other users are potentially malicious. Relatedly, we assume that contracts written by Bifrost are not malicious and the source code we audited is an accurate representation of the source code going forward.

Fundamentally, there are two types of successful attacks on a lending contract: attacks that may result in a loss of funds and denial of service attacks. A loss of funds is not limited to the loss of user funds, it also includes reserve funds, reward tokens, and the solvency of the system. Examples of situations that may lead to a loss of funds are:

- Attacker can steal funds from other users
- Attacker borrows more assets than used as collateral or otherwise violates limits on margin
- Attacker prevents their position from being liquidated
- Attacker unfairly increases their interest earned or decreases their interest to be paid
- Attacker unfairly increases their reward token earnings
- Manipulation of system parameters by an attacker
- Causes the system to pay more interest to depositors than is being earned from borrowers

Denial of service attacks are less critical because they do not result in a loss of funds, and the contract owner(s) can deploy new contracts that fix the vulnerabilities. Generally speaking, a denial-of-service attack happens when an attacker causes the contract to always revert for other users, e.g., run out of gas.

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, but the source code currently interacts with external contracts after any effects.

We identify in the appendix the external functions for each Bifrost 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 note that in the proxy contracts there are external functions that are not permissioned nor view only (*`handlerViewProxy`* and *`siViewProxy`*) that allow an attacker to call external functions in the implementation contracts. We do not consider this to be a vulnerability because the implementation contracts correctly permission their external functions.

Economic vulnerabilities require a deep understanding of a function's logic and its interactions with the rest of the contract and system. As such, these cannot be easily discovered by automated tools and safe coding practices are not a sufficient mitigation. For each external API, we considered whether it, along with the functions it calls, would enable an attacker to violate the intentions of the BiFi lending system. For example, we found one vulnerability due to an improperly initialized variable that would allow an attacker to instantly earn interest on a deposit and remove their deposit, atomically. This violates the intention that deposits earn interest by contributing to a pool of funds and all interest earned by depositors is less than or equal to interest paid by borrowers.

Oracle Manipulation

One type of economic vulnerability to which lending contracts are especially susceptible are oracle manipulation attacks. The premise of a lending contract is that the borrowed assets are fully backed by collateral that is worth more than the borrowed assets. This requires the contract to know the value of the collateral and the value of the borrowed assets. If an attacker can confuse the contract to believe that collateral is worth more or the borrowed asset is worth less than in reality, they would be able to borrow too much.

ETH	`0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419`

USDT	`0xEe9F2375b4bdF6387aa8265dD4FB8F16512A1d46`
	divided by
	`0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419`

DAI	`0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9`
LINK	`0x2c1d072e956AFFC0D435Cb7AC38EF18d24d9127c`

Currently, the BiFi contracts use the Chainlink oracle contracts. Chainlink is the de facto standard for a decentralized oracle on the Ethereum blockchain. Chainlink's price oracles rely on off-chain price feeds, which avoids the problems caused by flash loans that could influence the price of on-chain exchanges. There are still some risks to the BiFi contract. If Chainlink is malicious, it could execute a reentrancy attack on the BiFi contracts. Or, if the Chainlink on-chain price feed is not updated, the contract will incorrectly value assets. We do not believe these are serious concerns and we believe that the current oracles are better than a homegrown solution.

Liquidation

Once a user has borrowed assets based on their collateral, the system needs to monitor the assets to ensure that the value of the borrowed assets is never greater than the value of the collateral. If the value of the collateral decreases or the value of the borrowed assets increase, then it may be necessary for the system to liquidate part of the user's collateral to ensure that the loan-to-value (LTV) remains below 100%.

The BiFi contract requires a third-party to initiate a liquidation and pay for the borrowed assets. In exchange, the third-party receives a proportional amount of the borrower's collateral. The liquidator also receives a fee that depends on the borrower's current LTV. If the LTV is 95%, then the liquidator receives a fee of 5%, whereas if the LTV is 99% then the liquidator receives only a 1% fee. This incentivizes third parties to liquidate as soon as possible to maximize their fee. With the availability of flash loans on the Ethereum blockchain, liquidators only need enough capital to pay for gas to liquidate almost any borrower.

One concern with the liquidation on the BiFi contract is that underwater borrowers will never be liquidated, because the value of collateral received by the liquidator will be less than the value of the assets they must pay. Bifrost should consider establishing an insurance fund that can be used by a trusted liquidator to absorb these losses. Otherwise, the lending contract will continue to accrue interest that may never be paid and eventually run out of its reserves.

The threshold for determining when a borrower can be liquidated depends on the margin call limit of their collateral. The margin call limit should depend on how quickly an asset can be liquidated with a minimal loss due to spread, so stable assets should have a higher margin call limit than highly volatile assets. At this time, the margin call limits for the assets supported by Bifrost are 95%.

ETH	Bifrost	95%
	Compound	75%
	AAVE	82.5%
	Maker	66.6% - 76.9%

We identify two concerns with the current margin call limit: it is high compared to Bifrost's peers, and it does not take into account the volatility of the borrowed asset. Two of the largest DeFi lending contracts on Ethereum are AAVE and Compound with a combined $10B of assets. The margin call limits for these contracts are around 80%. This is noticeably lower than Bifrost's margin call limit of 95% which raises the question of whether the current Bifrost limit is too risky especially during periods of network congestion. Also, while the volatility of the underlying collateral is a risk, the volatility of the borrowed asset is also a risk and is not accounted for in the current Bifrost margin call limit. Bifrost should consider if the current parameters accurately reflect the risk of the supported assets.

One change to the contract code during the audit was in the `partialLiquidationUserReward` function. Previously, the borrower's collateral was transferred to the liquidator within the BiFI system, and the liquidator would need to call into the BiFi contract to withdraw the reward. In order to reduce the gas costs of liquidation, this was changed so that the reward is transferred out of the BiFi contract into the liquidator's wallet. The downside of this change is that the transfer may fail if there is not enough liquidity in the reward asset. While this does not substantially change the incentive model of liquidation, it does highlight an additional risk that a borrower eligible for liquidation may not be liquidated due to lack of liquidity in the asset used as collateral.

Bug Fixes

During the assessment, the Bifrost team quickly fixed the two issues we found that could result in a loss of funds. One of these issues had already been identified by Bifrost during code refactoring. Additionally, the Bifrost team identified and fixed a separate issue that resulted in a divide-by-zero during liquidation but did not have any security impact. For each of the hot fixes, the Bifrost team provided an explanation of the fix along with the changes to the contract source code. 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 how each issue 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-BIFI-001	Wrong variable usage in `_applyInterestHandlers`	Fixed (Critical)
2	THE-BIFI-002	Insufficient variable initialization in `setNewCustomer`	Fixed (Critical)
3	THE-BIFI-003	Uninitialized variable when calculating reward in `interestUpdateReward`	Medium

Issue #1: Wrong variable usage in _applyInterestHandlers

ID	Summary	Severity
THE-BIFI-001	Wrong variable is used when calculating *withdrawableAsset* in *_applyInterestHandlers*	Fixed (Critical)

When determining how many tokens can be borrowed or withdrawn, coinHandler and tokenHandler call into *applyInterestHandlers* in the *marketManager* contract (tokenManager.sol):

```
function applyInterestHandlers(address payable userAddr, uint256 callerID, bool allFlag)
external override returns (uint256, uint256, uint256)
{
    uint256 userBorrowableAmount;
    uint256 collateralable;
    uint256 callerPrice;
    (userBorrowableAmount, collateralable, , , , callerPrice) =
_applyInterestHandlers(userAddr, callerID, allFlag);
    return (userBorrowableAmount, collateralable, callerPrice);
}

function _applyInterestHandlers(address payable userAddr, uint256 callerID, bool allFlag)
internal returns (uint256, uint256, uint256, uint256, uint256, uint256)
{
    // ...
    userAssetsInfo.userBorrowableAsset = 0;
    if (userAssetsInfo.depositAssetBorrowLimitSum > userAssetsInfo.borrowAssetSum)
    {
        userAssetsInfo.userBorrowableAsset = sub(userAssetsInfo.depositAssetBorrowLimitSum,
userAssetsInfo.borrowAssetSum);
    }
    /* Set the allowed amount that the user can withdraw based on the user borrow */
    userAssetsInfo.withdrawableAsset = 0;
    if (userAssetsInfo.depositAssetBorrowLimitSum > userAssetsInfo.borrowAsset)
    {
```

```
        userAssetsInfo.withdrawableAsset =
unifiedDiv(sub(userAssetsInfo.depositAssetBorrowLimitSum, userAssetsInfo.borrowAsset),
userAssetsInfo.callerBorrowLimit);
    }
    // ...
}
```

When calculating *withdrawableAsset*, the *borrowAsset* variable is used. The *borrowAssetSum* variable should be used instead.

An attack scenario would look like:

- Attacker deposits 1000 DAI
- Attacker borrows a minimal amount of LINK (e.g. 0.00000001 LINK)
- Attacker borrows the maximum amount of ETH with remaining assets (e.g. 0.5 ETH)
- Attacker withdraws 999 DAI

The withdraw succeeds because the last token handled in the loop is LINK, so the *borrowAsset* variable will contain only the value of the LINK that was borrowed. This is subtracted from the borrow asset limit sum, which is the sum of all of the assets (1000 DAI) multiplied by the borrow limit percentage. The result is that *withdrawableAsset* is slightly less than 1000 DAI and the attacker is able to withdraw too many assets.

Hot Fix

This vulnerability was found concurrently by the Bifrost team as they were refactoring *applyInterestHandlers*. The vulnerability was fixed by changing *borrowAsset* to *borrowAssetSum*:

```
function applyInterestHandlers(address payable userAddr, uint256 callerID, bool allFlag)
external override returns (uint256, uint256, uint256, uint256, uint256, uint256)
{
    // ...
    if (userAssetsInfo.depositAssetBorrowLimitSum > userAssetsInfo.borrowAssetSum)
    {
    /* Set the amount that the user can borrow from the borrow limit and previous borrows. */
        userAssetsInfo.userBorrowableAsset =
userAssetsInfo.depositAssetBorrowLimitSum.sub(userAssetsInfo.borrowAssetSum);

        /* Set the allowed amount that the user can withdraw based on the user borrow */
```



```
        userAssetsInfo.withdrawableAsset =
userAssetsInfo.depositAssetBorrowLimitSum.sub(userAssetsInfo.borrowAssetSum).unifiedDiv(userAs
setsInfo.callerBorrowLimit);
    }
    // ...
}
```

Issue #2: Insufficient variable initialization in setNewCustomer

ID	Summary	Severity
THE-BIFI-002	`userDepositEXR`, `userBorrowEXR` initialization for new customer is insufficient	Fixed (Critical)

When a new customer is making their first transaction, the user's information is initialized in storage by `setNewCustomer`. At this time, the user's initial EXR is set to `unifiedPoint`.

```
function setNewCustomer(address payable userAddr) onlyBifiContract circuitBreaker external
override returns (bool)
{
    intraUsers[userAddr].userAccessed = true;
    intraUsers[userAddr].userDepositEXR = unifiedPoint;
    intraUsers[userAddr].userBorrowEXR = unifiedPoint;
    return true;
}
```

Additionally, `_applyInterest` has a fast path for new customers that skips the call to `_updateInterestAmount`:

```
function _applyInterest(address payable userAddr) internal returns (uint256, uint256)
{
    bool newCustomer = _checkNewCustomer(userAddr);
    _checkFirstAction();

    // new Custormer has no amount
    if (newCustomer)
    {
        return (0, 0);
    }

    return _updateInterestAmount(userAddr);
}
```

EXchange Rate (EXR) is a cumulative interest rate that is used to efficiently calculate compound interest per block without applying the interest for every user on every block. There is a global exchange rate that represents the interest applied to the lending token contract, and a user exchange rate represents the interest that has been applied to the user's balance in the lending token contract. When the user's interest needs to be applied, the interest rate is calculated by performing *`(GlobalEXR) / (UserEXR)`* so that the user's exchange rate catches up to the global exchange rate.

```
function _getDeltaEXR(uint256 newGlobalEXR, uint256 lastUserEXR) internal pure returns (bool,
uint256)
{
    uint256 EXR = unifiedDiv(newGlobalEXR, lastUserEXR);
    if (EXR >= unifiedPoint)
    {
        return (false, sub(EXR, unifiedPoint));
    }

    return (true, sub(unifiedPoint, EXR));
}
```

However, since a new user's EXR is set to *`unifiedPoint`* in *`setNewCustomer`*, instead of the current *`GlobalEXR`*, the new user will receive more interest on their initial deposit than should be received. This allows for an attacker to extract assets from the lending contract:

- Create a new smart contract with the logic:
    - Use a flash loan to borrow 400 ETH
    - Deposit 400 ETH in BiFi contract
    - Withdraw 401 ETH from BiFi contract
- During testing, an attacker would gain 0.269 ETH per transaction before gas fees

The new customer's EXR should be initialized with *`GlobalEXR`*, for example:

```
function setNewCustomer(address payable userAddr) onlyBifiContract circuitBreaker external
override returns (bool)
{
    intraUsers[userAddr].userAccessed = true;
    intraUsers[userAddr].userDepositEXR = globalDepositEXR;
    intraUsers[userAddr].userBorrowEXR = globalBorrowEXR;
    return true;
}
```

Hot Fix

The Bifrost team applied two fixes, either of which would be sufficient to fix this vulnerability. In `_applyInterest`, always call `_updateInterestAmount` even if this is a new customer. And, in `_checkNewCustomer`, set the user's EXR to the global EXR.

```
function _applyInterest(address payable userAddr) internal returns (uint256, uint256)
{
    _checkNewCustomer(userAddr);
    _checkFirstAction();
    return _updateInterestAmount(userAddr);
}

function _checkNewCustomer(address payable userAddr) internal returns (bool)
{
    if (handlerDataStorage.getUserAccessed(userAddr))
    {
        return false;
    }

    handlerDataStorage.setUserAccessed(userAddr, true);
    (uint256 gDEXR, uint256 gBEXR) = handlerDataStorage.getGlobalEXR();
    handlerDataStorage.setUserEXR(userAddr, gDEXR, gBEXR);
    return true;
}
```

Issue #3: Uninitialized variable when calculating reward in interestUpdateReward

ID	Summary	Severity
THE-BIFI-003	Problems may occur due to uninitialized variable used in *interestUpdateReward*, a function that updates the interest of each token and receives a reward.	Medium

```
// marketManager/tokenManager.sol
function interestUpdateReward() external override returns (bool)
{
    uint256 thisBlock = block.number;
    uint256 interestRewardUpdated = dataStorageInstance.getInterestRewardUpdated();
    uint256 delta = thisBlock - interestRewardUpdated;
    if (delta == 0)
    {
        return false;
    }

    dataStorageInstance.setInterestRewardUpdated(thisBlock);
    for (uint256 handlerID; handlerID < tokenHandlerLength; handlerID++)
    {
        // interestUpdate
    }
    uint256 globalRewardPerBlock = dataStorageInstance.getInterestUpdateRewardPerblock();
    uint256 rewardAmount = delta.mul(globalRewardPerBlock);

    /* transfer reward tokens */
    return _rewardTransfer(msg.sender, rewardAmount);
}
```

The *interestUpdateReward* is a publicly accessible function that is called by an external user to update the interest rate of each token serviced by the BiFi contracts and receive incentive tokens as a reward. The reward for the user is calculated as: **(current block.number - block.number of the previous reward) * rewardPerBlock**.

```
// marketManager/managerDataStorage/managerDataStorage.sol
```



```
constructor () public
{
    owner = msg.sender;
    globalRewardPerBlock = 0x478291c1a0e982c98;
    globalRewardDecrement = 0x7ba42eb3bfc;
    globalRewardTotalAmount = (4 * 100000000) * (10 ** 18);
    alphaRate = 2 * (10 ** 17);
    alphaLastUpdated = block.number;
    /*
            rewardParamUpdateRewardPerBlock = 1u * (10u ** 17u); // == 0.1
            rewardParamUpdated = block.getNumber();

            interestUpdateRewardPerblock = 5u * (10u ** 16u); // == 0.05
            interestRewardLastUpdated = block.getNumber();
            */
}
```

However, the `interestUpdateRewardPerblock` and `interestRewardLastUpdated` variables are not initialized in the constructor. Instead, they are updated separately by the owner during deployment with calls to the appropriate setters. There may exist a race condition where a user who calls the `interestUpdateReward` function after `setInterestUpdateRewardPerblock` is called by the deployer can get **(block.number at the time of the function call - 0) * rewardPerBlock** amount of incentive tokens, unless the deployer first initializes `interestRewardLastUpdated`.

The attack scenario is:

- After `managerDataStorage` is newly deployed, the administrator initializes the `interestUpdateRewardPerblock` variable to 0.05 through the setter.
- The attacker calls the `interestUpdateReward` function.
- *(current block.number - 0)*0.05* amounts of reward can be received. At 2021-02-03:12:00:00, attacker can get 589036.65 tokens.

We recommend that the `managerDataStorage` constructor initializes the `interestRewardLastUpdated` variable to the current block number. The `interestUpdateRewardPerblock` already defaults to zero and does not need to be initialized.

For example:

```
// marketManager/managerDataStorage/managerDataStorage.sol
constructor () public
```

```
{
    owner = msg.sender;
    globalRewardPerBlock = 0x478291c1a0e982c98;
    globalRewardDecrement = 0x7ba42eb3bfc;
    globalRewardTotalAmount = (4 * 100000000) * (10 ** 18);
    // ...
    setInterestRewardUpdated(block.number); // add setter
}
```

Note that we do not have any indication that the current contracts were deployed insecurely.

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	Reduce code duplication between coinHandler/coinSI and tokenHandler/tokenSI	Code quality	Minor
2	Use base class instead of duplicating storage variables across implementation contracts	Code quality	Minor
3	Update global EXR at most once per block	Correctness Optimization	Minor
4	Use SafeERC20 from OpenZeppelin	Code quality	Minor
5	Remove onlyMarketManager modifier on proxy contract external functions	Optimization	Minor
6	Cache result of getBetaRate in updateRewardLane	Optimization	Minor

Recommendation #1: Reduce code duplication between coinHandler/coinSI and tokenHandler/tokenSI

A significant amount of logic is defined in the coinHandler contract and is duplicated in the tokenHandler contract. This logic could be encapsulated in an abstract base contract, e.g. baseHandler, which is inherited by coinHandler and tokenHandler. The same principle would apply to coinSI and tokenSI as well. This would make it easy to fix bugs in these contracts and avoid duplicating work during source code audits.

To handle deposits of ether, `deposit` and `reserveDeposit` could be simple wrappers that call internal functions, e.g. `_internalDeposit` and `_internalReserveDeposit`, with appropriate arguments. Alternatively, you could use the WETH contract to wrap ether in an ERC20 token.

Recommendation #2: Use base class instead of duplicating storage variables across implementation contracts

The proxy pattern requires each of the implementation contracts to contain the same storage variables with the same types in the same order. Otherwise, the storage variables may be assigned different slots and overwrite each other. In the current code, the storage variables are copied to each individual implementation contract. A better design would be to create a class that each implementation contract inherits which defines the storage variables. This avoids the risk of forgetting to update the source code of an implementation contract or accidentally creating storage variables that overwrite each other.

Recommendation #3: Update global EXR at most once per block

For every call into the BiFi contract within a block that calls `_calcInterestAmount`, the global borrow and deposit exchange rates are recalculated. The action exchange rates and block delta values are fixed, but the total deposit amount and total borrow amount may change between these successive calls which changes the interest rate that is calculated and multiplied by the block delta. This results in the global exchange rates changing within one transaction which may not be the intended behavior.

One possible improvement is to move the code to update the global exchange rates from `_calcInterestAmount` into a new function, e.g. `_calcGlobalInterestAmount`, which is called from `_checkFirstAction` if `blockDelta` is greater than zero. Then, `_calcInterestAmount` would only need to update the user exchange rate. This would also allow you to remove the action exchange rate variables since they are only used in the calculation of the global exchange rate.

Recommendation #4: Use SafeERC20 from OpenZeppelin

ERC20 tokens are not always ERC20 compliant. Some tokens will return a boolean that indicates success or failure. Some tokens will throw on failure and return nothing on success. The SafeERC20 library is the de facto standard for handling these cases correctly, and it will throw on failure in both cases.

Recommendation #5: Remove onlyMarketManager modifier on proxy contract external functions

The proxy contract exposes two sets of functions to call into its implementation contracts: *`handlerProxy`* / *`handlerViewProxy`* and *`siProxy`* / *`siViewProxy`*. Currently, the view proxy functions are exactly the same as the non-view proxy functions, except without the *`onlyMarketManager`* modifier. As such, the *`onlyMarketManager`* modifier on those proxy functions are not providing any additional security and it uses additional gas (to load the *`marketManager`* variable) every time the market manager calls into a token handler or service incentive function.

In the future, if there is a functional difference between the proxy view and non-view external functions, then it may make sense to keep the *`onlyMarketManager`* modifier as a security precaution.

Recommendation #6: Cache result of getBetaRate in updateRewardLane

In *`updateRewardLane`*, the beta rate is needed when calculating either the market rate or the user's rate. Currently, if the market rate needs to be updated, the beta rate will be retrieved from the data storage twice. Gas could be saved by retrieving the beta rate only once at the top of the function, if it will be needed.

Appendix: External APIs

etherManager (tokenManager.sol)

applyInterestHandlers	**external**	
interestUpdateReward	**external**	
rewardClaimAll	**external**	
rewardUpdateOfInAction	**external**	
updateRewardParams	**external**	
setCircuitBreaker	external	onlyBreaker
partialLiquidationUser	external	onlyLiquidationManager
partialLiquidationUserReward	external	onlyLiquidationManager
handlerRegister	external	onlyOwner
ownerRewardTransfer	external	onlyOwner
ownershipTransfer	public	onlyOwner
setBreakerTable	external	onlyOwner
setHandlerSupport	public	onlyOwner
setLiquidationManager	external	onlyOwner
setOracleProxy	external	onlyOwner
setRewardErc20	public	onlyOwner
setTokenHandlersLength	external	onlyOwner
getCircuitBreaker	external	view
getGlobalRewardInfo	external	view
getMaxLiquidationReward	external	view
getOwner	public	view
getRewardErc20	public	view
getTokenHandlerBorrowLimit	external	view
getTokenHandlerID	external	view
getTokenHandlerInfo	external	view
getTokenHandlerMarginCallLimit	external	view
getTokenHandlerPrice	external	view
getTokenHandlersLength	external	view
getTokenHandlerSupport	external	view
getUserCollateralizableAmount	external	view

getUserExtraLiquidityAmount	external	view
getUserIntraHandlerAssetWithInterest	external	view
getUserLimitIntraAsset	external	view
getUserTotalIntraCreditAsset	external	view

managerDataStorage (managerDataStorage.sol)

setGlobalRewardDecrement	external	onlyManager
setGlobalRewardPerBlock	external	onlyManager
setGlobalRewardTotalAmount	external	onlyManager
setInterestRewardUpdated	external	onlyManager
setLiquidationManagerAddr	external	onlyManager
setRewardParamUpdated	external	onlyManager
setTokenHandler	external	onlyManager
setTokenHandlerSupport	external	onlyManager
ownershipTransfer	public	onlyOwner
setAlphaLastUpdated	external	onlyOwner
setAlphaRate	external	onlyOwner
setInterestUpdateRewardPerblock	external	onlyOwner
setManagerAddr	external	onlyOwner
setRewardParamUpdateRewardPerBlock	external	onlyOwner
setTokenHandlerAddr	external	onlyOwner
setTokenHandlerExist	external	onlyOwner
getAlphaLastUpdated	external	view
getAlphaRate	external	view
getGlobalRewardDecrement	external	view
getGlobalRewardPerBlock	external	view
getGlobalRewardTotalAmount	external	view
getInterestRewardUpdated	external	view
getInterestUpdateRewardPerblock	external	view
getLiquidationManagerAddr	external	view
getOwner	public	view
getRewardParamUpdated	external	view
getRewardParamUpdateRewardPerBlock	external	view

getTokenHandlerAddr	external	view
getTokenHandlerExist	external	view
getTokenHandlerID	external	view
getTokenHandlerInfo	external	view
getTokenHandlerSupport	external	view

tokenHandler (tokenHandler.sol)

applyInterest	**external**	
borrow	**external**	
deposit	**external**	
repay	**external**	
reserveDeposit	**external**	
withdraw	**external**	
setCircuitBreaker	external	onlyMarketManager
partialLiquidationUser	external	onlyMarketManager
partialLiquidationUserReward	external	onlyMarketManager
checkFirstAction	external	onlyMarketManager
setCircuitBreakWithOwner	external	onlyOwner
ownershipTransfer	external	onlyOwner
reserveWithdraw	external	onlyOwner
setTokenHandlerBorrowLimit	external	onlyOwner
setTokenHandlerMarginCallLimit	external	onlyOwner
setUnifiedTokenDecimal	external	onlyOwner
setUnderlyingTokenDecimal	external	onlyOwner
setMarketManager	public	onlyOwner
setInterestModel	public	onlyOwner
setHandlerDataStorage	public	onlyOwner
setErc20	public	onlyOwner
setSiHandlerDataStorage	public	onlyOwner
getTokenName	external	view
getTokenHandlerLimit	external	view
getTokenHandlerMarginCallLimit	external	view
getTokenHandlerBorrowLimit	external	view

getUserMaxBorrowAmount	external	view
getUserMaxWithdrawAmount	external	view
getSIRandBIR	external	view
getUserMaxRepayAmount	external	view
getERC20Addr	external	view
getUserAmount	external	view
getUserIntraDepositAmount	external	view
getUserIntraBorrowAmount	external	view
getDepositTotalAmount	external	view
getBorrowTotalAmount	external	view
getUserAmountWithInterest	external	view
getTokenDecimals	external	view
getUnifiedTokenDecimal	external	view
getUnderlyingTokenDecimal	external	view
getLimitOfAction	external	view
getOwner	public	view
getSiHandlerDataStorage	public	view
getMarketManagerAddr	public	view
getInterestModelAddr	public	view
getHandlerDataStorageAddr	public	view
getErc20Addr	public	view

marketHandlerDataStorage (handlerDataStorage.sol)

addBorrowAmount	external	onlyBifiContract
addBorrowTotalAmount	external	onlyBifiContract
addDepositAmount	external	onlyBifiContract
addDepositTotalAmount	external	onlyBifiContract
addReservedAmount	external	onlyBifiContract
addUserIntraBorrowAmount	external	onlyBifiContract
addUserIntraDepositAmount	external	onlyBifiContract
setActionEXR	external	onlyBifiContract
setAmount	external	onlyBifiContract
setBlocks	external	onlyBifiContract

setCircuitBreaker	external	onlyBifiContract
setEXR	external	onlyBifiContract
setInactiveActionDelta	external	onlyBifiContract
setLastUpdatedBlock	external	onlyBifiContract
setNewCustomer	external	onlyBifiContract
setUserAccessed	external	onlyBifiContract
setUserEXR	external	onlyBifiContract
subBorrowAmount	external	onlyBifiContract
subBorrowTotalAmount	external	onlyBifiContract
subDepositAmount	external	onlyBifiContract
subDepositTotalAmount	external	onlyBifiContract
subReservedAmount	external	onlyBifiContract
subUserIntraBorrowAmount	external	onlyBifiContract
subUserIntraDepositAmount	external	onlyBifiContract
syncActionEXR	external	onlyBifiContract
updateSignedReservedAmount	external	onlyBifiContract
ownershipTransfer	public	onlyOwner
setBorrowLimit	external	onlyOwner
setCoinHandler	external	onlyOwner
setInterestModelAddr	external	onlyOwner
setLimitOfAction	external	onlyOwner
setLiquidityLimit	external	onlyOwner
setLiquiditySensitivity	external	onlyOwner
setMarginCallLimit	external	onlyOwner
setMarketHandlerAddr	external	onlyOwner
setMinimumInterestRate	external	onlyOwner
setReservedAddr	external	onlyOwner
setTokenHandler	external	onlyOwner
getActionEXR	external	view
getAmount	external	view
getBorrowLimit	external	view
getBorrowTotalAmount	external	view
getDepositTotalAmount	external	view

getGlobalBorrowEXR	external	view
getGlobalDepositEXR	external	view
getGlobalEXR	external	view
getHandlerAmount	external	view
getInactiveActionDelta	external	view
getInterestModelAddr	external	view
getLastUpdatedBlock	external	view
getLimit	external	view
getLimitOfAction	external	view
getLiquidityLimit	external	view
getLiquiditySensitivity	external	view
getMarginCallLimit	external	view
getMarketHandlerAddr	external	view
getMinimumInterestRate	external	view
getOwner	public	view
getReservedAddr	external	view
getReservedAmount	external	view
getUserAccessed	external	view
getUserAmount	external	view
getUserEXR	external	view
getUserIntraBorrowAmount	external	view
getUserIntraDepositAmount	external	view

tokenSI (tokenSI.sol)

updateRewardLane	**external**	
claimRewardAmountUser	external	onlyMarketManager
setCircuitBreaker	external	onlyMarketManager
updateRewardPerBlockLogic	external	onlyMarketManager
ownershipTransfer	external	onlyOwner
setCircuitBreakWithOwner	external	onlyOwner
getBetaRate	external	view
getBetaRateBaseTotalAmount	external	view
getBetaRateBaseUserAmount	external	view

getMarketRewardInfo	external	view
getUserRewardInfo	external	view

marketSIHandlerDataStorage (marketSIHandlerDataStorage.sol)

ownershipTransfer	external	onlyOwner
setBetaRate	external	onlyOwner
setSIHandlerAddr	public	onlyOwner
setCircuitBreaker	external	onlySIHandler
setMarketRewardInfo	external	onlySIHandler
setUserRewardInfo	external	onlySIHandler
updateRewardPerBlockStorage	external	onlySIHandler
getBetaRate	external	view
getMarketRewardInfo	external	view
getRewardInfo	external	view
getSIHandlerAddr	public	view
getUserRewardInfo	external	view

etherLiquidationManager (liquidationManager.sol)

partialLiquidation	**external**	
setCircuitBreaker	external	onlyManager
ownershipTransfer	public	onlyOwner
setMarketManagerAddr	external	onlyOwner
checkLiquidation	external	view
getOwner	public	view

interestModel (interestModel.sol)

ownershipTransfer	external	onlyOwner
getInterestAmount	external	view
getOwner	public	view
getSIRandBIR	external	view
viewInterestAmount	external	view

proxy (reqTokenProxy.sol)

borrow	**public**	
deposit	**public**	
handlerViewProxy	**external**	
repay	**public**	
siViewProxy	**external**	
withdraw	**public**	
handlerProxy	external	onlyMarketManager
siProxy	external	onlyMarketManager
initialize	public	onlyOwner
migration	public	onlyOwner
ownershipTransfer	external	onlyOwner
setHandlerAddr	public	onlyOwner
setHandlerID	public	onlyOwner
setSiHandlerAddr	public	onlyOwner
getHandlerAddr	public	view
getHandlerID	public	view
getSiHandlerAddr	public	view

proxy (reqCoinProxy.sol)

borrow	**public**	
deposit	**public**	
handlerViewProxy	**external**	
repay	**public**	
siViewProxy	**external**	
withdraw	**public**	
handlerProxy	external	onlyMarketManager
siProxy	external	onlyMarketManager
initialize	public	onlyOwner
migration	public	onlyOwner
ownershipTransfer	external	onlyOwner
setHandlerAddr	public	onlyOwner

setHandlerID	public	onlyOwner
setSiHandlerAddr	public	onlyOwner
getHandlerAddr	public	view
getHandlerID	public	view
getSiHandlerAddr	public	view

oracleProxy (oracleProxy.sol)

ownershipTransfer	public	onlyOwner
setOracleFeed	external	onlyOwner
getOracleFeed	external	view
getOwner	public	view
getTokenPrice	external	view