CoFi Money Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

COFI Money is on a mission to transform idle assets into fully composable, yield-bearing tokens, that actively work for you and accrue rewards. COFI Money provides a seamless, transparent, secure, compliant and non-custodial solution.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	CoFi Money
Repository	cofi-ethereum
Commit	1418c2809662...
Date	May 2023

Issues Found

Severity	Count
Critical Risk	4
High Risk	3
Medium Risk	4
Low Risk	5
Informational	0
Gas Optimizations	0
Total Issues	**16**

Summary of Findings

Title	Status
[C-1] Anyone can burn tokens of other users	Resolved
[C-2] SwapExactInputSingle lacks slippage control	Resolved
[C-3] Vault.sol is susceptible to inflation attack	Resolved
[C-4] FiToShares and underlyingToFi can be abused to steal user funds	Resolved
[H-1] Missing expiration deadline in swap functions can lead to unfavorable prices and financial losses	Resolved
[H-2] Missing token transfer functions in swapUsdcForDai and swapDaiForUsdc in MakerRouterFacet.sol	Resolved
[H-3] Incorrect storage check in isRedeemEnabled	Resolved
[M-1] TransferFrom uses allowance even if spender == from	Resolved
[M-2] Unrestricted address opt-in vulnerability	Resolved
[M-3] Both functions in MakerRouterFacet will revert due to insufficient balance	Resolved
[M-4] Incorrect deduction from allowance without considering infinite allowance in transferFrom function	Resolved
[L-1] Hardcoded Addresses wont work for multi-chain deployments	Resolved

Summary of Findings

Title	Status
[L-2] Potential self-removal of admin account in toggleAdmin function	Resolved
[L-3] GPv2SafeERC20 safeTransfer functions can return true for addresses that do not exist	Resolved
[L-4] Missing rebase opt-out check in toggleFreeze function	Resolved
[L-5] Unsafe use of Address.isContract in isNonRebasingAccount function	Resolved

5 Findings

5.1 Critical Risk

[C-1] Anyone can burn tokens of other users

Context: FiToken.sol

Impact: This vulnerability allows arbitrary users to burn tokens from any account, potentially leading to unauthorized loss of assets.

Description: Based on the provided code, it seems that anyone can call the `burn` function and specify any `account` address to burn tokens from. In a secure system, only the account holder should have the ability to burn their own tokens. This creates a vulnerability where malicious users can burn other users' tokens without their permission.

```
    function burn(address account, uint256 amount) external {
        require(!paused, 'FiToken: Token paused');
        _burn(account, amount);
    }
```

My assumption is this code has been forked from here, but in this scenario it's access control is limited only to the vault, whereas in the Cofi's context, anyone can call the function.

Here's a simple Proof of Concept (PoC):

```
pragma solidity >=0.5.0 <0.8.0;


contract Attack {
    FiToken public tokenContract;

    constructor(YourTokenContract _tokenContract) public {
        tokenContract = _tokenContract;
    }

    function burnTokens(address victim, uint256 amount) public {
        tokenContract.burn(victim, amount);
    }
}
```

In this PoC, we define a contract `Attack` that accepts the address of your token contract in its constructor. This contract has a function `burnTokens` that accepts an `address` and an `uint256` as parameters, representing the victim's address and the amount of tokens to burn, respectively.

This function simply calls the `burn` function of your token contract with the victim's address and the amount as parameters, effectively burning tokens from the victim's account without their consent.

Recommendation: To circumvent this vulnerability, it is advisable to revise the `burn` function to only permit the transaction's sender (the `msg.sender`) to burn their own tokens. This ensures that only the token holder can burn their tokens, thwarting other users from burning tokens that they do not own.

```
function burn(uint256 amount) external {
    require(!paused, 'FiToken: Token paused');
    _burn(msg.sender, amount);
}
```

Should you wish to maintain the functionality of third-party account burns, consider introducing a separate function with restricted access control or implement similar functionality but through the use of signatures.

[C-2] SwapExactInputSingle lacks slippage control

Context: SwapRouterFacet.sol

Impact

The current configuration of the `swapExactInputSingle` function in the `SwapRouterFacet` contract could lead to considerable financial losses due to a lack of slippage protection. This absence of slippage protection could open the door for price manipulation by MEV (Miner Extractable Value) bots, which could perform "sandwich" attacks to their advantage and the users' disadvantage.

Description

The function `swapExactInputSingle` is meant for token swapping operations. In its current state, it sets the `amountOutMinimum` to 0, indicating no minimum limit on the tokens the transaction should yield. This absence of a lower limit exposes the transaction to potential manipulation. Adversaries could artificially inflate the price before the transaction, profit from it, and then quickly reverse their actions, leading to substantial losses.

```
ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
    .ExactInputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        fee: poolFee,
        recipient: address(this),
        deadline: block.timestamp,
        amountIn: amountIn,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    });
```

Recommendation

To mitigate this vulnerability, it is recommended to include a parameter for `amountOutMinimum` in the `swapExact-InputSingle` function. This parameter would specify the minimum number of tokens expected from the swap operation, providing a slippage protection mechanism and hindering any manipulative efforts by bots.

```
 function swapExactInputSingle(
      uint256 amountIn,
+       uint256 amountOutMinimum,
      address tokenIn,
      address tokenOut,
      uint24  poolFee
  )
      external
      returns (uint256 amountOut)
  {
      IERC20(tokenIn).approve(address(swapRouter), amountIn);

      ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
          .ExactInputSingleParams({
              tokenIn: tokenIn,
              tokenOut: tokenOut,
              fee: poolFee,
              recipient: address(this),
              deadline: block.timestamp,
              amountIn: amountIn,
+               amountOutMinimum: amountOutMinimum,
              sqrtPriceLimitX96: 0
          });

      amountOut = swapRouter.exactInputSingle(params);
  }
```

[C-3] Vault.sol is susceptible to inflation attack

Context: Vault.sol

Impact: The vulnerability allows an attacker to manipulate the rate curve of the vault, leading to rounding errors in the calculation of shares for depositors. This can result in depositors losing a significant portion or the entirety of their deposited assets. The effectiveness of the attack depends on the offset between the precision of shares and assets, with smaller offsets making the attack more profitable for the attacker.

Description: The vulnerability arises due to the lack of a sufficient offset between the precision of shares and assets in the vault. The code description provides insights into the vulnerability but does not include any code blocks.

The attack can be summarized as follows:

1. The exchange rate of the vault is determined by the number of assets and shares held in the vault.
2. Depositing tokens results in the number of shares being rounded down, causing a loss of value for the depositor.
3. A malicious attacker can exploit this rounding error by donating assets to the vault, manipulating the rate curve to favor themselves.
4. By carefully depositing a small amount of tokens followed by a large "donation" to the vault, the attacker can shift the exchange rate, causing subsequent deposits to result in significantly fewer shares for depositors.
5. If the rounding error reduces the depositor's shares to 0, their entire deposit effectively becomes a donation to the vault, benefiting the attacker.
6. The attacker typically waits for a legitimate user to deposit into the vault and front-runs their transaction, executing the attack described above.

The vulnerability allows the attacker to steal a significant portion of deposited tokens, particularly when the offset between share and asset precision is minimal. The attack can be customized to target a specific fraction of a

user's deposit, with the attacker's required commitment being equivalent to their potential earnings.

Recommendation: It is strongly recommended to update the smart contract to the latest version provided by OpenZeppelin (OZ) where they have fixed this vulnerability. By using the latest version, you can ensure that the deposit function correctly calculates the shares and mitigates the risk of an inflation attack.

https://github.com/OpenZeppelin/openzeppelin-contracts/releases/tag/v4.9.0-rc.0
https://github.com/OpenZeppelin/openzeppelin-contracts/pull/3979

[C-4] FiToShares and underlyingToFi can be abused to steal user funds

Context SupplyFacet.sol

Impact

This vulnerability allows an unauthorized user to execute the redeem operation in the `fiToShares` and `underlyingToFi` functions for any other whitelisted customer, potentially stealing their deposits, because the redeem operation in the `FiToken` contract skips the approval check and is executed directly, enabling an attacker to perform unauthorized redemptions.

Description

`underlyingToFi` allows conversion of a specified amount of underlying assets to a fiAsset, with the converted assets transferred from a given account to a recipient, ensuring that the minimum required amount of fiAssets (before fees) is received. It also verifies if minting is enabled for the specified fiAsset and captures any applicable minting fees, transferring them to a fee collector address. Finally, it mints the resulting fiAssets and transfers them to the recipient.

```
function underlyingToFi(
    uint256 amount,
    uint256 minAmountOut, // E.g., 1,000 * 0.9975 = 997.50. Auto-set to 0.25%.
    address fiAsset,
    address depositFrom,
    address recipient
)
    external
    isWhitelisted
    minDeposit(amount, fiAsset) // -> 0
    returns (uint256 mintAfterFee)
{
    require(
        LibToken._isMintEnabled(fiAsset) == 1,
        'SupplyFacet: Mint for token disabled'
    );

    uint256 assets = LibVault._getAssets(
        // Add permit for Vault transfer.
        LibVault._wrap(
            amount,
            s.vault[fiAsset],
            depositFrom
        ),
        s.vault[fiAsset]
    );

    require(assets >= minAmountOut, 'SupplyFacet: Slippage exceeded');

    uint256 fee = LibToken._getMintFee(fiAsset, assets);
    mintAfterFee = assets - fee;

    // Capture mint fee in fiAssets.
    if (fee > 0) {
        LibToken._mint(fiAsset, s.feeCollector, fee);
```

```
        emit LibToken.MintFeeCaptured(fiAsset, fee);
    }

    LibToken._mint(fiAsset, recipient, mintAfterFee);
}
```

On the other hand, `fiToShares` allows the redemption of a specified amount of fiAssets to their corresponding collateral yieldAsset, transferring the redeemed yieldAssets to a recipient while ensuring the minimum required amount of yieldAssets (after fees) is received. It verifies if redemption is enabled for the specified fiAsset, calculates and captures any applicable redemption fees, and transfers the remaining fiAssets to a fee collector address. Finally, it converts the redeemed fiAssets into yieldAssets (shares) based on the corresponding vault ratio and transfers them to the recipient.

```
function fiToShares(
        uint256 amount,
        uint256 minAmountOut,
        address fiAsset,
        address depositFrom,
        address recipient
    )   external
        isWhitelisted
        minWithdraw(amount, fiAsset)
        returns (uint256 burnAfterFee)
    {
        require(
            LibToken._isRedeemEnabled(fiAsset) == 1,
            'SupplyFacet: Redeem for token disabled'
        );

        // Redeem operation in FiToken contract skips approval check.
        LibToken._redeem(fiAsset, depositFrom, amount);

        uint256 fee = LibToken._getRedeemFee(fiAsset, amount);
        burnAfterFee = amount - fee;

        // Redemption fee is captured by retaining 'fee' amount.
        LibToken._burn(fiAsset, s.feeCollector, burnAfterFee);
        if (fee > 0) {
            emit LibToken.RedeemFeeCaptured(fiAsset, fee);
        }

        uint256 shares = LibVault._getShares(burnAfterFee, s.vault[fiAsset]);
        require(shares >= minAmountOut, 'SupplyFacet: Slippage exceeded');

        LibToken._transfer(s.vault[fiAsset], shares, recipient);
    }
```

If a user want's to call both functions, he needs to be whitelisted. To get whitelisted, the user has to go through a KYC process. Once that successfully executes the protocol whitelists their address

The KYC works by having the user sign a tx to confirm they are the wallet owner. Then routed to the protocol's KYC provider Sumsub who does the check. Upon successful response a webhook triggers that executes toggleWhitelist().

The issue here lies that as noted in both functions, the redeem operation in FiToken skips the approval check and is executed directly.

```
function _redeem(
    address fiAsset,
    address from,
    uint256 amount
) internal {
    AppStorage storage s = LibAppStorage.diamondStorage();

    IFiToken(fiAsset).redeem(from, s.feeCollector, amount);
}
```

This means, a whitelisted user can call `fiToShares` and `underlyingToFi` for **any other** whitelisted customer and steal their deposit without their consent.

Recommendation

To mitigate this vulnerability, it is recommended to refactor the `fiToShares` and `underlyingToFi` functions to redeem from the `msg.sender` instead of `depositFrom`. By redeeming from the `msg.sender`, the execution will be limited to the authorized user who initiated the transaction. If the protocol intends to enable third-party redeeming, it should implement a mechanism that uses signatures to validate the authorization, in addition to the existing KYC process.

Implementing these changes will ensure that the redeem operation can only be executed by the authorized user and prevent unauthorized access to other users' deposits.

5.2 High Risk

[H-1] Missing expiration deadline in swap functions can lead to unfavorable prices and financial losses

Context: SwapRouterFacet.sol

Impact

The current implementation of the swap functions in the given code snippets does not provide an effective expiration deadline mechanism for transactions. This absence of a proper deadline could lead to transactions remaining unexecuted in the mempool for extended periods, potentially resulting in unfavorable prices for users and exposing them to financial losses.

Description

The `ISwapRouter.ExactOutputSingleParams` and `ISwapRouter.ExactInputSingleParams` configurations in the provided code set the deadline parameter to `block.timestamp`. This approach does not offer adequate protection, as a validator can hold the transaction, and the block it is eventually put into will have a `block.timestamp` value, rendering the deadline ineffective.

Automated Market Makers (AMMs) and other advanced protocols often allow users to set a deadline parameter to enforce a time limit for transaction execution. Without a proper deadline, the transaction could linger in the mempool and be executed at a much later time, potentially resulting in a worse price for the user. 1, 2

```
ISwapRouter.ExactOutputSingleParams memory params = ISwapRouter
    .ExactOutputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        fee: poolFee,
        recipient: address(this),
        deadline: block.timestamp,
        amountOut: amountOut,
        amountInMaximum: amountInMaximum,
        sqrtPriceLimitX96: 0
    });
```

```
ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
    .ExactInputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        fee: poolFee,
        recipient: address(this),
        deadline: block.timestamp,
        amountIn: amountIn,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    });
```

Recommendation

It is recommended to modify the protocol to allow users interacting with AMMs to set expiration deadlines. This change would provide better protection against transaction delays and help prevent potential financial losses due to unfavorable execution prices.

To implement this recommendation, consider updating the code as follows:

1. Add a user-specified `uint256 deadline` parameter to the functions that involve AMM interactions.
2. Replace the current `block.timestamp` values in the `ISwapRouter.ExactOutputSingleParams` and `ISwapRouter.ExactInputSingleParams` configurations with the new `deadline` parameter.

After making these modifications, thoroughly test the updated code to ensure that it works as intended and that no additional vulnerabilities or unintended side effects are introduced.

[H-2] Missing token transfer functions in swapUsdcForDai and swapDaiForUsdc in MakerRouterFacet.sol

Context: MakerRouterFacet.sol

Impact:

The absence of token transfer functions in the `swapUsdcForDai` and `swapDaiForUsdc` functions can lead to potential vulnerabilities and unintended behavior. Tokens received or generated during the swapping process are not properly handled, potentially resulting in loss of funds or incorrect token balances.

Description:

In the provided code snippet, the `swapUsdcForDai` and `swapDaiForUsdc` functions have comments indicating that the corresponding tokens should be transferred to the specified vault. However, no actual token transfer functions are implemented, and the comments themselves do not have any impact on the execution of the code.

As a result, any tokens received or generated during the swapping process are not being transferred to the intended vault or any other designated address. This can lead to a situation where tokens remain in the contract's address, which can cause confusion, loss of funds, or inconsistent token balances.

Recommendation:

To address this vulnerability, you should implement the necessary token transfer functions within the `swapUsdcForDai` and `swapDaiForUsdc` functions. Below is an example of how you can modify the code to include the token transfers:

```
function swapUsdcForDai(uint256 amountUsdcIn) external onlyAdmin {
    require(daiPSM.tin() == 0, "MakerRouter: maker fee not 0");

    // First ensure USDC resides in this address.
    USDC.safeApprove(GEM_JOIN, amountUsdcIn); /// approve DAI PSM to spend USDC
    daiPSM.sellGem(address(this), amountUsdcIn); /// sell USDC for DAI
    // Deploy DAI to new Vault.
    DAI.transfer(address(newVault), DAI.balanceOf(address(this))); // Transfer DAI to new vault
}
```

```
function swapDaiForUsdc(uint256 amountDaiIn) external onlyAdmin {
    require(daiPSM.tout() == 0, "MakerRouter: maker fee not 0");

    // First ensure USDC resides in this address.
    DAI.safeApprove(address(daiPSM), amountDaiIn); /// approve DAI PSM to spend DAI
    daiPSM.buyGem(address(this), amountDaiIn / USDC_SCALING_FACTOR); /// sell DAI for USDC
    // Deploy USDC to new Vault.
    USDC.transfer(address(newVault), USDC.balanceOf(address(this))); // Transfer USDC to new vault
}
```

[H-3] Incorrect storage check in isRedeemEnabled

Context: LibToken.sol

Impact

The impact of this vulnerability is that the `_isRedeemEnabled` function is checking the wrong storage variable, which may lead to incorrect behavior when determining if redeeming is enabled for a specific `fiAsset`. This can potentially result in unintended consequences and erroneous decisions based on the incorrect information.

Description

The `_isRedeemEnabled` function in the provided code snippet is designed to access the storage and determine whether redeeming is enabled for a particular `fiAsset`. However, instead of checking the correct storage variable `s.redeemEnabled`, the function erroneously checks `s.mintEnabled`. This incorrect check may produce misleading results, as it retrieves the value associated with `fiAsset` from the wrong storage mapping.

```
function _isRedeemEnabled(address fiAsset) internal view returns (uint8) {
    AppStorage storage s = LibAppStorage.diamondStorage();

    return s.mintEnabled[fiAsset];
}
```

Recommendation

To address this vulnerability, it is recommended to modify the code in the `_isRedeemEnabled` function to check the correct storage variable, `s.redeemEnabled`, for determining redeeming status. The code should be updated as follows:

```
function _isRedeemEnabled(address fiAsset) internal view returns (uint8) {
    AppStorage storage s = LibAppStorage.diamondStorage();

    return s.redeemEnabled[fiAsset];
}
```

5.3 Medium Risk

[M-1] TransferFrom uses allowance even if spender == from

Context: FiToken.sol

Impact: The vulnerability in `transferFrom` function can lead to tokens becoming irreversibly stranded across different protocols. It breaks compatibility with a significant number of protocols that exclusively rely on the `transferFrom` method for token transfers (pull-only approach) instead of using both `transfer` and `transferFrom` (push and pull).

Description: The `transferFrom` function in `FiToken` attempts to use the allowance even when the `spender` is the same as the `from` address. Consequently, the tokens involved in such transactions may become irreversibly stranded across different protocols due to the inconsistency in behavior.

Here is the relevant code block demonstrating the vulnerability:

```
require(!frozen[_from], 'FiToken: Sender account is frozen');
    require(!frozen[_to], 'FiToken: Recipient account is frozen');

    _allowances[_from][msg.sender] = _allowances[_from][msg.sender].sub(
        _value
    );

    _executeTransfer(_from, _to, _value);
```

Recommendation: To mitigate this vulnerability and ensure compatibility with other protocols, it is recommended to adjust the behavior of the `transferFrom` function by modifying the code block to include an additional condition to check if `spender` is equal to `from`:

```
if (_from != msg.sender) {
    _allowances[_from][msg.sender] = _allowances[_from][msg.sender].sub(
        _value
        );
}
```

[M-2] Unrestricted address opt-in vulnerability

Context: FiToken.sol

Impact: The lack of access control in the FiToken.rebaseOptInExternal function allows anyone to opt in addresses on behalf of other users without their consent. This poses a significant security risk and violates the principle of user autonomy and control over their own accounts.

Description: The FiToken.rebaseOptInExternal function, as currently implemented, lacks any access control mechanism. This means that any user can invoke the function and opt in an address of their choice, even if they do not have the authorization to do so. This unrestricted access can lead to unauthorized opt-ins and potential misuse of other users' accounts.

Here is the relevant code snippet:

```
function rebaseOptInExternal(address _account) public nonReentrant {
    require(_isNonRebasingAccount(_account), 'FiToken: Account has not opted out');

    // Convert balance into the same amount at the current exchange rate
    uint256 newCreditBalance = _creditBalances[_account]
        .mul(_rebasingCreditsPerToken)
        .div(_creditsPerToken(_account));

    // Decreasing non rebasing supply
    nonRebasingSupply = nonRebasingSupply.sub(balanceOf(_account));

    _creditBalances[_account] = newCreditBalance;

    // Increase rebasing credits, totalSupply remains unchanged so no
    // adjustment necessary
    _rebasingCredits = _rebasingCredits.add(_creditBalances[_account]);

    rebaseState[_account] = RebaseOptions.OptIn;

    // Delete any fixed credits per token
    delete nonRebasingCreditsPerToken[_account];
}
```

control mechanism. This means that any user can invoke the function and opt in an address of their choice, even if they do not have the authorization to do so. This unrestricted access can lead to unauthorized opt-ins and potential misuse of other users' accounts.

Here is the relevant code snippet:

```
function rebaseOptInExternal(address _account) public nonReentrant {
    require(_isNonRebasingAccount(_account), 'FiToken: Account has not opted out');

    // Convert balance into the same amount at the current exchange rate
    uint256 newCreditBalance = _creditBalances[_account]
        .mul(_rebasingCreditsPerToken)
        .div(_creditsPerToken(_account));

    // Decreasing non rebasing supply
    nonRebasingSupply = nonRebasingSupply.sub(balanceOf(_account));

    _creditBalances[_account] = newCreditBalance;

    // Increase rebasing credits, totalSupply remains unchanged so no
    // adjustment necessary
    _rebasingCredits = _rebasingCredits.add(_creditBalances[_account]);

    rebaseState[_account] = RebaseOptions.OptIn;

    // Delete any fixed credits per token
    delete nonRebasingCreditsPerToken[_account];
}
```

This code block allows any caller to execute the rebaseOptInExternal function, opting in an address of their choice without proper authorization. The function performs various operations related to rebasing and modifies the internal state of the contract, impacting the balance and credit calculations for the specified account.

However, this unrestricted access to opting in other addresses violates user expectations and can lead to unauthorized actions, potentially disrupting the proper functioning of the system and compromising user funds and privacy.

Recommendation: To address this vulnerability and ensure proper access control, it is recommended to implement a suitable authorization mechanism for the rebaseOptInExternal function. The following steps are suggested:

Implement an access control mechanism such as EIP-712 or similar standards to verify the authenticity of the opt-in request.

Require a valid ECDSA signature from the address being opted in, demonstrating their explicit consent.

By introducing these measures, the contract can ensure that only authorized users can opt in an address on their behalf, maintaining user autonomy and protecting against unauthorized access and misuse.

[M-3] Both functions in MakerRouterFacet will revert due to insufficient balance

Context: MakerRouterFacet.sol

Impact

The MakerRouterFacet contract is currently susceptible to a vulnerability that causes both the swapUsdcForDai and swapDaiForUsdc functions to revert due to insufficient balance. This limitation prevents the functions from executing as intended, as there is no mechanism to ensure that the required funds are available in the contract before the swap operations.

Description

The MakerRouterFacet contract includes two functions, swapUsdcForDai and swapDaiForUsdc, which both contain comments indicating that the presence of USDC in the contract address needs to be ensured before execution. However, there is no actual functionality implemented to handle this verification step. As a result, the only way to execute these functions successfully is by manually sending the required funds to the contract beforehand, which deviates from the expected flow of the system.

Here are the relevant code snippets:

```
function swapUsdcForDai(uint256 amountUsdcIn) external onlyAdmin {
    require(daiPSM.tin() == 0, "MakerRouter: maker fee not 0");

    // First ensure USDC resides in this address.
    USDC.safeApprove(GEM_JOIN, amountUsdcIn); /// approve DAI PSM to spend USDC
    daiPSM.sellGem(address(this), amountUsdcIn); /// sell USDC for DAI
    // Deploy DAI to new Vault.
}

function swapDaiForUsdc(uint256 amountDaiIn) external onlyAdmin {
    require(daiPSM.tout() == 0, "MakerRouter: maker fee not 0");

    // First ensure USDC resides in this address.
    DAI.safeApprove(address(daiPSM), amountDaiIn); /// approve DAI PSM to spend DAI
    daiPSM.buyGem(address(this), amountDaiIn / USDC_SCALING_FACTOR); /// sell DAI for USDC
    // Deploy USDC to new Vault.
}
```

These code blocks demonstrate that the comments indicate the need to ensure the presence of USDC in the contract before executing the swap operations. However, there is no accompanying code that performs the necessary verification or balance checks, leading to reverts when attempting to execute these functions without pre-sending the required funds.

Recommendation

To address this vulnerability and restore the expected flow, it is recommended to modify the swapUsdcForDai and swapDaiForUsdc functions to include functionality that pulls the required funds from the sender's address. This can be achieved by using the `safeTransferFrom` function of the respective ERC20 token contracts.

The suggested modifications are as follows:

```
function swapUsdcForDai(uint256 amountUsdcIn) external onlyAdmin {
    require(daiPSM.tin() == 0, "MakerRouter: maker fee not 0");

    // Pull USDC from the sender's address.
    USDC.safeTransferFrom(msg.sender, address(this), amountUsdcIn);
    ...
}

function swapDaiForUsdc(uint256 amountDaiIn) external onlyAdmin {
    require(daiPSM.tout() == 0, "MakerRouter: maker fee not 0");

    // Pull DAI from the sender's address.
    DAI.safeTransferFrom(msg.sender, address(this), amountDaiIn);
    ...
```

[M-4] Incorrect deduction from allowance without considering infinite allowance in transferFrom function

Context: FiToken.sol

Impact: The impact of this vulnerability is that the transferFrom function incorrectly deducts from the allowance even when the allowance is set to an infinite value. This can result in unexpected behavior and may lead to unauthorized transfers of tokens due to incorrect deduction from the allowance.

Description: The transferFrom function in the provided code snippet is responsible for transferring tokens from one account (_from) to another (_to). In the process, it deducts the transferred amount (_value) from the allowance granted by _from to msg.sender (the caller of the function).

```
_allowances[_from][msg.sender] = _allowances[_from][msg.sender].sub(_value);
```

However, this code does not consider the case where the allowance is set to an infinite value. As a result, it mistakenly deducts the _value from the allowance, regardless of whether it is necessary or not. This can potentially allow unauthorized transfers of tokens even when the allowance is intended to be infinite.

Recommendation

To address this vulnerability, it is recommended to modify the code in the transferFrom function to check if the allowance is set to an infinite value before deducting from it. The code should be updated as follows:

```
if (_allowances[_from][msg.sender] != uint256(-1)) {
    _allowances[_from][msg.sender] = _allowances[_from][msg.sender].sub(_value);
}
```

By including this additional check, the deduction from the allowance will only occur if the allowance is not set to an infinite value (uint256(-1)). This ensures that the deduction is performed correctly.

5.4 Low Risk

[L-1] Hardcoded Addresses wont work for multi-chain deployments

Context SwapRouterFacet.sol, MakerRouterFacet.sol

Impact The impact of this vulnerability is that the smart contracts rely on hardcoded addresses for external contracts and routers, which may not work correctly for multi-chain deployments. If the addresses of the referenced contracts change or the deployment is on a different chain, the contracts may fail to interact with the intended contracts and routers, leading to incorrect behavior and potential malfunctions.

Description In the provided code snippets, there are multiple instances where addresses of external contracts and routers are hardcoded. These hardcoded addresses include references to Maker DAI-USDC PSM, DAI contract, USDC contract, and a contract used for selling USDC for DAI. Additionally, there is a constant `routerAddress` that contains a hardcoded address for a router.

```
IDSSPSM public constant daiPSM = IDSSPSM(0x89B78CfA322F6C5dE0aBcEecab66Aee45393cC5A); IERC20 public
↪ constant DAI = IERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F); IERC20 public constant USDC =
↪ IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48); address public constant GEM_JOIN =
↪ 0x0A59649758aa4d66E25f08Dd01271e891fe52199;
```

```
address public constant routerAddress = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
```

This approach of using hardcoded addresses restricts the smart contracts to specific networks and may cause compatibility issues when deploying the contracts on different chains or when the addresses of the referenced contracts change.

Recommendation To address this vulnerability, it is recommended to pass the required addresses as constructor parameters during contract deployment instead of using hardcoded addresses. By making the addresses configurable during deployment, the smart contracts can be deployed on different networks and adapt to changes in contract addresses.

[L-2] Potential self-removal of admin account in toggleAdmin function

Context: PointToken.sol, FiToken.sol

Impact The impact of this vulnerability is that the toggleAdmin function allows for the removal of the admin account itself. If the admin account address is mistakenly or maliciously passed to the toggleAdmin function, the ownership of the contract can be revoked without any possibility of recovery. This can lead to a loss of control over the contract and potential misuse or unauthorized modifications.

Description In the provided code snippets, there are two instances of the toggleAdmin function. This function is responsible for toggling the admin status of an account, allowing it to perform certain privileged actions. However, due to a naive mistake or malicious actions, if the admin account address is passed as an argument to the toggleAdmin function, the function will toggle the admin status of that account, including the admin account itself.

```
function toggleAdmin(address _account) external isAdmin {
    admin[_account] = !admin[_account];
}
```

```
function toggleAdmin(address _account) external onlyAdmin {
    admin[_account] = !admin[_account];
}
```

This vulnerability exposes the contract to potential loss of ownership and control, as the admin account can unintentionally or maliciously remove its own admin status, leaving no means for recovery.

Recommendation To address this vulnerability, it is recommended to use the Ownable contract provided by OpenZeppelin. The Ownable contract implements a more robust ownership mechanism and avoids the risk of accidentally or maliciously removing the admin account.

By utilizing Ownable, you can replace the vulnerable toggleAdmin function with the secure ownership management provided by OpenZeppelin's Ownable contract. This will ensure proper ownership control and prevent unauthorized removal of the admin account.

[L-3] GPv2SafeERC20 safeTransfer functions can return true for addresses that do not exist

Context: LibToken.sol

Impact

The GPv2SafeERC20 contract, utilized by LibToken, lacks a check to verify the existence of the token being transferred. This implementation oversight can lead to confusion within the system, as address.call() will return true for non-existent addresses, falsely indicating a successful transfer. While this vulnerability may not currently pose an issue, future changes in the system could potentially expose it to honeypot attacks.

Description

The GPv2SafeERC20 contract, used in LibToken, does not include a necessary check to ensure the existence of the token being transferred. When using the address.call() function, which returns true for non-existent addresses, the system may be misled into believing that the transfer has been successfully completed, despite the token not actually existing.

Here are the relevant code snippets:

```
function safeTransfer(IERC20 token, address to, uint256 value) internal {
    bytes4 selector_ = token.transfer.selector;

    // solhint-disable-next-line no-inline-assembly
    assembly {
        let freeMemoryPointer := mload(0x40)
        mstore(freeMemoryPointer, selector_)
        mstore(add(freeMemoryPointer, 4), and(to, 0xffffffffffffffffffffffffffffffffffffffff))
        mstore(add(freeMemoryPointer, 36), value)

        if iszero(call(gas(), token, 0, freeMemoryPointer, 68, 0, 0)) {
            returndatacopy(0, 0, returndatasize())
            revert(0, returndatasize())
        }
    }

    require(getLastTransferResult(token), 'GPv2: failed transfer');
}
```

```
function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
    bytes4 selector_ = token.transferFrom.selector;

    // solhint-disable-next-line no-inline-assembly
    assembly {
        let freeMemoryPointer := mload(0x40)
        mstore(freeMemoryPointer, selector_)
        mstore(add(freeMemoryPointer, 4), and(from, 0xffffffffffffffffffffffffffffffffffffffff))
        mstore(add(freeMemoryPointer, 36), and(to, 0xffffffffffffffffffffffffffffffffffffffff))
        mstore(add(freeMemoryPointer, 68), value)

        if iszero(call(gas(), token, 0, freeMemoryPointer, 100, 0, 0)) {
            returndatacopy(0, 0, returndatasize())
            revert(0, returndatasize())
        }
    }

    require(getLastTransferResult(token), 'GPv2: failed transferFrom');
}
```

These code blocks demonstrate that the GPv2SafeERC20 contract lacks the necessary checks to verify whether the token being transferred actually exists. The contract relies on address.call(), which returns true for non-existent addresses, leading to the false assumption of a successful transfer.

While this vulnerability may not be problematic at present, future changes to the system may expose it to potential honeypot attacks. Malicious actors could create a fake token contract at a non-existent address and manipulate the transfer functions to deceive users.

Recommendation

To mitigate this vulnerability, it is recommended to introduce checks in the LibToken contract to verify the existence of the token before utilizing the safeTransfer and safeTransferFrom functions. One approach is to check whether the specified token address contains any code. If the contract at the given address does not have any code, it indicates that the token does not exist, and the transfer should be rejected.

Alternatively, considering the usage of OpenZeppelin's safe transfer mechanism throughout other

[L-4] Missing rebase opt-out check in toggleFreeze function

Context FiToken.sol

Impact

Potential freezing of accounts without ensuring they are opted out of rebases.

Description The toggleFreeze function is designed to toggle the frozen state of an account. However, the comment within the function claims that if freezing, it should be ensured that the account is opted out of rebases. Unfortunately, the code does not include any check to verify if the account has been opted out of rebases. As a result, accounts can be frozen without ensuring they are protected from rebases, which can lead to unexpected behavior and potential vulnerabilities.

Recommendation

It is recommended to add a require statement in the toggleFreeze function to check whether the account has been opted out of rebases before toggling the frozen state. This can be done by introducing a modifier or a conditional statement that verifies the opt-out status of the account. By performing this check, the function will provide an additional layer of security and prevent freezing an account that has not been properly opted out of rebases. Here's an example of how the code could be modified:

```
function toggleFreeze(address _account) external onlyAdmin {
    require(isOptedOut[_account] || !frozen[_account], "Account must be opted out of rebases before freezing.");
    frozen[_account] = !frozen[_account];
}
```

By including the require statement and appropriate conditions, the function will now check whether the account has been opted out of rebases before allowing the frozen state to be toggled.

[L-5] Unsafe use of Address.isContract in isNonRebasingAccount function

Context: FiToken.sol

Impact

The current implementation of `_isNonRebasingAccount` function may lead to incorrect assumptions about whether an address is a contract or an externally-owned account (EOA). This can result in inaccurate account status determination and manipulation, potentially leading to unexpected behavior, security issues, and broken interactions with smart wallets like Gnosis Safe.

Description

The `_isNonRebasingAccount` function uses `Address.isContract` to determine if an address is a contract.

```
function _isNonRebasingAccount(address _account) internal returns (bool) {
    bool isContract = Address.isContract(_account);
    if (isContract && rebaseState[_account] == RebaseOptions.NotSet) {
        _ensureRebasingMigration(_account);
    }
    return nonRebasingCreditsPerToken[_account] > 0;
}
```

However, `Address.isContract` can return false for not only externally-owned accounts but also for contracts under certain circumstances:

- When the contract is in construction
- When the address is designated for a contract to be created
- When a contract existed at the address but was later destroyed

Furthermore, `Address.isContract` might return true for a contract scheduled for destruction by `SELFDESTRUCT` within the same transaction, even though this operation only takes effect at the end of a transaction.

This method of determining account type is therefore unreliable and can lead to incorrect decisions by the `_isNonRebasingAccount` function, especially if the function assumes an address is an EOA when it is actually a contract in one of the mentioned states.

Recommendation

It's recommended to revise the `_isNonRebasingAccount` function to treat both contracts and EOAs the same with regards to rebasing. This would reduce the risk of possible attack vectors and prevent potential inaccuracies when determining account status.

Preventing calls from contracts should be avoided as it breaks composability and support for smart wallets, which could be vital for the overall functionality and security of the system. If a distinction between EOAs and contracts is necessary, consider using a more reliable method to differentiate between the two.