Brahma Fi

Date	May 2022
Auditors	David Oz Kashi, Sergii Kravchenko, George Kobakhidze

1 Executive Summary

This report presents the results of our engagement with **Brahma Fi** to review **ETH Maxi**.

The review was conducted over two weeks, from **May 16, 2022** to **May 26, 2022** by **David Oz Kashi**, **Sergii Kravchenko**, and **George Kobakhidze**. A total of 30 person-days were spent.

2 Scope

Our review focused on the commit hash `3c42187d0564dea9db97c03835b6ab7993ab96a3` . The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Brahma Fi** team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Recommendations

3.1 Use the same Solidity version across contracts

Description

Most contracts use the same Solidity version with `pragma solidity ^0.8.0` . The only exception is the `Batcher` contract that has `pragma solidity ^0.8.4` .

Recommendation

It would be best to utilise and document the same version within all contract code to avoid any issues and inconsistencies that may arise across Solidity versions. From conversations with the Brahma-fi team, it appears that all contracts were compiled with and tested on Solidity version 0.8.4, so perhaps that one can be picked.

3.2 General Recommendations

Description

Among the many contracts, there are a few minor inaccuracies in the comments or the code and some parts could potentially be improved with industry standard practices.

Examples

- 2 step changes for privileged contract addresses.

Some contracts in the system have setters for privileged addresses that control the contract logic, such as the keeper. Especially for those that are intended to be controlled by a private key as opposed to a contract, it would be best to do a two step change for those addresses. First, nominate the address, and second accept the nomination from that address ensuring that the access is indeed secured. In fact, one contract already uses this pattern:

code/contracts/Vault.sol:L337-L359

```
/// @notice Nominates new governance address.
/// @dev  Governance will only be changed if the new governance accepts it. It will be pending till then.
/// @param _governance The address of new governance.
function setGovernance(address _governance) public {
    onlyGovernance();
    pendingGovernance = _governance;
}

/// @notice Emitted when governance is updated.
/// @param oldGovernance The address of the current governance.
/// @param newGovernance The address of new governance.
event UpdatedGovernance(
    address indexed oldGovernance,
    address indexed newGovernance
);

/// @notice The nomine of new governance address proposed by `setGovernance` function can accept the governance.
/// @dev  This can only be called by address of pendingGovernance.
function acceptGovernance() public {
    require(msg.sender == pendingGovernance, "INVALID_ADDRESS");
    emit UpdatedGovernance(governance, pendingGovernance);
    governance = pendingGovernance;
}
```

Consider adding the same pattern to these and others where you may see it as appropriate:

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L180-L184

```
/// @notice keeper setter
/// @param _keeper new keeper address
function setKeeper(address _keeper) public onlyAuthorized {
    keeper = _keeper;
}
```

code/contracts/Vault.sol:L365-L372

```
/// @notice Sets new keeper address.
/// @dev  This can only be called by governance.
/// @param _keeper The address of new keeper.
function setKeeper(address _keeper) public {
    onlyGovernance();
    keeper = _keeper;
    emit UpdatedKeeper(_keeper);
}
```

- Emit events for critical changes.

Some contracts emit events during critical changes, such as `Vault.setPerformanceFee()` with `emit UpdatePerformanceFee`, whereas other contracts, like trade executors, do not have events for likewise crucial changes. Consider creating and implementing events for functions like:

code/contracts/Batcher/Batcher.sol:L362-L372

```
/// @inheritdoc IBatcher
function setVaultLimit(uint256 maxAmount) external override {
    onlyGovernance();
    vaultInfo.maxAmount = maxAmount;
}

/// @notice Function to enable/disable deposit signature check
function setDepositSignatureCheck(bool enabled) public {
    onlyGovernance();
    checkValidDepositSignature = enabled;
}
```

code/contracts/ConvexTradeExecutor.sol:L30-L39

```
/// @param _harvester address of harvester
function setHandler(address _harvester) external onlyGovernance {
    ConvexPositionHandler._configHandler(_harvester, vaultWantToken());
}

/// @notice Governance function to set max accepted slippage of swaps
///@param _slippage Max accepted slippage during harvesting
function setSlippage(uint256 _slippage) external onlyGovernance {
    ConvexPositionHandler._setSlippage(_slippage);
}
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L449-L453

```
/// @notice Keeper function to set max accepted slippage of swaps
/// @param _slippage Max accepted slippage during harvesting
function _setSlippage(uint256 _slippage) internal {
    maxSlippage = _slippage;
}
```

code/contracts/ConvexExecutor/Harvester.sol:L87-L96

```
/// @notice Keeper function to set position handler to harvest for
/// @param _addr address of the position handler
function setPositionHandler(address _addr)
    external
    override
    validAddress(_addr)
    onlyKeeper
{
    positionHandler = _addr;
}
```

code/contracts/LyraTradeExecutor.sol:L65-L75

```
/// @notice Socket registry setter, called by keeper
/// @param _socketRegistry address of new socket registry
function setSocketRegistry(address _socketRegistry) public onlyKeeper {
    socketRegistry = _socketRegistry;
}

/// @notice L2 Position Handler setter, called by keeper
/// @param _l2HandlerAddress address of new position handler on L2
function setL2Handler(address _l2HandlerAddress) public onlyKeeper {
    positionHandlerL2Address = _l2HandlerAddress;
}
```

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L74-L76

```
function setSlippage(uint256 _slippage) public onlyAuthorized {
    slippage = _slippage;
}
```

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L174-L184

```
/// @notice socket registry setter
/// @param _socketRegistry new address of socket registry
function setSocketRegistry(address _socketRegistry) public onlyAuthorized {
    socketRegistry = _socketRegistry;
}

/// @notice keeper setter
/// @param _keeper new keeper address
function setKeeper(address _keeper) public onlyAuthorized {
    keeper = _keeper;
}
```

- Unnecessary variables with hardcoded values

The `ConvexPositionHandler` has constants that are used throughout the contract as well as variables that are initialised within the `_configHandler()` function with exactly the same hardcoded values. Consider keeping and using only one of the two sets

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L72-L85

```
/// @notice address of LP Token (ETH/stETH pool)
IERC20 public immutable lpToken =
    IERC20(0x06325440D014e39736583c165C2963BA99fAf14E);

/// @notice address of Convex reward pool
IConvexRewards public immutable baseRewardPool =
    IConvexRewards(0x0A760466E1B4621579a82a39CB56Dda2F4E70f03);
/// @notice address of Convex booster
IConvexBooster public immutable convexBooster =
    IConvexBooster(0xF403C135812408BFbE8713b5A23a04b3D48AAE31);

/// @notice address of ETH/stETH pool
ICurvePool public immutable ethStEthPool =
    ICurvePool(0xDC24316b9AE028F1497c275EB9192a3Ea0f67022);
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L92-L96

```
/// @param _wantToken address of want token
function _configHandler(address _harvester, address _wantToken) internal {
    address ETH_STETH_POOL = 0xDC24316b9AE028F1497c275EB9192a3Ea0f67022;
    address LP_TOKEN = 0x06325440D014e39736583c165C2963BA99fAf14E;
    address CONVEX_BOOSTER = 0xF403C135812408BFbE8713b5A23a04b3D48AAE31;
```

- 2 functions with almost the same name doing the same thing

`LyraPositionHandlerL2` and `UniswapV3Controller` (that `LyraPositionHandlerL2` inherits from) have two functions that do the same thing `setSlippage()` and `_setSlippage()` respectively. The former is public with a `onlyAuthorized` modifier whereas the latter is internal only. Consider implementing `LyraPositionHandlerL2.setSlippage()` such that it instead calls `_setSlippage()`

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L74-L76

```
function setSlippage(uint256 _slippage) public onlyAuthorized {
    slippage = _slippage;
}
```

code/contracts/LyraL2/UniswapV3Controller.sol:L43-L45

```
function _setSlippage(uint256 _slippage) internal {
    slippage = _slippage;
}
```

- Typos

There are minor typos in a few function names as well as some inconsistent comments. Consider fixing to make things like code search by function name / comment description easier.

- BaseTradeExecutor “initate” instead of “init**l**ate”

code/contracts/BaseTradeExecutor.sol:L63-L67

```
function initateWithdraw(bytes calldata _data) public override onlyKeeper {
    require(!withdrawalStatus.inProcess, "WITHDRW_IN_PROGRESS");
    withdrawalStatus.inProcess = true;
    _initiateWithdraw(_data);
}
```

- BaseTradeExecutor “WITHDRW_IN_PROGRESS” and “WIHDRW_COMPLETED” are inconsistently or incorrectly shortened as opposed to “WITHDRAW_IN_PROGRESS” and “WITHDRAW_COMPLETED”

code/contracts/BaseTradeExecutor.sol:L63-L73

```
function initateWithdraw(bytes calldata _data) public override onlyKeeper {
    require(!withdrawalStatus.inProcess, "WITHDRW_IN_PROGRESS");
    withdrawalStatus.inProcess = true;
    _initiateWithdraw(_data);
}

function confirmWithdraw() public override onlyKeeper {
    require(withdrawalStatus.inProcess, "WIHDRW_COMPLETED");
    _confirmWithdraw();
    withdrawalStatus.inProcess = false;
}
```

- `LyraTradeExecutor.confirmWithdraw()` incorrect comment. It is the same as the one for `_confirmDeposit()` and says it confirms transfer **to** the L2 contract as opposed to **from** the L2 contract.

code/contracts/LyraTradeExecutor.sol:L97-L102

```
/// @notice To confirm transfer of want tokens to L2
/// @dev Handle anything related to deposit confirmation
function _confirmWithdraw() internal override {
    if (address(this).balance > 0)
        IWETH9(wantTokenL1).deposit{value: address(this).balance}();
}
```

- `SocketV1Controller.decodeSocketRegistryCalldata` comment suggests that this is “not in use due to undertainity in bungee api response”, however this is called by `verifySocketCalldata()` in `sendTokens()` which in turn is called by `LyraPositionHandler._deposit()`

code/contracts/LyraExecutor/SocketV1Controller.sol:L37-L41

```
/// @notice Decode the socket request calldata
/// @dev Currently not in use due to undertainity in bungee api response
/// @param _data Bungee txn calldata
/// @return userRequest parsed calldata
function decodeSocketRegistryCalldata(bytes memory _data)
```

4 Security Specification

4.1 Privileged users trust assumptions

Keepers and governance are heavily trusted by the protocol users. Among other trust assumptions, they are trusted to:

- Keepers are trusted to move batched funds to the vault contract.
- Keepers act as a trusted intermediary to access L2 components.
- Batcher contract depositors trust the keeper to call `batchDeposit` at some point, otherwise funds are locked in the contract.
- Batcher contract depositors trust the keeper to process their withdraw requests.
- Governance can sweep out any token from basically any contract with no regard for accounting.

4.2 Observations

- `Vault.totalExecutorFunds` may run out of gas, which will require the governance to withdraw the funds held in some executor(s) and then call `removeExecutor`.
- `Batcher` is using digital signatures to allow only a specific list of depositors. While this approach saves gas, it is worth taking into account that this right currentl can not be revoked once given.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.

- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 The virtual price may not correspond to the actual price in the pool Major

Description

A Curve pool has a function that returns a "virtual price" of the LP token; this price is resistant to flash-loan attacks and any manipulations in the Curve pool. While this price formula works well in some cases, there may be a significant period when a trade cannot be executed with this price. So the deposit or withdrawal will also be done under another price and will have a different result than the one estimated under the "virtual price".

When depositing into Curve, Brahma is doing it in 2 steps. First, when depositing the user's ETH to the Vault, the user's share is calculated according to the "virtual price". And then, in a different transaction, the funds are deposited into the Curve pool. These funds only consist of ETH, and if the deposit price does not correspond (with 0.3% slippage) to the virtual price, it will revert.

So we have multiple problems here:

1. If the chosen slippage parameter is very low, the funds will not be deposited/withdrawn for a long time due to reverts.
2. If the slippage is large enough, the attacker can manipulate the price to steal the slippage. Additionally, because of the 2-steps deposit, the amount of Vault's share minted to the users may not correspond to the LP tokens minted during the second step.

5.2 ConvexPositionHandler._claimRewards incorrectly calculates amount of LP tokens to unstake

Major

Description

`ConvexPositionHandler._claimRewards` is an internal function that harvests Convex reward tokens and takes the generated yield in ETH out of the Curve pool by calculating the difference in LP token price. To do so, it receives the current share price of the curve LP tokens and compares it to the last one stored in the contract during the last rewards claim. The difference in share price is then multiplied by the LP token balance to get the ETH yield via the `yieldEarned` variable:

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L293-L300

```
uint256 currentSharePrice = ethStEthPool.get_virtual_price();
if (currentSharePrice > prevSharePrice) {
    // claim any gain on lp token yields
    uint256 contractLpTokenBalance = lpToken.balanceOf(address(this));
    uint256 totalLpBalance = contractLpTokenBalance +
        baseRewardPool.balanceOf(address(this));
    uint256 yieldEarned = (currentSharePrice - prevSharePrice) *
        totalLpBalance;
```

However, to receive this ETH yield, LP tokens need to be unstaked from the Convex pool and then converted via the Curve pool. To do this, the contract introduces `lpTokenEarned`:

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L302

```
uint256 lpTokenEarned = yieldEarned / NORMALIZATION_FACTOR; // 18 decimal from virtual price
```

This calculation is incorrect. It uses yieldEarned which is denominated in ETH and simply divides it by the normalization factor to get the correct number of decimals, which still returns back an amount denominated in ETH, whereas an amount denominated in LP tokens should be returned instead.

This could lead to significant accounting issues including losses in the "no-loss" parts of the vault's strategy as 1 LP token is almost always guaranteed to be worth more than 1 ETH. So, when the intention is to withdraw `X` ETH worth of an LP token, withdrawing `X` LP tokens will actually withdraw `Y` ETH worth of an LP token, where `Y>X`. As a result, less than expected ETH will remain in the Convex handler part of the vault, and the ETH yield will go to the Lyra options, which are much riskier. In the event Lyra options don't work out and there is more ETH withdrawn than expected, there is a possibility that this would result in a loss for the vault.

Recommendation

The fix is straightforward and that is to calculate `lpTokenEarned` using the `currentSharePrice` already received from the Curve pool. That way, it is the amount of LP tokens that will be sent to be unwrapped and unstaked from the Convex and Curve pools. This will also take care of the normalization factor. `uint256 lpTokenEarned = yieldEarned / currentSharePrice;`

5.3 The `WETH` tokens are not taken into account in the `ConvexTradeExecutor.totalFunds` function Major

Description

The `totalFunds` function of every executor should include all the funds that belong to the contract:

code/contracts/ConvexTradeExecutor.sol:L21-L23

```
function totalFunds() public view override returns (uint256, uint256) {
    return ConvexPositionHandler.positionInWantToken();
}
```

The `ConvexTradeExecutor` uses this function for calculations:

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L121-L137

```
function positionInWantToken()
    public
    view
    override
    returns (uint256, uint256)
{
    (
        uint256 stakedLpBalanceInETH,
        uint256 lpBalanceInETH,
        uint256 ethBalance
    ) = _getTotalBalancesInETH(true);

    return (
        stakedLpBalanceInETH + lpBalanceInETH + ethBalance,
        block.number
    );
}
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L337-L365

```
function _getTotalBalancesInETH(bool useVirtualPrice)
    internal
    view
    returns (
        uint256 stakedLpBalance,
        uint256 lpTokenBalance,
        uint256 ethBalance
    )
{
    uint256 stakedLpBalanceRaw = baseRewardPool.balanceOf(address(this));
    uint256 lpTokenBalanceRaw = lpToken.balanceOf(address(this));

    uint256 totalLpBalance = stakedLpBalanceRaw + lpTokenBalanceRaw;

    // Here, in order to prevent price manipulation attacks via curve pools,
    // When getting total position value -> its calculated based on virtual price
    // During withdrawal -> calc_withdraw_one_coin() is used to get an actual estimate of ETH received if we were to remove liquidi
    // The following checks account for this
    uint256 totalLpBalanceInETH = useVirtualPrice
        ? _lpTokenValueInETHFromVirtualPrice(totalLpBalance)
        : _lpTokenValueInETH(totalLpBalance);

    lpTokenBalance = useVirtualPrice
        ? _lpTokenValueInETHFromVirtualPrice(lpTokenBalanceRaw)
        : _lpTokenValueInETH(lpTokenBalanceRaw);

    stakedLpBalance = totalLpBalanceInETH - lpTokenBalance;
    ethBalance = address(this).balance;
}
```

This function includes ETH balance, LP balance, and staked balance. But WETH balance is not included here. WETH tokens are initially transferred to the contract, and before the withdrawal, the contract also stores WETH.

Recommendation

Include WETH balance into the `totalFunds`.

5.4 LyraPositionHandlerL2 inaccurate modifier onlyAuthorized may lead to funds loss if keeper is compromised Medium

Description

The `LyraPositionHandlerL2` contract is operated either by the L2 keeper or by the L1 `LyraPositionHandler` via the `L2CrossDomainMessenger`. This is implemented through the `onlyAuthorized` modifier:

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L187-L195

```
modifier onlyAuthorized() {
    require(
        ((msg.sender == L2CrossDomainMessenger &&
            OptimismL2Wrapper.messageSender() == positionHandlerL1) ||
            msg.sender == keeper),
        "ONLY_AUTHORIZED"
    );
    _;
}
```

This is set on:

1. `withdraw()`
2. `openPosition()`
3. `closePosition()`
4. `setSlippage()`
5. `deposit()`
6. `sweep()`
7. `setSocketRegistry()`
8. `setKeeper()`

Functions 1-3 have a corresponding implementation on the L1 `LyraPositionHandler`, so they could indeed be called by it with the right parameters. However, 4-8 do not have an implemented way to call them from L1, and this modifier creates an unnecessarily expanded list of authorised entities that can call them.

Additionally, even if their implementation is provided, it needs to be done carefully because `msg.sender` in their case is going to end up being the `L2CrossDomainMessenger`. For example, the `sweep()` function sends any specified token to `msg.sender`, with the intention likely being that the recipient is under the team's or the governance's control – yet, it will be `L2CrossDomainMessenger` and the tokens will likely be lost forever instead.

On the other hand, the `setKeeper()` function would need a way to be called by something other than the keeper because it is intended to change the keeper itself. In the event that the access to the L2 keeper is compromised, and the L1 `LyraPositionHandler` has no way to call `setKeeper()` on the `LyraPositionHandlerL2`, the whole contract and its funds will be compromised as well. So, there needs to be some way to at least call the `setKeeper()` by something other than the keeper to ensure security of the funds on L2.

Examples

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L153-L184

```
function closePosition(bool toSettle) public override onlyAuthorized {
    LyraController._closePosition(toSettle);
    UniswapV3Controller._estimateAndSwap(
        false,
        LyraController.sUSD.balanceOf(address(this))
    );
}

/*////////////////////////////////////////////////////////////
                      MAINTAINANCE FUNCTIONS
////////////////////////////////////////////////////////////*/

/// @notice Sweep tokens
/// @param _token Address of the token to sweepr
function sweep(address _token) public override onlyAuthorized {
    IERC20(_token).transfer(
        msg.sender,
        IERC20(_token).balanceOf(address(this))
    );
}

/// @notice socket registry setter
/// @param _socketRegistry new address of socket registry
function setSocketRegistry(address _socketRegistry) public onlyAuthorized {
    socketRegistry = _socketRegistry;
}

/// @notice keeper setter
/// @param _keeper new keeper address
function setKeeper(address _keeper) public onlyAuthorized {
    keeper = _keeper;
}
```

Recommendation

Create an additional modifier for functions intended to be called just by the keeper (`onlyKeeper`) such as functions 4-7, and create an additional modifier `onlyGovernance` for the `setKeeper()` function. As an example, the L1 `Vault` contract also has a `setKeeper()` function that has a `onlyGovernance()` modifier. Please note that this will likely require implementing a function for the system's governance that can call `LyraPositionHandlerL2.setKeeper()` via the `L2CrossDomainMessenger`.

5.5 Harvester.harvest swaps have no slippage parameters Medium

Description

As part of the vault strategy, all reward tokens for staking in the Convex ETH-stETH pool are claimed and swapped into ETH. The swaps for these tokens are done with no slippage at the moment, i.e. the expected output amount for all of them is given as 0.

In particular, one reward token that is most susceptible to slippage is LDO, and its swap is implemented through the Uniswap router:

code/contracts/ConvexExecutor/Harvester.sol:L142-L155

```
function _swapLidoForWETH(uint256 amountToSwap) internal {
    IUniswapSwapRouter.ExactInputSingleParams
        memory params = IUniswapSwapRouter.ExactInputSingleParams({
            tokenIn: address(ldo),
            tokenOut: address(weth),
            fee: UNISWAP_FEE,
            recipient: address(this),
            deadline: block.timestamp,
            amountIn: amountToSwap,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });
    uniswapRouter.exactInputSingle(params);
}
```

The swap is called with `amountOutMinimum: 0`, meaning that there is no slippage protection in this swap. This could result in a significant loss of yield from this reward as MEV bots could "sandwich" this swap by manipulating the price before this transaction and immediately reversing their action after the transaction, profiting at the expense of our swap. Moreover, the Uniswap pools seem to have low liquidity for the LDO token as opposed to Balancer or Sushiswap, further magnifying slippage issues and susceptibility to frontrunning.

The other two tokens - CVX and CRV - are being swapped through their Curve pools, which have higher liquidity and are less susceptible to slippage. Nonetheless, MEV strategies have been getting more advanced and calling these swaps with 0 as expected output may place these transactions in danger of being frontrun and “sandwiched” as well.

code/contracts/ConvexExecutor/Harvester.sol:L120-L126

```
if (cvxBalance > 0) {
    cvxeth.exchange(1, 0, cvxBalance, 0, false);
}
// swap CRV to WETH
if (crvBalance > 0) {
    crveth.exchange(1, 0, crvBalance, 0, false);
}
```

In these calls `.exchange` , the last `0` is the `min_dy` argument in the Curve pools swap functions that represents the minimum expected amount of tokens received after the swap, which is 0 in our case.

Recommendation

Introduce some slippage parameters into the swaps.

5.6 Harvester.rewardTokens doesn’t account for LDO tokens Medium

Description

As part of the vault’s strategy, the reward tokens for participating in Curve’s ETH-stETH pool and Convex staking are claimed and swapped for ETH. This is done by having the `ConvexPositionHandler` contract call the reward claims API from Convex via `baseRewardPool.getReward()`, which transfers the reward tokens to the handler’s address. Then, the tokens are iterated through and sent to the harvester to be swapped from `ConvexPositionHandler` by getting their list from `harvester.rewardTokens()` and calling `harvester.harvest()`

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L274-L290

```
// get list of tokens to transfer to harvester
address[] memory rewardTokens = harvester.rewardTokens();
//transfer them
uint256 balance;
for (uint256 i = 0; i < rewardTokens.length; i++) {
    balance = IERC20(rewardTokens[i]).balanceOf(address(this));

    if (balance > 0) {
        IERC20(rewardTokens[i]).safeTransfer(
            address(harvester),
            balance
        );
    }
}

// convert all rewards to WETH
harvester.harvest();
```

However, `harvester.rewardTokens()` doesn’t have the LDO token’s address in its list, so they will not be transferred to the harvester to be swapped.

code/contracts/ConvexExecutor/Harvester.sol:L77-L82

```
function rewardTokens() external pure override returns (address[] memory) {
    address[] memory rewards = new address[](2);
    rewards[0] = address(crv);
    rewards[1] = address(cvx);
    return rewards;
}
```

As a result, `harvester.harvest()` will not be able to execute its `_swapLidoForWETH()` function since its `ldoBalance` will be 0. This results in missed rewards and therefore yield for the vault as part of its normal flow.

There is a possible mitigation in the current state of the contract that would require governance to call `sweep()` on the LDO balance from the `BaseTradeExecutor` contract (that `ConvexPositionHandler` inherits) and then transferring those LDO tokens to the harvester contract to perform the swap at a later rewards claim. This, however, requires transactions separate from the intended flow of the system as well as governance intervention.

Recommendation

Add the LDO token address to the `rewardTokens()` function by adding the following line `rewards[2] = address(ldo);`

5.7 Keeper design complexity Medium

Description

The current design of the protocol relies on the keeper being operated correctly in a complex manner. Since the offchain code for the keeper wasn’t in scope of this audit, the following is a commentary on the complexity of the keeper operations in the context of the contracts. Keeper logic such as the order of operations and function argument parameters with log querying are some examples where if the keeper doesn’t execute them correctly, there may be inconsistencies and issues with accounting of vault shares and vault funds resulting in unexpected behaviour. While it may represent little risk or issues to the current Brahma-fi team as the vault is recently live, the keeper logic and exact steps should be well documented so that public keepers (if and when they are enabled) can execute the logic securely and future iterations of the vault code can account for any intricacies of the keeper logic.

Examples

1. Order of operations: Convex rewards & new depositors profiting at the expense of old depositors' yielded reward tokens. As part of the vault's strategy, the depositors' ETH is provided to Curve and the LP tokens are staked in Convex, which yield rewards such as CRV, CVX, and LDO tokens. As new depositors provide their ETH, the vault shares minted for their deposits will be less compared to old deposits as they account for the increasing value of LP tokens staked in these pools. In other words, if the first depositor provides 1 ETH, then when a new depositor provides 1 ETH much later, the new depositor will get less shares back as the `totalVaultFunds()` will increase:

code/contracts/Vault.sol:L97-L99

```
shares = totalSupply() > 0
    ? (totalSupply() * amountIn) / totalVaultFunds()
    : amountIn;
```

code/contracts/Vault.sol:L127-L130

```
function totalVaultFunds() public view returns (uint256) {
    return
        IERC20(wantToken).balanceOf(address(this)) + totalExecutorFunds();
}
```

code/contracts/ConvexTradeExecutor.sol:L21-L23

```
function totalFunds() public view override returns (uint256, uint256) {
    return ConvexPositionHandler.positionInWantToken();
}
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L121-L137

```
function positionInWantToken()
    public
    view
    override
    returns (uint256, uint256)
{
    (
        uint256 stakedLpBalanceInETH,
        uint256 lpBalanceInETH,
        uint256 ethBalance
    ) = _getTotalBalancesInETH(true);

    return (
        stakedLpBalanceInETH + lpBalanceInETH + ethBalance,
        block.number
    );
}
```

However, this does not account for the reward tokens yielded throughout that time. From the smart contract logic alone, there is no requirement to first execute the reward token harvest. It is up to the keeper to execute `ConvexTradeExecutor.claimRewards` in order to claim and swap their rewards into ETH, which only then will be included into the yield in the above `ConvexPositionHandler.positionInWantToken` function. If this is not done prior to processing new deposits and minting new shares, new depositors would unfairly benefit from the reward tokens' yield that was generated before they deposited but accounted for in the vault funds only after they deposited.

2. Order of operations: closing Lyra options before processing new deposits.

The other part of the vault's strategy is utilising the yield from Convex to purchase options from Lyra on Optimism. While Lyra options are risky and can become worthless in the event of bad trades, only yield is used for them, therefore keeping user deposits' initial value safe. However, their value could also yield significant returns, increasing the overall funds of the vault. Just as with `ConvexTradeExecutor`, `LyraTradeExecutor` also has a `totalFunds()` function that feeds into the vault's `totalVaultFunds()` function. In Lyra's case, however, it is a manually set value by the keeper that is supposed to represent the value of Lyra L2 options:

code/contracts/LyraTradeExecutor.sol:L42-L53

```
function totalFunds()
    public
    view
    override
    returns (uint256 posValue, uint256 lastUpdatedBlock)
{
    return (
        positionInWantToken.posValue +
            IERC20(vaultWantToken()).balanceOf(address(this)),
        positionInWantToken.lastUpdatedBlock
    );
}
```

code/contracts/LyraTradeExecutor.sol:L61-L63

```
function setPosValue(uint256 _posValue) public onlyKeeper {
    LyraPositionHandler._setPosValue(_posValue);
}
```

code/contracts/LyraExecutor/LyraPositionHandler.sol:L218-L221

```
function _setPosValue(uint256 _posValue) internal {
    positionInWantToken.posValue = _posValue;
    positionInWantToken.lastUpdatedBlock = block.number;
}
```

Solely from the smart contract logic, there is a possibility that a user deposits when Lyra options are valued high, meaning the total vault funds are high as well, thus decreasing the amount of shares the user would have received if it weren't for the Lyra options' value. Consequently, if after the deposit the Lyra options become worthless, decreasing the total vault funds, the user's newly minted shares will now represent less than what they have deposited.

While this is not currently mitigated by smart contract logic, it may be worked around by the keeper first settling and closing all Lyra options and transferring all their yielded value in ETH, if any, to the Convex trade executor. Only then the keeper would process new deposits and mint new shares. This order of operations is critical to maintain the vault's intended safe strategy of maintaining the user's deposited value, and is dependent entirely on the keeper offchain logic.

3. Order of operations: additional trade executors and their specific management Similarly to the above examples, as more trade executors and position handlers are added to the vault, the complexity for the keeper will go up significantly, requiring it to maintain all correct orders of operations **not just** to keep the shares and funds accounting intact, but simply for the trade executors to function normally. For example, in the case of Lyra, the keepers need to manually call `confirmDeposit` and `confirmWithdraw` to update their `depositStatus` and `withdrawalStatus` respectively to continue normal operations or otherwise new deposits and withdrawals wouldn't be processed. On the other hand, the Convex executor does it automatically. Due to the system design, there may be no single standard way to handle a trade executor. New executors may also require specific calls to be done manually, increasing overall complexity keeper logic to support the system.

4. Keeper calls & arguments: `depositFunds` / `batchDeposit` and `initiateWithdrawal` / `batchWithdraw` `userAddresses[]` array + gas overhead With the current gated approach and batching for deposits and withdrawals to and from the vault, users aren't able to directly mint and redeem their vault shares. Instead, they interact with the `Batcher` contract that then communicates with the `Vault` contract with the help of the keeper. However, while each user's deposit and withdrawal amounts are registered in the contract state variables such as `depositLedger[user]` and `withdrawLedger[user]`, and there is an event emitted with the user address and their action, to process them the keeper is required to keep track of all the user addresses in the batch they need to process. In particular, the keeper needs to provide `address[] memory users` for both `batchDeposit()` and `batchWithdraw()` functions that communicate with the vault. There is no stored list of users within the contract that could provide or verify the right users, so it is entirely up to the keeper's offchain logic to query the logs and retrieve the addresses required. Therefore, depending on the size of the `address[] memory users` array, the keepers may need to consider the transaction gas limit, possibly requiring splitting the array up and doing several transactions to process all of them. In addition, in the event of withdrawals, the keepers need to calculate how much of the `wantToken` (WETH in our case) will be required to process the withdrawals, and call `withdrawFromExecutor()` with that amount to provide enough assets to cover withdrawals from the vault.

5. Timing: 50 block radius for updates on trade executors that need to have their values updated via a call Some trade executors, like the Convex one, can retrieve their funds value at any time from Layer 1, thereby always being up to date with the current block. Others, like the Lyra trade executor, require the keeper to update their position value by initiating a call, which also updates their `positionInWantToken.lastUpdatedBlock` state variable. However, this variable is also called during during the `vault.totalVaultFunds()` call during deposits and withdrawals via `totalExecutorFunds()`, which eventually calls `areFundsUpdated(blockUpdated)`. This is a check to ensure that the current transaction's `block.number <= _blockUpdated + BLOCK_LIMIT`, where `BLOCK_LIMIT` =50 blocks, i.e. roughly 12-15 min. As a result, keepers need to make sure that all executors that require a call for this have their position values updated before and rather close to processing and deposits or withdrawals, or `areFundsUpdated()` will revert those calls.

Recommendation

Document the exact order of operations, steps, necessary logs and parameters that keepers need to keep track of in order for the vault strategy to succeed.

5.8 `Vault.deposit` - Possible front running attack Medium

Description

To determine the number of shares to mint to a depositor, `(totalSupply() * amountIn) / totalVaultFunds()` is used. Potential attackers can spot a call to `Vault.deposit` and front-run it with a transaction that sends tokens to the contract, causing the victim to receive fewer shares than what he expected.

In case `totalVaultFunds()` is greater than `totalSupply() * amountIn`, then the number of shares the depositor receives will be 0, although `amountIn` of tokens will be still pulled from the depositor's balance.

An attacker with access to enough liquidity and to the mem-pool data can spot a call to `Vault.deposit(amountIn, receiver)` and front-run it by sending at least `totalSupplyBefore * (amountIn - 1) + 1` tokens to the contract . This way, the victim will get 0 shares, but `amountIn` will still be pulled from its account balance. Now the price for a share is inflated, and all shareholders can redeem this profit using `Vault.withdraw`.

The attack vector mentioned above is the general front runner case, the most profitable attack vector will be the case when the attacker is able to determine the share price (for instance if the attacker mints the first share). In this scenario, the attacker will need to send at least `attackerShares * (amountIn -1) + 1` to the contract,(`attackerShares` is completely controlled by the attacker), and this amount can be then entirely redeemed by the attacker himself (alongside the victim's deposit) by calling `Vault.withdraw`. The attacker can lower the risk of losing the funds he sent to the contract to some other front-runner by using the flashbots api. Although both `Vault.deposit` and `Vault.withdraw` are callable only by the `Batcher` contract, the keeper bot can still be tricked to process user deposits in a way that allows this attack to happen.

Recommendation

The specific case that's mentioned in the last paragraph can be mitigated by adding a validation check to `Vault.Deposit` enforcing that `shares > 0`. However, it will not solve the general case since the victim can still lose value due to rounding errors. In order to fix that, `Vault.Deposit` should validate that `shares >= amountMin` where `amountMin` is an argument that should be determined by the depositor off-chain.

5.9 Approving MAX_UINT amount of ERC20 tokens Minor

Description

Approving the maximum value of uint256 is a known practice to save gas. However, this pattern was proven to increase the impact of an attack many times in the past, in case the approved contract gets hacked.

Examples

code/contracts/BaseTradeExecutor.sol:L19

```
IERC20(vaultWantToken()).approve(vault, MAX_INT);
```

code/contracts/Batcher/Batcher.sol:L48

```
IERC20(vaultInfo.tokenAddress).approve(vaultAddress, type(uint256).max);
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L106-L112

```
IERC20(LP_TOKEN).safeApprove(ETH_STETH_POOL, type(uint256).max);

// Approve max LP tokens to convex booster
IERC20(LP_TOKEN).safeApprove(
    address(CONVEX_BOOSTER),
    type(uint256).max
);
```

code/contracts/ConvexExecutor/Harvester.sol:L65-L69

```
crv.safeApprove(address(crveth), type(uint256).max);
// max approve CVX to CVX/ETH pool on curve
cvx.safeApprove(address(cvxeth), type(uint256).max);
// max approve LDO to uniswap swap router
ldo.safeApprove(address(uniswapRouter), type(uint256).max);
```

code/contracts/LyraL2/LyraPositionHandlerL2.sol:L63-L71

```
IERC20(wantTokenL2).safeApprove(
    address(UniswapV3Controller.uniswapRouter),
    type(uint256).max
);
// approve max susd balance to uniV3 router
LyraController.sUSD.safeApprove(
    address(UniswapV3Controller.uniswapRouter),
    type(uint256).max
);
```

Recommendation

Consider approving the exact amount that's needed to be transferred, or alternatively, add an external function that allows the revocation of approvals.

5.10 Batcher.depositFunds may allow for more deposits than vaultInfo.maxAmount Minor

Description

As part of a gradual rollout strategy, the Brahma-fi system of contracts has a limit of how much can be deposited into the protocol. This is implemented through the `Batcher` contract that allows users to deposit into it and keep the amount they have deposited in the `depositLedger[recipient]` state variable. In order to cap how much is deposited, the user's input `amountIn` is evaluated within the following statement:

code/contracts/Batcher/Batcher.sol:L109-L116

```
require(
    IERC20(vaultInfo.vaultAddress).totalSupply() +
        pendingDeposit -
        pendingWithdrawal +
        amountIn <=
        vaultInfo.maxAmount,
    "MAX_LIMIT_EXCEEDED"
);
```

However, while `pendingDeposit`, `amountIn`, and `vaultInfo.maxAmount` are denominated in the vault asset token (WETH in our case), `IERC20(vaultInfo.vaultAddress).totalSupply()` and `pendingWithdrawal` represent vault shares tokens, creating potential mismatches in this evaluation.

As the yield brings in more and more funds to the vault, the amount of share minted for each token deposited in decreases, so `totalSupply()` becomes less than the total deposited amount (not just vault funds) as the strategy succeeds over time. For example, at first `X` deposited tokens would mint `X` shares. After some time, this would create additional funds in the vault through yield, and another `X` deposit of tokens would mint **less** than `X` shares, say `X-Y`, where `Y` is some number greater than 0 representing the difference in the number of shares minted. So, while there were `2*X` deposited tokens, `totalSupply()=(2*X-Y)` shares would have been minted in total. However, at the time of the next deposit, a user's `amountIn` will be added with `totalSupply()=(2*X-Y)` number of shares instead of a greater `2*X` number of deposited tokens. So, this will undershoot the actual amount of tokens deposited after this user's deposit, thus potentially evaluating it less than `maxAmount`, and letting more user deposits get inside the vault than what was intended.

Recommendation

Consider either documenting this potential discrepancy or keeping track of all deposits in a state variable and using that inside the `require` statement..

5.11 The `Deposit` and `Withdraw` event are always emitted with zero amount Minor

Description

The events emitted during the deposit or withdraw are supposed to contain the relevant amounts of tokens involved in these actions. But in fact the current balance of the address is used in both cases. These balances will be equal to zero by that time:

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L151-L155

```
IWETH9(address(wantToken)).withdraw(depositParams._amount);

_convertEthIntoLpToken(address(this).balance);

emit Deposit(address(this).balance);
```

code/contracts/ConvexExecutor/ConvexPositionHandler.sol:L207-L209

```
IWETH9(address(wantToken)).deposit{value: address(this).balance}();

emit Withdraw(address(this).balance);
```

5.12 `BaseTradeExecutor.confirmDeposit | confirmWithdraw` - Violation of the “checks-effects-interactions” pattern Minor

Description

Both `confirmDeposit, confirmWithdraw` might be re-entered by the keeper (in case it is a contract), in case the derived contract allows the execution of untrusted code.

Examples

code/contracts/BaseTradeExecutor.sol:L57-L61

```
function confirmDeposit() public override onlyKeeper {
    require(depositStatus.inProcess, "DEPOSIT_COMPLETED");
    _confirmDeposit();
    depositStatus.inProcess = false;
}
```

code/contracts/BaseTradeExecutor.sol:L69-L73

```
function confirmWithdraw() public override onlyKeeper {
    require(withdrawalStatus.inProcess, "WIHDRW_COMPLETED");
    _confirmWithdraw();
    withdrawalStatus.inProcess = false;
}
```

Recommendation

Although the impact is very limited, it is recommended to implement the “checks-effects-interactions” in both functions.

5.13 Batcher doesn’t work properly with arbitrary tokens Minor

Description

The `Batcher` and the `Vault` contracts initially operate with ETH and WETH. But the contracts are supposed to be compatible with any other ERC-20 tokens.

For example, in the `Batcher.deposit` function, there is an option to transfer ETH instead of the token, which should only be happening if the token is WETH. Also, the token is named `WETH`, but if the intention is to use the Batcher contract with arbitrary tokens token, it should be named differently.

code/contracts/Batcher/Batcher.sol:L89-L100

```
if (ethSent > 0) {
    amountIn = ethSent;
    WETH.deposit{value: ethSent}();
}
/// If no wei sent, use amountIn and transfer WETH from txn sender
else {
    IERC20(vaultInfo.tokenAddress).safeTransferFrom(
        msg.sender,
        address(this),
        amountIn
    );
}
```

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
interfaces/BasePositionHandler.sol	cb7eb1ed869d31b2f97cc36c025b1a0aa630fd8e
library/AddArrayLib.sol	393868cb0414892e35ce3ceadd6e2457e2f526f6
library/Math.sol	7401e9ae2b668aa9428627a83b0a53f58e11b591
contracts/BaseTradeExecutor.sol	9c08d481463debccbcda7af365799fd99ead6aaf
contracts/Batcher/EIP712.sol	385a2972608861d04b9306bd477b5552ed92c388
contracts/Batcher/Batcher.sol	66c88aeb6806033aa79c151b33d4eea22169cdef
contracts/LyraL2/OptimismL2Wrapper.sol	3f9161d6e7270630468ba8aac78cdbd0c0b4216b
contracts/LyraL2/LyraPositionHandlerL2.sol	700b5da119914b9018dc716da2faf8dc95f1c5dc
contracts/LyraL2/UniswapV3Controller.sol	573ceb693240249449a7b8ec9c2848bda81f9a0f
contracts/LyraL2/LyraController.sol	93265667bde12d6c52170de9533a025a05945916
contracts/LyraL2/SocketV1Controller.sol	bd56fb5edb69d5a5ec706d26ab5396291c3dcb57
contracts/Vault.sol	825a4ee02a1223e7196c6f2dfabb24a19e9d8fb0
contracts/LyraTradeExecutor.sol	b0f270388b7f5a6eb2e6cc95258b1d307cc0f20a
contracts/LyraExecutor/OptimismWrapper.sol	24234fc661797384a6737b2fe14c1d300047cf3b
contracts/LyraExecutor/SocketV1Controller.sol	7f33bca869994e24ce042a85b60a63969a76bed6
contracts/LyraExecutor/LyraPositionHandler.sol	41380f763675d6c03b3a271f673035c990b1f390
contracts/ConvexTradeExecutor.sol	6dec70809ff3ffb715eae217732a9d7084895f16
contracts/ConvexExecutor/Harvester.sol	34d6ec3a29fb729cde47bfb5e2c7de959400a6bd
contracts/ConvexExecutor/ConvexPositionHandler.sol	1e1d650b165c710429eff5599bce592d33e46857

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.