Mimo DeFi contest
Findings & Analysis Report

2022-07-18

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Mimo DeFi smart contract system written in Solidity. The audit contest took place between April 28 - May 2 2022.

Wardens

46 Wardens contributed reports to the Mimo DeFi contest:

1. 0xDjango
2. unforgiven
3. Picodes
4. AlleyCat
5. smiling_heretic
6. robee

7. defsec
8. pauliax
9. Dravee
10. 0x1f8b
11. broccolirob
12. MaratCerby
13. ych18
14. joestakey
15. berndartmueller
16. hyh
17. delfin454000
18. cccz
19. sorrynotsorry
20. kebabsec (okkothejawa and FlameHorizon)
21. z3s
22. 0xNazgul
23. 0x4non
24. rotcivegaf
25. Funen
26. GimelSec (rayn and sces60107)
27. samruna
28. 0x52
29. dipp
30. peritoflores
31. sikorico
32. GalloDaSballo
33. Hawkeye (0xwags and 0xmint)
34. ilan
35. luduvigo

36. shenwilly
37. simon135
38. 0xkatana
39. slywaters
40. Tomio
41. 0v3rf10w
42. oyc_109
43. Tadashi

This contest was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 33 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 26 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Mimo DeFi contest repository, and is composed of 26 smart contracts written in the Solidity programming language and includes 2,432 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] User can call liquidate() and steal all collateral due to arbitrary router call

Submitted by 0xDjango

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/core/contracts/liquidityMining/v2/PARMinerV2.sol#L126

https://github.com/Uniswap/v2-periphery/blob/2efa12e0f2d808d9b49737927f0e416fafa5af68/contracts/UniswapV2Router02.sol#L299

https://github.com/Uniswap/solidity-lib/blob/c01640b0f0f1d8a85cba8de378cc48469fcfd9a6/contracts/libraries/TransferHelper.sol#L47-L50

A malicious user is able to steal all collateral of an unhealthy position in `PARMinerV2.sol`. The code for the `liquidate()` function is written so that the following steps are followed:

- User calls `PARMinerV2.liquidate()`
- PARMinerV2 performs the liquidation with `_a.parallel().core().liquidatePartial()`
- PARMinerV2 receives the liquidated collateral
- An arbitrary router function is called to swap the collateral to PAR
- Finally, `PARMinerV2.liquidate()` checks that PARMinerV2's PAR balance is higher than the balance at the beginning of the function call.

The exploit occurs with the arbitrary router call. The malicious user is able to supply the `dexTxnData` parameter which dictates the function call to the router. If the user supplied a function such as UniswapV2Router's `swapExactTokenForETH()`, then control flow will be given to the user, allowing them to perform the exploit.

Note: The Mimo developers have stated that the routers used by the protocol will be DEX Aggregators such as 1inch and Paraswap, but this submission will be referring to UniswapV2Router for simplicity. It can be assumed that the dex aggregators currently allow swapping tokens for ETH.

Continuing the exploit, once the attacker has gained control due to the ETH transfer, they are able to swap the ETH for PAR. Finally, they deposit the PAR with `PARMinerV2.deposit()`. This will cause the final check of `liquidate()` to pass because PARMinerV2's PAR balance will be larger than the start of the liquidation call.

The attacker is able to steal all collateral from every unhealthy position that they liquidate. In the most extreme case, the attacker is able to open their own risky positions with the hope that the position becomes unhealthy. They will borrow the PAR and then liquidate themselves to take back the collateral. Thus effectively stealing PAR.

Proof of Concept

Steps for exploit:

- Attacker monitors unhealthy positions. Finds a position to liquidate.
- Attacker calls `PARMinerV2.liquidate()`

- Position liquidated. Collateral transferred back to `PARMinerV2`
- In the `liquidate()` function, attacker supplies bytes for `UniswapV2Router.swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline)`. For `to`, they supply the attacker contract.
- `swapExactTokensForETH()` firstly swaps the collateral for ETH and then transfers the ETH to the user with `TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);`
- `TransferHelper.safeTransferETH()` contains a call to the receiver via `(bool success, ) = to.call{value: value}(new bytes(0));`
- Therefore, the attacker contract will indeed gain control of execution.

The attacker contract will then perform the following steps:

- Swap the received ETH to PAR.
- Deposit the PAR in `PARMinerV2`
- Withdraw the deposited PAR.

Recommended Mitigation Steps

The arbitrary call to the router contracts is risky because of the various functions that they can contain. Perhaps a solution is to only allow certain calls such as swapping tokens to tokens, not ETH. This would require frequently updated knowledge of the router's functions, though would be beneficial for security.

Also, adding a check that the `_totalStake` variable has not increased during the liquidation call will mitigate the risk of the attacker depositing the PAR to increase the contract's balance. The attacker would have no option but to transfer the PAR to PARMinerV2 as is intended.

m19 (Mimo Defi) disagreed with severity and commented:

> We believe in theory this attack is actually possible, but highly unlikely to happen. It also begs the question of whether it's really worth it for an attacker to do this because they could just call VaultsCore.liquidate() themselves (for example with a flashloan) and stake all the PAR they profit that way directly.

m19 (Mimo DeFi) confirmed and commented:

> We misunderstood this exploit wrong and we confirm it. Basically, if the attacker was liquidating a 10,000 PAR position, he could potentially end up with a 10,000 PAR stake + liquidation profits. Our previous understanding was that he could only end up with the profits.

> At the very least we'll implement a check that `totalStake` hasn't changed, we will carefully consider if more changes are needed.

[H-02] Fund loss or theft by attacker with creating a flash loan and setting SuperVault as receiver so executeOperation() will be get called by lendingPool but with attackers specified params

Submitted by unforgiven, also found by Picodes

According to Aave documentation, when requesting flash-loan, it's possible to specify a `receiver`, so function `executeOperation()` of that `receiver` will be called by `lendingPool`. https://docs.aave.com/developers/v/2.0/guides/flash-loans In the `SuperVault` there is no check to prevent this attack so attacker can use this and perform `griefing attack` and make miner contract lose all its funds. or he can create specifically crafted `params` so when `executeOperation()` is called by `lendingPool`, attacker could steal vault's user funds.

Proof of Concept

To exploit this attacker will do this steps:

1. will call `Aave lendingPool` to get a flash-loan and specify `SuperVault` as `receiver` of flash-loan. and also create a specific `params` that invoke `Operation.REBALANCE` action to change user vault's collateral.
2. `lendingPool` will call `executeOperation()` of `SuperVault` with attacker specified data.
3. `executeOperation()` will check `msg.sender` and will process the function call which will cause some dummy exchanges that will cost user exchange fee

and flash-loan fee.

4. attacker will repeat this attack until user losses all his funds.

```
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address,
  bytes calldata params
) external returns (bool) {
  require(msg.sender == address(lendingPool), "SV002");
  (Operation operation, bytes memory operationParams) = abi.de
  IERC20 asset = IERC20(assets[0]);
  uint256 flashloanRepayAmount = amounts[0] + premiums[0];
  if (operation == Operation.LEVERAGE) {
    leverageOperation(asset, flashloanRepayAmount, operationPa
  }
  if (operation == Operation.REBALANCE) {
    rebalanceOperation(asset, amounts[0], flashloanRepayAmount
  }
  if (operation == Operation.EMPTY) {
    emptyVaultOperation(asset, amounts[0], flashloanRepayAmour
  }

  asset.approve(address(lendingPool), flashloanRepayAmount);
  return true;
}
```

To steal user fund in `SupperVault` attacker needs more steps. in all these actions (`Operation.REBALANCE` , `Operation.LEVERAGE` , `Operation.EMPTY`) contract will call `aggregatorSwap()` with data that are controlled by attacker.

```
function aggregatorSwap(
  uint256 dexIndex,
  IERC20 token,
  uint256 amount,
  bytes memory dexTxData
) internal {
  (address proxy, address router) = _dexAP.dexMapping(dexIndex
  require(proxy != address(0) && router != address(0), "SV201"
  token.approve(proxy, amount);
```

```
        router.call(dexTxData);
    }
```

Attacker can put special data in `dexTxData` that make contract to do an exchange with bad price. To do this, attacker will create a smart contract that will do this steps:

1. manipulate price in exchange with flash loan.
2. make a call to `executeOperation()` by `Aave flash-loan` with `receiver` and specific `params` so that `SuperVault` will make calls to manipulated exchange for exchanging.
3. do the reverse of #1 and pay the flash-loan and steal the user fund.

The details are: Attacker can manipulate swapping pool price with flash-loan, then Attacker will create specific `params` and perform steps 1 to 4. so contract will try to exchange tokens and because of attacker price manipulation and specific `dexTxData`, contract will have bad deals. After that, attacker can reverse the process of swap manipulation and get his flash-loan tokens and some of `SuperVault` funds and. then pay the flash-loan.

Tools Used

VIM

Recommended Mitigation Steps

There should be some state variable which stores the fact that `SuperVault` imitated flash-loan. When contract tries to start flash-loan, it sets the `isFlash` to `True` and `executeOperation()` only accepts calls if `isFlash` is `True`. and after the flash loan code will set `isFlash` to `False`.

m19 (Mimo DeFi) confirmed and commented:

> We definitely confirm this issue and intend to fix it.

Medium Risk Findings (5)

[M-01] Decimal token underflow could produce loss of funds

Submitted by 0x1f8b, also found by broccolirob and pauliax

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/core/contracts/oracles/GUniLPOracle.sol#L47

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/core/contracts/oracles/GUniLPOracle.sol#L51

It is possible to produce underflows with specific tokens which can cause errors when calculating prices.

Proof of Concept

The pragma is `pragma solidity 0.6.12;` therefore, integer overflows must be protected with safe math. But in the case of GUniLPOracle, there is a decimal subtraction that could underflow if any token in the pool has more than 18 decimals. This could cause an error when calculating price values.

Recommended Mitigation Steps

Ensure that tokens have less than 18 decimals.

m19 (Mimo DeFi) confirmed and commented:

> We confirm this issue.

[M-02] Users can use updateBoost function to claim unfairly large rewards from liquidity mining contracts for themselves at cost of other users.

*Submitted by smiling*heretic, also found by unforgiven_

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/core/contracts/liquid

ityMining/v2/PARMinerV2.sol#L159-L165

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/core/contracts/liquidityMining/v2/GenericMinerV2.sol#L88-L94

Users aware of this vulnerability could effectively steal a portion of liquidity mining rewards from honest users.

Affected contracts are: `SupplyMinerV2`, `DemandMinerV2`, `PARMinerV2`

`VotingMinerV2` is less affected because locking veMIMO in `votingEscrow` triggers a call to `releaseMIMO` of this miner contract (which in turn updates user's boost multiplier).

Proof of Concept

Let's focus here on `SupplyMinerV2`. The exploits for other liquidity mining contracts are analogous.

Scenario 1:

Both Alice and Bob deposit 1 WETH to `coreVaults` and borrow 100 PAR from `coreVaults`. They both have no locked veMIMO.

Now they wait for a month without interacting with the protocol. In the meantime, `SupplyMinerV2` accumulated 100 MIMO for rewards.

Alice locks huge amount of veMIMO in `votingEscrow`, so now her `boostMultiplier` is 4.

Let's assume that Alice and Bob are the only users of the protocol. Because they borrowed the same amounts of PAR, they should have the same stakes for past month, so a fair reward for each of them (for this past month) should be 50 MIMO. If they simply repay their debts now, 50 MIMO is indeed what they get.

However if Alice calls `supplyMiner.updateBoost(alice)` before repaying her debt, she can claim 80 MIMO and leave only 20 MIMO for Bob. She can basically

apply the multiplier 4 to her past stake.

Scenario 2:

Both Alice and Bob deposit 1 WETH to `coreVaults` and borrow 100 PAR from `coreVaults`. Bob locks huge amount of veMIMO in `votingEscrow` for 4 years, so now his `boostMultiplier` is 4.

Alice and Bob wait for 4 years without interacting with the protocol.

`SupplyMinerV2` accumulated 1000 MIMO rewards.

Because of his locked veMIMO, Bob should be able to claim larger reward than Alice. Maybe not 4 times larger but definitely larger.

However, if Alice includes a transaction with call `supplyMiner.updateBoost(bob)` before Bob's `vaultsCore.repay()`, then she can claim 500 MIMO. She can effectively set Bob's `boostMultiplier` for past 4 years to 1.

Tools Used

Tested in Foundry

Recommended Mitigation Steps

I have 2 ideas:

1. Remove `updateBoost` function. There shouldn't be a way to update boost multiplier without claiming rewards and updating `_userInfo.accAmountPerShare`. So `releaseRewards` should be sufficient.
2. A better, but also much more difficult solution, would be to redesign boost updates in such a way that distribution of rewards no longer depends on when and how often boost multiplier is updated. If the formula for boost multiplier stays the same, this approach might require calculating integrals of the multiplier as a function of time.

m19 (Mimo DeFi) confirmed and commented:

> We agree this is an issue and intend to fix it.

[M-03] SuperVault's leverageSwap and emptyVaultOperation can become stuck

Submitted by hyh, also found by cccz, berndartmueller, delfin454000, joestakey, robee, defsec, and 0xDjango

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/supervaults/contracts/SuperVault.sol#L320-L326

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/supervaults/contracts/SuperVault.sol#L198-L199

leverageSwap and emptyVaultOperation can be run repeatedly for the same tokens. If these tokens happen to be an ERC20 that do not allow for approval of positive amount when allowance already positive, both functions can become stuck.

https://github.com/d-xo/weird-erc20#approval-race-protections

In both cases, logic doesn't seem to guarantee full usage of the allowance given. If it's not used fully, the token will revert each next approve attempt, which will render the functions unavailable for the token.

While emptyVaultOperation can be cured by emptying the balance and rerun, in the leverageSwap case there is no such fix possible.

Setting severity to medium as this clearly impacts leverageSwap and emptyVaultOperation availability to the users.

Proof of Concept

leverageSwap calls target token for maximum approval of core each time:

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/supervaults/contracts/SuperVault.sol#L320-L326

```
///@param token The leveraged asset to swap PAR for
function leverageSwap(bytes memory params, IERC20 token) inter
  (uint256 parToSell, bytes memory dexTxData, uint dexIndex) =
    params,
    (uint256, bytes, uint )
  );
  token.approve(address(a.core()), 2**256 - 1);
```

Some tokens do not have maximum amount as an exception, simply reverting any attempt to approve positive from positive, for example current USDT contract, L205:

https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code

I.e. if leverageSwap be run again with USDT it will revert all the times after the first.

emptyVaultOperation approves core for the whole balance of stablex:

https://github.com/code-423n4/2022-04-mimo/blob/b18670f44d595483df2c0f76d1c57a7bfbfbc083/supervaults/contracts/SuperVault.sol#L198-L199

```
IERC20 par = IERC20(a.stablex());
par.approve(address(a.core()), par.balanceOf(address(this)))
```

Recommended Mitigation Steps

Consider adding zero amount approval before actual amount approval, i.e. force zero allowance before current approval.

m19 (Mimo DeFi) acknowledged

gzeoneth (judge) commented:

> Having `approve(0)` first will still revert with USDT because the interface expect it to return a bool but USDT return void. Fund also won't be stuck because it will revert. Judging as Med Risk as function availability could be impacted. Unlike the

> core protocol, `SuperVault` can take any token as input and USDT is listed on various lending protocol like AAVE.

[M-04] Non-standard ERC20 Tokens are Not Supported

Submitted by ych18, also found by MaratCerby, robee, and defsec

When trying to call `SuperVault.executeOperation` the transaction reverts. This is because the call to `asset.approve()` in line{97} doesn't match the expected function signature of `approve()` on the target contract like in the case of USDT.

This issue exists in any call to approve function when the asset could be any ERC20.

Recommendation : consider using safeApprove of OZ

m19 (Mimo DeFi) acknowledged

gzeoneth (judge) decreased severity to Medium and commented:

> Judging as Med Risk as function availability could be impacted. Unlike the core protocol, `SuperVault` can take any token as input and USDT is listed on various lending protocol like AAVE.

[M-05] ABDKMath64 performs multiplication on results of division

Submitted by AlleyCat

https://github.com/code-423n4/2022-04-mimo/blob/main/core/contracts/libraries/ABDKMath64x64.sol#L626

https://github.com/code-423n4/2022-04-mimo/blob/main/core/contracts/libraries/ABDKMath64x64.sol#L629

https://github.com/code-423n4/2022-04-mimo/blob/main/core/contracts/libraries/ABDKMath64x64.sol#L630

Solidity could truncate the results, performing multiplication before division will prevent rounding/truncation in solidity math.

Recommended Mitigation Steps

Consider ordering multiplication first.

m19 (Mimo DeFi) acknowledged

Low Risk and Non-Critical Issues

For this contest, 33 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **Dravee** received the top score from the judge.

The following wardens also submitted reports: robee, defsec, pauliax, 0xDjango, berndartmueller, hyh, joestakey, 0x52, dipp, kebabsec, peritoflores, Picodes, sikorico, sorrynotsorry, z3s, 0x1f8b, 0x4non, AlleyCat, cccz, delfin454000, Funen, GalloDaSballo, GimelSec, Hawkeye, ilan, luduvigo, MaratCerby, rotcivegaf, samruna, shenwilly, simon135, *and* unforgiven.

Table of Contents

See original submission for details.

[L-01] `approve` should be replaced with `safeApprove` or `safeIncreaseAllowance()` / `safeDecreaseAllowance()`

`approve` is subject to a known front-running attack. Consider using `safeApprove` instead:

```
core/contracts/liquidityMining/v2/PARMinerV2.sol:
   58:     _par.approve(address(_a.parallel().core()), uint256(-
  125:     collateralToken.approve(proxy, collateralToken.balanc

core/echidna/TInceptionVaultHealthy.sol:
   33:     _weth.approve(address(a), _adminDepositAmount);
   39:     _link.approve(address(v), _userDepositAmount);
```

```
  47:     _par.approve(address(_inceptionVaultsCore), _MAX_INT);

core/echidna/TInceptionVaultUnhealthy.sol:
  37:     _weth.approve(address(a), _adminDepositAmount);
  43:     _link.approve(address(v), _userDepositAmount);
  52:     _par.approve(address(_inceptionVaultsCore), _MAX_INT);

core/echidna/TInceptionVaultUnhealthyAssertion.sol:
  36:     _weth.approve(address(a), _adminDepositAmount);
  42:     _link.approve(address(v), _userDepositAmount);
  51:     _par.approve(address(_inceptionVaultsCore), _MAX_INT);

core/echidna/TInceptionVaultUnhealthyProperty.sol:
  35:     _weth.approve(address(a), _adminDepositAmount);
  41:     _link.approve(address(v), _userDepositAmount);
  50:     _par.approve(address(_inceptionVaultsCore), _MAX_INT);

supervaults/contracts/SuperVault.sol:
   97:     asset.approve(address(lendingPool), flashloanRepayAmo
  149:     IERC20(toCollateral).approve(address(a.core()), depos
  199:     par.approve(address(a.core()), par.balanceOf(address
  273:     token.approve(address(a.core()), amount);
  289:     token.approve(address(a.core()), depositAmount);
  326:     token.approve(address(a.core()), 2**256 - 1);
  345:     token.approve(proxy, amount);
```

Keep in mind though that it would be actually better to replace safeApprove() with safeIncreaseAllowance() or safeDecreaseAllowance().

See this discussion: SafeERC20.safeApprove() Has unnecessary and unsecure added behavior

[L-02] Add constructor initializers

As per OpenZeppelinâ€™s (OZ) recommendation, â€œThe guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment.â€

Note that this behaviour is also incorporated the OZ Wizard since the UUPS vulnerability discovery: â€œAdditionally, we modified the code generated by the

Wizard 19 to include a constructor that automatically initializes the implementation when deployed.â€

Furthermore, this thwarts any attempts to frontrun the initialization tx of these contracts:

```
core/contracts/inception/AdminInceptionVault.sol:
  35:   function initialize(

core/contracts/inception/InceptionVaultsCore.sol:
  40:   function initialize(

core/contracts/inception/InceptionVaultsDataProvider.sol:
  30:   function initialize(IInceptionVaultsCore inceptionVaults

core/contracts/inception/priceFeed/ChainlinkInceptionPriceFeed.s
  29:   function initialize(

supervaults/contracts/SuperVault.sol:
  49:   function initialize(
```

[L-03] Missing address(0) checks

According to Slither:

```
AdminInceptionVault.initialize(address,IAddressProvider,IDebtNot
  - owner = _owner (contracts/inception/AdminInceptionVault.sol#
InceptionVaultsCore.initialize(address,IInceptionVaultsCore.Vaul
  - owner = _owner (contracts/inception/InceptionVaultsCore.sol#
DemandMinerV2.setFeeCollector(address).feeCollector (contracts/l
  - _feeCollector = feeCollector (contracts/liquidityMining/v2/I
PARMinerV2.liquidate(uint256,uint256,uint256,bytes).router (cont
  - router.call(dexTxData) (contracts/liquidityMining/v2/PARMine
```

[L-04] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus

decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious Manager making a frontrunning/sandwich attack on the fees).

Consider adding a timelock to:

```
File: DemandMinerV2.sol
56:   function setFeeConfig(FeeConfig memory newFeeConfig) exter
57:     _feeConfig = newFeeConfig;
58:     emit FeeConfigSet(newFeeConfig);
59:   }
```

[L-05] Fee in `DemandMinerV2.setFeeConfig()` should be upper-bounded

```
File: DemandMinerV2.sol
56:   function setFeeConfig(FeeConfig memory newFeeConfig) exter
57:     _feeConfig = newFeeConfig;
58:     emit FeeConfigSet(newFeeConfig);
59:   }
```

[N-01] Unused named returns

Using both named returns and a return statement isn't necessary. Removing one of those can improve code clarity:

```
core/contracts/inception/priceFeed/ChainlinkInceptionPriceFeed.s
  73:   function getAssetPrice() public view override returns (u
```

[N-02] Useless import: SafeMath

```
File: SuperVault.sol
6: import "@openzeppelin/contracts/utils/math/SafeMath.sol"; //@
```

[N-03] The visibility for constructor is ignored

```
File: SuperVaultFactory.sol
17:   constructor(address _base) public {
```

m19 (Mimo DeFi) commented:

> Very clear and well structured QA report

gzeoneth (judge) commented:

> For L-01 I don't think there are front-running risk but the suggestion to use `safeIncreaseAllowance` is fine Otherwise looks good and I think the severity of each issue is well labeled.

Gas Optimizations

For this contest, 26 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **Dravee** received the top score from the judge.

The following wardens also submitted reports: joestakey, robee, 0xNazgul, defsec, 0xkatana, slywaters, Tomio, 0v3rf10w, 0x4non, 0xDjango, delfin454000, MaratCerby, oyc_109, pauliax, rotcivegaf, sorrynotsorry, 0x1f8b, Funen, GimelSec, kebabsec, Picodes, samruna, Tadashi, ych18, *and* z3s.

Table of Contents

[G-01] Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it

To help the optimizer, declare a `storage` type variable and use it instead of repeatedly fetching the reference in a map or an array.

The effect can be quite significant.

As an example, instead of repeatedly calling `someMap[someIndex]`, save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

Instances include (check the `@audit` tags):

```
core/contracts/dex/DexAddressProvider.sol:
  53:     return (_dexMapping[index].proxy, _dexMapping[index].r
```

[G-02] Caching external values in memory

See the `@audit` tags for further details:

```
core/contracts/dex/DexAddressProvider.sol:
  22:     require(_a.controller().hasRole(_a.controller().MANAGE

supervaults/contracts/SuperVault.sol:
  369:     if (ga.mimo().balanceOf(address(this)) > 0) {  //@auc
```

[G-03] Using an existing memory variable instead of reading storage

See the `@audit` tags for further details:

```
core/contracts/oracles/BalancerV2LPOracle.sol:
  41:     (address _pool, IBalancerVault.PoolSpecialization toke
```

[G-04] `BalancerV2LPOracle.sol` : Tighly pack storage variables

I suggest going from (see `@audit` tags):

```
File: BalancerV2LPOracle.sol
14: contract BalancerV2LPOracle is AggregatorV3Interface, BNum {
15:   using SafeMath for uint256;
```

```
16:
17:   string public override description; //@audit gas: 32 bytes
18:   uint256 public override version = 3; //@audit gas: 32 byte
19:   uint8 public override decimals; //@audit gas: 1 byte, can
20:
21:   bytes32 public poolId; //@audit gas: 32 bytes
22:   IBalancerVault public vault; //@audit gas: 20 bytes
23:   IBalancerPool public pool; //@audit gas: 20 bytes
24:   AggregatorV3Interface public oracleA; //@audit gas: 20 byt
25:   AggregatorV3Interface public oracleB; //@audit gas: 20 byt
```

to

```
contract BalancerV2LPOracle is AggregatorV3Interface, BNum {
  using SafeMath for uint256;

  string public override description; //@audit gas: 32 bytes (sl
  uint256 public override version = 3; //@audit gas: 32 bytes (s
  uint8 public override decimals; //@audit gas: 1 byte (slot 3)

  IBalancerVault public vault; //@audit gas: 20 bytes (slot 3)
  IBalancerPool public pool; //@audit gas: 20 bytes  (slot 4)
  bytes32 public poolId; //@audit gas: 32 bytes <= this is the c
  AggregatorV3Interface public oracleA; //@audit gas: 20 bytes
  AggregatorV3Interface public oracleB; //@audit gas: 20 bytes
```

Which would save 1 storage slot.

[G-05] Variables that should be constant

According to slither:

```
BalancerV2LPOracle.version (contracts/oracles/BalancerV2LPOracle
GUniLPOracle.version (contracts/oracles/GUniLPOracle.sol#16) sho
```

[G-06] `> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
core/contracts/inception/InceptionVaultsCore.sol:122:    require
core/contracts/liquidityMining/v2/GenericMinerV2.sol:58:    requ
core/contracts/liquidityMining/v2/GenericMinerV2.sol:70:    requ
core/contracts/liquidityMining/v2/GenericMinerV2.sol:175:    rec
core/contracts/liquidityMining/v2/GenericMinerV2.sol:195:    rec
core/contracts/liquidityMining/v2/PARMinerV2.sol:52:    require(
core/contracts/liquidityMining/v2/PARMinerV2.sol:71:    require(
core/contracts/liquidityMining/v2/PARMinerV2.sol:254:    require
core/contracts/liquidityMining/v2/PARMinerV2.sol:284:    require
core/contracts/oracles/GUniLPOracle.sol:112:    require(rA > 0 |
```

Also, please enable the Optimizer.

[G-07] `<=` is cheaper than `<`

Strict inequalities (`<`) are more expensive than non-strict ones (`<=`). This is due to some supplementary checks (ISZERO, 3 gas)

I suggest using `<=` instead of `<` here:

```
core/contracts/libraries/ABDKMath64x64.sol:697:      return uint
```

[G-08] Splitting `require()` statements that use `&&` saves gas

Instead of using the `&&` operator in a single require statement to check multiple conditions, I suggest using multiple require statements with 1 condition per require

statement (saving 3 gas per `&`):

```
core/contracts/libraries/ABDKMath64x64.sol:
   35:     require(x >= -0x8000000000000000 && x <= 0x7FFFFFFFF
   83:     require(result >= MIN_64x64 && result <= MAX_64x64);
  107:     require(result >= MIN_64x64 && result <= MAX_64x64);
  120:     require(result >= MIN_64x64 && result <= MAX_64x64);
  133:     require(result >= MIN_64x64 && result <= MAX_64x64);
  207:     require(result >= MIN_64x64 && result <= MAX_64x64);
  288:     require(result >= MIN_64x64 && result <= MAX_64x64);
  413:     require(result >= MIN_64x64 && result <= MAX_64x64);

core/contracts/liquidityMining/v2/GenericMinerV2.sol:
   58:     require(boostConfig.a >= 1 && boostConfig.d > 0 && bc
   70:     require(newBoostConfig.a >= 1 && newBoostConfig.d > 0
  331:     require(multiplier >= 1e18 && multiplier <= _boostCor

core/contracts/liquidityMining/v2/PARMinerV2.sol:
   52:     require(boostConfig.a >= 1 && boostConfig.d > 0 && bc
   71:     require(newBoostConfig.a >= 1 && newBoostConfig.d > 0
  426:     require(multiplier >= 1e18 && multiplier <= _boostCor

supervaults/contracts/SuperVault.sol:
  344:     require(proxy != address(0) && router != address(0),
```

[G-09] `require()` should be used for checking error conditions on inputs and return values while `assert()` should be used for invariant checking

Properly functioning code should **never** reach a failing assert statement, unless there is a bug in your contract you should fix. Here, I believe the assert should be a require or a revert:

```
core/contracts/libraries/ABDKMath64x64.sol:641:      assert(xh =
```

As the Solidity version is `0.6.12 < 0.8.0`, the remaining gas would not be refunded in case of failure.

[G-10] Amounts should be checked for 0 before calling a transfer

Checking non-zero transfer values can avoid an expensive external call and save gas.

While this is done at some places, it's not consistently done in the solution.

I suggest adding a non-zero-value check here:

```
core/contracts/inception/AdminInceptionVault.sol:81:    asset.sa
core/contracts/inception/AdminInceptionVault.sol:101:    asset.s
core/contracts/inception/AdminInceptionVault.sol:124:    stablex
core/contracts/inception/AdminInceptionVault.sol:131:    _mimo.s
core/contracts/inception/AdminInceptionVault.sol:139:    par.saf
core/contracts/inception/AdminInceptionVault.sol:145:    this fu
core/contracts/inception/AdminInceptionVault.sol:151:    asset.s
core/contracts/inception/InceptionVaultsCore.sol:67:    _incepti
core/contracts/inception/InceptionVaultsCore.sol:93:    _incepti
core/contracts/inception/InceptionVaultsCore.sol:186:    stablex
core/contracts/inception/InceptionVaultsCore.sol:234:    stablex
core/contracts/inception/InceptionVaultsCore.sol:235:    stablex
core/contracts/inception/InceptionVaultsCore.sol:239:    _incept
core/contracts/liquidityMining/v2/DemandMinerV2.sol:67:    _toke
core/contracts/liquidityMining/v2/DemandMinerV2.sol:72:      _to
core/contracts/liquidityMining/v2/DemandMinerV2.sol:87:      _to
core/contracts/liquidityMining/v2/DemandMinerV2.sol:90:    _toke
core/contracts/liquidityMining/v2/PARMinerV2.sol:92:    _par.saf
core/contracts/liquidityMining/v2/PARMinerV2.sol:101:    _par.sa
core/contracts/liquidityMining/v2/PARMinerV2.sol:127:    _par.sa
supervaults/contracts/SuperVault.sol:129:    IERC20(asset).trans
supervaults/contracts/SuperVault.sol:247:    require(asset.trans
supervaults/contracts/SuperVault.sol:274:    token.transferFrom(
supervaults/contracts/SuperVault.sol:290:    token.transferFrom(
```

[G-11] An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
core/contracts/dex/DexAddressProvider.sol:16:    for (uint256 i;
```

[G-12] `++i` costs less gas compared to `i++` or `i += 1`

`++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`

Instances include:

```
core/contracts/dex/DexAddressProvider.sol:16:    for (uint256 i;
core/contracts/inception/AdminInceptionVault.sol:108:    for (ui
core/contracts/libraries/ABDKMath64x64.sol:396:            resul
core/contracts/libraries/ABDKMath64x64.sol:403:          absXShi
core/contracts/libraries/ABDKMath64x64.sol:463:    if (xc >= 0x2
core/contracts/libraries/ABDKMath64x64.sol:624:      if (xc >= (
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

[G-13] Usage of a non-native 256 bits uint as a counter in for-loops increases gas cost

Due to how the EVM natively works on 256 bit numbers, using a 8 bit number in for-loops introduces additional costs as the EVM has to properly enforce the limits of this smaller type.

See the warning at this link: https://docs.soliditylang.org/en/v0.8.0/internals/layout_in_storage.html#layout-of-state-variables-in-storage :

> When using elements that are smaller than 32 bytes, your contractâ€™s gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size. It is only beneficial to use reduced-size arguments if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Affected code:

```
core/contracts/inception/AdminInceptionVault.sol:108:    for (ui
```

Consider manually checking for the upper bound before the for-loop and using the `uint256` type as a counter in the mentioned for-loops.

[G-14] Public functions to external

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

```
clone(bytes) should be declared external:
 - SuperVaultFactory.clone(bytes) (contracts/SuperVaultFactory.s
```

parallel() should be declared external:
 - DexAddressProvider.parallel() (contracts/dex/DexAddressProvic
dexMapping(uint256) should be declared external:
 - DexAddressProvider.dexMapping(uint256) (contracts/dex/DexAddr
deposit(address,uint256) should be declared external:
 - AdminInceptionVault.deposit(address,uint256) (contracts/incep
borrow(uint256,uint256) should be declared external:
 - AdminInceptionVault.borrow(uint256,uint256) (contracts/incept
a() should be declared external:
 - AdminInceptionVault.a() (contracts/inception/AdminInceptionVa
debtNotifier() should be declared external:
 - AdminInceptionVault.debtNotifier() (contracts/inception/Admir
weth() should be declared external:
 - AdminInceptionVault.weth() (contracts/inception/AdminInceptic
mimo() should be declared external:
 - AdminInceptionVault.mimo() (contracts/inception/AdminInceptic
inceptionCore() should be declared external:
 - AdminInceptionVault.inceptionCore() (contracts/inception/Admi
collateralCount() should be declared external:
 - AdminInceptionVault.collateralCount() (contracts/inception/Ac
collaterals(uint8) should be declared external:
 - AdminInceptionVault.collaterals(uint8) (contracts/inception/A
collateralId(address) should be declared external:
 - AdminInceptionVault.collateralId(address) (contracts/inceptic
a() should be declared external:
 - InceptionVaultFactory.a() (contracts/inception/InceptionVault
debtNotifier() should be declared external:
 - InceptionVaultFactory.debtNotifier() (contracts/inception/Inc
weth() should be declared external:
 - InceptionVaultFactory.weth() (contracts/inception/InceptionVa
mimo() should be declared external:
 - InceptionVaultFactory.mimo() (contracts/inception/InceptionVa
adminInceptionVaultBase() should be declared external:
 - InceptionVaultFactory.adminInceptionVaultBase() (contracts/ir
inceptionVaultsCoreBase() should be declared external:
 - InceptionVaultFactory.inceptionVaultsCoreBase() (contracts/ir
inceptionVaultsDataProviderBase() should be declared external:
 - InceptionVaultFactory.inceptionVaultsDataProviderBase() (cont
inceptionVaultCount() should be declared external:
 - InceptionVaultFactory.inceptionVaultCount() (contracts/incept
priceFeedCount() should be declared external:
 - InceptionVaultFactory.priceFeedCount() (contracts/inception/I
inceptionVaults(uint256) should be declared external:
 - InceptionVaultFactory.inceptionVaults(uint256) (contracts/inc
priceFeeds(uint8) should be declared external:
 - InceptionVaultFactory.priceFeeds(uint8) (contracts/inception/

priceFeedIds(address) should be declared external:
 - InceptionVaultFactory.priceFeedIds(address) (contracts/incept
cumulativeRate() should be declared external:
 - InceptionVaultsCore.cumulativeRate() (contracts/inception/Inc
lastRefresh() should be declared external:
 - InceptionVaultsCore.lastRefresh() (contracts/inception/Incept
vaultConfig() should be declared external:
 - InceptionVaultsCore.vaultConfig() (contracts/inception/Incept
a() should be declared external:
 - InceptionVaultsCore.a() (contracts/inception/InceptionVaultsC
inceptionCollateral() should be declared external:
 - InceptionVaultsCore.inceptionCollateral() (contracts/inceptic
adminInceptionVault() should be declared external:
 - InceptionVaultsCore.adminInceptionVault() (contracts/inceptic
inceptionVaultsData() should be declared external:
 - InceptionVaultsCore.inceptionVaultsData() (contracts/inceptic
inceptionPriceFeed() should be declared external:
 - InceptionVaultsCore.inceptionPriceFeed() (contracts/inceptior
a() should be declared external:
 - InceptionVaultsDataProvider.a() (contracts/inception/Inceptic
inceptionVaultsCore() should be declared external:
 - InceptionVaultsDataProvider.inceptionVaultsCore() (contracts/
inceptionVaultCount() should be declared external:
 - InceptionVaultsDataProvider.inceptionVaultCount() (contracts/
baseDebt() should be declared external:
 - InceptionVaultsDataProvider.baseDebt() (contracts/inception/I
a() should be declared external:
 - ChainlinkInceptionPriceFeed.a() (contracts/inception/priceFee
inceptionCollateral() should be declared external:
 - ChainlinkInceptionPriceFeed.inceptionCollateral() (contracts/
assetOracle() should be declared external:
 - ChainlinkInceptionPriceFeed.assetOracle() (contracts/inceptic
eurOracle() should be declared external:
 - ChainlinkInceptionPriceFeed.eurOracle() (contracts/inception/
deposit(uint256) should be declared external:
 - DemandMinerV2.deposit(uint256) (contracts/liquidityMining/v2/
withdraw(uint256) should be declared external:
 - DemandMinerV2.withdraw(uint256) (contracts/liquidityMining/v2
token() should be declared external:
 - DemandMinerV2.token() (contracts/liquidityMining/v2/DemandMir
feeCollector() should be declared external:
 - DemandMinerV2.feeCollector() (contracts/liquidityMining/v2/De
feeConfig() should be declared external:
 - DemandMinerV2.feeConfig() (contracts/liquidityMining/v2/Demar
releaseRewards(address) should be declared external:
 - GenericMinerV2.releaseRewards(address) (contracts/liquidityMi

- PARMinerV2.releaseRewards(address) (contracts/liquidityMining
updateBoost(address) should be declared external:
- GenericMinerV2.updateBoost(address) (contracts/liquidityMinir
stake(address) should be declared external:
- GenericMinerV2.stake(address) (contracts/liquidityMining/v2/G
- PARMinerV2.stake(address) (contracts/liquidityMining/v2/PARMi
stakeWithBoost(address) should be declared external:
- GenericMinerV2.stakeWithBoost(address) (contracts/liquidityMi
- PARMinerV2.stakeWithBoost(address) (contracts/liquidityMining
pendingMIMO(address) should be declared external:
- GenericMinerV2.pendingMIMO(address) (contracts/liquidityMinir
- PARMinerV2.pendingMIMO(address) (contracts/liquidityMining/v2
pendingPAR(address) should be declared external:
- GenericMinerV2.pendingPAR(address) (contracts/liquidityMining
- PARMinerV2.pendingPAR(address) (contracts/liquidityMining/v2/
par() should be declared external:
- GenericMinerV2.par() (contracts/liquidityMining/v2/GenericMir
- PARMinerV2.par() (contracts/liquidityMining/v2/PARMinerV2.sol
a() should be declared external:
- GenericMinerV2.a() (contracts/liquidityMining/v2/GenericMiner
- PARMinerV2.a() (contracts/liquidityMining/v2/PARMinerV2.sol#2
boostConfig() should be declared external:
- GenericMinerV2.boostConfig() (contracts/liquidityMining/v2/Ge
- PARMinerV2.boostConfig() (contracts/liquidityMining/v2/PARMir
totalStake() should be declared external:
- GenericMinerV2.totalStake() (contracts/liquidityMining/v2/Ger
- PARMinerV2.totalStake() (contracts/liquidityMining/v2/PARMine
totalStakeWithBoost() should be declared external:
- GenericMinerV2.totalStakeWithBoost() (contracts/liquidityMini
- PARMinerV2.totalStakeWithBoost() (contracts/liquidityMining/v
userInfo(address) should be declared external:
- GenericMinerV2.userInfo(address) (contracts/liquidityMining/v
- PARMinerV2.userInfo(address) (contracts/liquidityMining/v2/PA
deposit(uint256) should be declared external:
- PARMinerV2.deposit(uint256) (contracts/liquidityMining/v2/PAR
withdraw(uint256) should be declared external:
- PARMinerV2.withdraw(uint256) (contracts/liquidityMining/v2/PA
liquidate(uint256,uint256,uint256,bytes) should be declared exte
- PARMinerV2.liquidate(uint256,uint256,uint256,bytes) (contract
restakePAR(address) should be declared external:
- PARMinerV2.restakePAR(address) (contracts/liquidityMining/v2/
updateBoost(address) should be declared external:
- PARMinerV2.updateBoost(address) (contracts/liquidityMining/v2
liquidateCallerReward() should be declared external:
- PARMinerV2.liquidateCallerReward() (contracts/liquidityMining
baseDebtChanged(address,uint256) should be declared external:

```
 - SupplyMinerV2.baseDebtChanged(address,uint256) (contracts/lic
collateral() should be declared external:
 - SupplyMinerV2.collateral() (contracts/liquidityMining/v2/Supp
syncStake(address) should be declared external:
 - VotingMinerV2.syncStake(address) (contracts/liquidityMining/v
```

[G-15] No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint`, `false` for `bool`, `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
core/contracts/inception/InceptionVaultsCore.sol:218:    uint256
core/contracts/libraries/ABDKMath64x64.sol:153:      bool negati
core/contracts/libraries/ABDKMath64x64.sol:222:    bool negative
core/contracts/libraries/ABDKMath64x64.sol:387:      uint256 res
core/contracts/libraries/ABDKMath64x64.sol:437:    int256 msb =
```

I suggest removing explicit initializations for default values.

[G-16] Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g.,

> `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
supervaults/contracts/SuperVault.sol:39:    require(hasRole(DEFA
supervaults/contracts/SuperVault.sol:56:    require(address(_a)
supervaults/contracts/SuperVault.sol:57:    require(address(_ga)
supervaults/contracts/SuperVault.sol:58:    require(address(_ler
supervaults/contracts/SuperVault.sol:59:    require(address(dexA
supervaults/contracts/SuperVault.sol:83:    require(msg.sender =
supervaults/contracts/SuperVault.sol:109:    require(token.balar
supervaults/contracts/SuperVault.sol:156:    require(fromCollate
supervaults/contracts/SuperVault.sol:207:    require(vaultCollat
supervaults/contracts/SuperVault.sol:233:    require(IERC20(a.st
supervaults/contracts/SuperVault.sol:247:    require(asset.trans
supervaults/contracts/SuperVault.sol:255:    require(IERC20(a.st
supervaults/contracts/SuperVault.sol:264:    require(token.trans
supervaults/contracts/SuperVault.sol:292:    require(IERC20(a.st
supervaults/contracts/SuperVault.sol:313:    require(IERC20(a.st
supervaults/contracts/SuperVault.sol:344:    require(proxy != ac
supervaults/contracts/SuperVault.sol:370:      require(ga.mimo()
supervaults/contracts/SuperVaultFactory.sol:18:    require(addre
```

I suggest replacing revert strings with custom errors.

m19 (Mimo DeFi) commented:

> This was a very thorough gas optimization report and is very helpful for us.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-

risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top