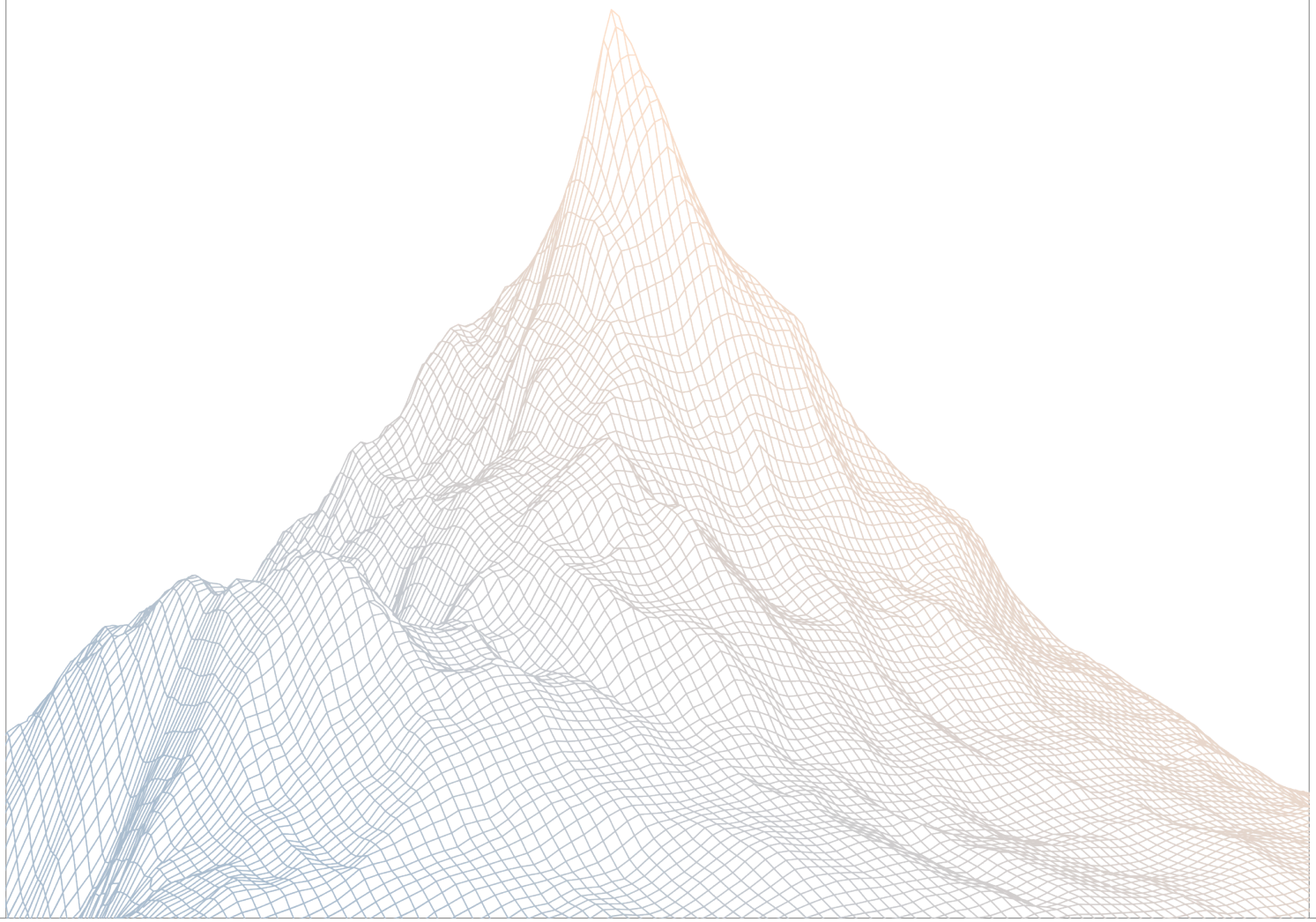
Zenith

MORE Vaults

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: May 2nd to May 20th, 2025

AUDITED BY: ether_sky
zzykxx

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About More Vaults

MORE Vaults is an on-chain portfolio-construction and asset-management protocol that implements the EIP-2535 Diamond proxy standard to provide a fully modular, upgradeable smart-contract architecture.

Each vault is a Diamond composed of specialized facets. Every facet handles a discrete class of functionality such as accounting logic, selectors, risk controls, or functions. Because facets can embed adapters to third-party DeFi platforms, a single vault can deploy capital across multiple protocols while keeping unified, transparent accounting.

The protocol is built to streamline the workflow of professional traders, fund managers, and automated strategies. By sharply reducing the operational friction of rebalancing or reallocating capital as market conditions evolve, MORE Vaults lets curators focus on alpha generation rather than low-level contract maintenance. Depositors receive a single receipt token that continually tracks their pro-rata claim on the strategy.

2.2 Scope

The engagement involved a review of the following targets:

Target	More-Vaults
Repository	https://github.com/deathwing00000/More-Vaults
Commit Hash	`00d6a328ab8d2c5d21f896773fb98c41c7056364`

Files	`facets/*`
	`registry/*`
	`factory/*`
	`libraries/*`
	`MoreVaultsDiamond.sol`

Target	Morigami
Repository	https://github.com/deathwing00000/Morigami
Commit Hash	`925fcf71904a1dbb95579fdced59513af0af5f49`
Files	`Changes on top of Origami fork`

2.3 Audit Timeline

May 2, 2025	Audit start
May 20, 2025	Audit end
June 16, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	4
High Risk	9
Medium Risk	18
Low Risk	8
Informational	7
Total Issues	**46**

3

Findings Summary

ID	Description	Status
C-1	The latestPrice function returns the price in the decimal of the quote token	Resolved
C-2	UniswapV2 LP value can be manipulated by an attacker in UniswapV2Facet::accountingUniswapV2Facet()	Resolved
C-3	accountingCurveFacet() uses curve's calc_withdraw_one_coin() to price LPs which can be manipulated by attackers	Resolved
C-4	The conversion to underlying tokens does not account for token decimals	Resolved
H-1	In MorigamiGenericLpPool, the actual token amounts can exceed the specified minimum amounts	Resolved
H-2	The MultiRewards contract does not include a getRewardTokens function	Acknowledged
H-3	The debtBalance function in the MorigamiAaveV3BorrowAndLendMultiBorrowTokens contract contains an error	Resolved
H-4	MoreVaultsLib::removeTokenIfnecessary() doesn't take into account staked tokens	Resolved
H-5	There is a bug in the liabilities function of the MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens contract	Resolved
H-6	The input token validation in CurveFacet is incorrect	Resolved
H-7	MorigamiUniswapV2LpTokenOracle::latestPrice() result can be manipulated by an attacker	Resolved
H-8	MorigamiCurveLpTokenOracle::latestPrice() uses curve's calc_withdraw_one_coin() to price LPs, which can be manipulated	Resolved
H-9	The liquidity token is incorrect when depositing a native token into the Uniswap V2 pool	Resolved
M-1	Vault owners can change a vault registry from permissioned to permissionless at any time	Resolved

ID	Description	Status
M-2	VaultFacet::mint() and VaultFacet::redeem() round in the wrong direction	Resolved
M-3	UniswapV2Facet doesn't ensure ETH is an available asset when performing ETH operations	Resolved
M-4	Unbounded looping can lead to protocol DOS	Resolved
M-5	MoreVaultsLib::removeFunction() doesn't remove facet address from ds. facetsForAccounting leading to a DOS	Resolved
M-6	accountingMultiRewardsFacet() doesn't work properly if a reward token that is not an available assets is added to a multirewards contract	Resolved
M-7	accountingCurveLiquidityGaugeV6Facet() doesn't work properly if a reward token that is not an available assets is added to a gauge	Resolved
M-8	accountingCurveLiquidityGaugeV6Facet() doesn't account for CRV rewards	Resolved
M-9	accountingCurveFacet() and accountingUniswapV2Facet() can account for staked LP tokens even if the vault doesn't control them anymore	Resolved
M-10	Use the SafeERC20 library for handling token approvals	Resolved
M-11	AaveV3Facet::flashloan() can leave debt tokens whose underlying assets are not available assets	Resolved
M-12	accountingAaveV3Facet() doesn't account for unclaimed rewards	Resolved
M-13	accountingAaveV3Facet() can revert for underflow leading to a DOS on vault operations	Resolved
M-14	The AccessControlFacet should also be added alongside the DiamondCutFacet in the constructor of MoreVaultsDiamond	Resolved
M-15	MORELeverageFacet is incompatible with MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens	Acknowledged

ID	Description	Status
M-16	_rebalanceUpFlashLoanCallback() uses single params.repaySurplusThreshold parameter for every token	Resolved
M-17	When adding liquidity to a non-existent Uniswap V2 pool, address(0) may be mistakenly held as the LP token	Resolved
M-18	Vaults allow to front-run and back-run transactions to either profit or avoid losses	Resolved
L-1	Inflation attack is still possible in case of multiple deposits from users	Resolved
L-2	The VaultFacet currently does not support native token deposits	Resolved
L-3	CurveFacet::_exchange() doesn't take in consideration adding/removing liquidity with swap_type either 5 and 7	Resolved
L-4	Hardcoded 3 hours staleness check for all oracles	Resolved
L-5	The validity of individual function selectors is not checked when adding a new facet to the vault	Resolved
L-6	The mTokens should be validated for removability within the repayWithATokens function	Resolved
L-7	flashLoanCallback() forces to flashloan all available tokens when rebalancing	Resolved
L-8	Morigami protocol attempts to set elements of uninitialized arrays	Resolved
I-1	It's impossible to add selectors to an already existing facet	Resolved
I-2	ConfigurationFacet::setFee() allows to change fees at any point with immediate effect	Resolved
I-3	Lack of 2-step ownership transfer	Resolved

ID	Description	Status
I-4	accountingAaveV3Facet() rounds debt value in the wrong direction	Resolved
I-5	accountingAaveV3Facet() double-counts the value of an aToken that's an available asset	Resolved
I-6	The vetoActions() function can be improved by allowing to veto multiple actions at once	Resolved
I-7	Vault curator has multiple ways to steal funds from a vault	Resolved

4

Findings

4.1 Critical Risk

A total of 4 critical risk findings were identified.

[C-1] The latestPrice function returns the price in the decimal of the quote token

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MorigamiCurveLpTokenOracle.sol

Description:

Suppose the **base asset** is a **Curve LP token** with **18 decimals**, and the **quote asset** is **USDC**, which has **6 decimals**. In this case, the `assetScalingFactor` is calculated as: `10^(18 + 18 - 6) = 10^30`

- MorigamiOracleBase.sol#L52

```
uint8 public constant override decimals = 18;

constructor(BaseOracleParams memory params) {
    description = params.description;
    baseAsset = params.baseAssetAddress;
    quoteAsset = params.quoteAssetAddress;
    if (params.quoteAssetDecimals > decimals + params.baseAssetDecimals)
    revert CommonEventsAndErrors.InvalidParam();
    assetScalingFactor = 10 ** (decimals + params.baseAssetDecimals
    - params.quoteAssetDecimals);
}
```

According to the comments in the code, the `latestPrice` function is expected to return the `latest oracle price` **in 18-decimal precision**.

- MorigamiOracleBase.sol#L56

```
/**
    * @notice Return the latest oracle price, to `decimals` precision
    * @dev This may still revert - eg if deemed stale, div by 0, negative
    price
    * @param priceType What kind of price - Spot or Historic
    * @param roundingMode Round the price at each intermediate step such that
    the final price rounds in the specified direction.
    */
function latestPrice(
    PriceType priceType,
    MorigamiMath.Rounding roundingMode
) public virtual override view returns (uint256 price);
```

However, in the `MorigamiCurveLpTokenOracle`, the `latestPrice` function returns the `amount of the quote asset` (`USDC`) for **1 LP token (the base asset)** as the `latest Oracle price`. This `price` is expressed using the **decimals of the quote token**, which is **6 decimals** in the case of `USDC`.

- MorigamiCurveLpTokenOracle.sol#L39-L42

```
/**
    * @notice Return the current conversion rate a.k.a price for 1 lpToken in
    quoteAsset
    * @dev The price is calculated as the amount of quoteAsset that would be
    received for 1 lpToken(baseAsset)
    */
function latestPrice(
    PriceType,
    MorigamiMath.Rounding
) public view override returns (uint256 price) {
    // should take into account decimals difference between baseAsset and
    quoteAsset
    price = curvePool.calc_withdraw_one_coin(
        1 * 10 ** (IERC20Metadata(baseAsset).decimals()),
        qouteTokenIndex
    );
}
```

Let's say:

- `1 LP token` is worth `5 USDC`.
- So, `latestPrice()` returns `5 * 10^6`.

Now, if we convert **5 LP tokens** (which is `5 * 10^18` in `18-decimal format`) using the `convertAmount` function, the expected result is: `5 LP * 5 USDC = 25 USDC = 25 * 10^6 (in USDC decimals)`

But what the function actually computes is: `(5 * 10^18) * (5 * 10^6) / 10^30 = 25 * 10^(-6) = 0.000025 USDC`

- MorigamiOracleBase.sol#L106-L110

```
function convertAmount(
    address fromAsset,
    uint256 fromAssetAmount,
    PriceType priceType,
    MorigamiMath.Rounding roundingMode
) external override view returns (uint256 toAssetAmount) {
    if (fromAsset == baseAsset) {
        // The numerator needs to round in the same way to be conservative
        uint256 _price = latestPrice(
            priceType,
            roundingMode
        );

        return fromAssetAmount.mulDiv(
            _price,
            assetScalingFactor,
            roundingMode
        );
    }
}
```

Incorrect because the `latestPrice()` returned a value in `6-decimal precision`, not `18`, causing the scaling to be off by a factor of `10^12`. The same issue also occurs in the `MorigamiUniswapV2LpTokenOracle`.

Recommendations:

```
function latestPrice(
    PriceType,
    MorigamiMath.Rounding
) public view override returns (uint256 price) {
    // should take into account decimals difference between baseAsset and
    quoteAsset
    price = curvePool.calc_withdraw_one_coin(
        1 * 10 ** (IERC20Metadata(baseAsset).decimals()),
        qouteTokenIndex
    );
    price = price * 10 ** decimals / 10 **
(IERC20Metadata(quoteAsset).decimals());
```

```
}
```

MORE Vaults: Resolved with @96c47b4...

Zenith: Verified.

[C-2] UniswapV2 LP value can be manipulated by an attacker in `UniswapV2Facet::accountingUniswapV2Facet()`

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- UniswapV2Facet

Description:

The function UniswapV2Facet::accountingUniswapV2Facet() calculates the value of UniswapV2 LP tokens by querying the reserves of a pool.

The reserves of a UniswapV2 pool are spot balances that can be manipulated by an attacker.

Here's an example attack scenario:

1. Vault curator adds liquidity to a WETH/USDC pool and the vault receives LP tokens
2. An attacker deposits in the vault and receives shares
3. The attacker swaps a huge amount of WETH for USDC directly in the WETH/USDC pool, which increases the total value of the pool and the valuation of LPs
4. The attacker redeems his shares from the vault and receives vault assets. Because of the manipulation the LPs are now worth more than they should.
5. The attacker swaps back USDC for WETH in the WETH/USDC pool, rebalancing it

As a practical example on why this works let's take the following UniswapV2 pool (WETH/USDC):

- USDC: amount: 10000000, value: ~10000000$ (1 USDC = 1$)
- WETH: amount: 5000, value: ~10000000$ (1 WETH = 2000$)
- Total pool value: 10000000$ + 10000000$ = 20000000$

The attacker now swaps 4000 WETH for USDC, which according to UniswapV2 `x*y=k` formula will swap to: $$(10000000 - USDC) \cdot (5000 + 4000) = 50000000000$$ $$USDC = 4444444$$

New pool state:

- USDC: amount: 10000000-4444444=5555556, value: ~5555556$ (1 USDC = 1$)
- WETH: amount: 5000+4000=9000, value: ~18000000$ (1 WETH = 2000$)
- Total pool value: 18000000$ + 5555556$ = 23555556$

The total value of the pool is greater after the swap, this implies the LP will be priced at a higher price than they should by UniswapV2Facet::accountingUniswapV2Facet().

Recommendations:

Price UniswapV2 LPs using formulas that can't be manipulated. Here's a good resource on how to achieve that.

MORE Vaults: Resolved @fd104a...

Zenith: Verified.

[C-3] `accountingCurveFacet()` uses curve's `calc_withdraw_one_coin()` to price LPs which can be manipulated by attackers

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- CurveFacet

Description:

The function CurveFacet::accountingCurveFacet() uses curve's calc_withdraw_one_coin() in order to price LP tokens:

```
uint256 lpTokenBalance = IERC20(lpToken).balanceOf(address(this)) +
    ds.staked[lpToken];
sum += MoreVaultsLib.convertToUnderlying(
    ICurveViews(lpToken).coins(0),
    (
        ICurveViews(lpToken).calc_withdraw_one_coin(
            lpTokenBalance,
            int128(0)
        )
    )
```

The function calc_withdraw_one_coin() returns the amount of tokens users should get when withdrawing LPs but it uses spot balances for calculations. Attackers can manipulate the returned value by imbalancing the pool (trading huge amounts of tokens) in order to profit.

Recommendations:

Use curve's get_virtual_price() to price LP tokens.

Keep in mind that get_virtual_price() can also be subject to manipulation via read-only reentrancy, in order to protect from this attack the protocol can call a function that triggers the reentrancy lock in the curve pool, such as remove_liquidity():

1. Call remove_liquidity() by passing `0` as `_amount`
2. Call get_virtual_price()

MORE Vaults: Resolved with @7588ddc...

Zenith: Verified.

[C-4] The conversion to underlying tokens does not account for token decimals

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MoreVaultsLib.sol

Description:

Accurate conversion to `underlying tokens` is critical for correctly tracking `total assets` in the `vaults`. However, the current `convertToUnderlying` function does not properly account for `token decimals`, leading to incorrect `asset` valuations.

- MoreVaultsLib.sol#L195-L198

```
function convertToUnderlying(
    address _token,
    uint amount
) internal view returns (uint) {
    IMoreVaultsRegistry registry = IMoreVaultsRegistry(
        AccessControlLib.vaultRegistry()
    );
    IAaveOracle oracle = registry.oracle();
    address oracleDenominationAsset = registry.getDenominationAsset();
    IAggregatorV2V3Interface aggregator = IAggregatorV2V3Interface(
        oracle.getSourceOfAsset(_token)
    );
178:    uint256 inputTokenPrice = getLatestPrice(aggregator);
179:    uint8 inputTokenOracleDecimals = aggregator.decimals();

    uint256 finalPriceForConversion = inputTokenPrice;
    if (underlyingToken ≠ oracleDenominationAsset) {
        aggregator = IAggregatorV2V3Interface(
            oracle.getSourceOfAsset(underlyingToken)
        );
        uint256 underlyingTokenPrice = getLatestPrice(aggregator);
        uint8 underlyingTokenOracleDecimals = aggregator.decimals();
        uint256 inputToUnderlyingPrice = inputTokenPrice.mulDiv(
```

```
            10 ** underlyingTokenOracleDecimals,
            underlyingTokenPrice
        );
        finalPriceForConversion = inputToUnderlyingPrice;
    }

195:    uint256 convertedAmount = amount.mulDiv(
        finalPriceForConversion,
        10 ** inputTokenOracleDecimals
    );

    return convertedAmount;
}
```

For example, the `underlying token` is `USDC` with `6 decimals` and the `converted token` is `WETH` with `18 decimals`. And the `WETH price` is `1000 USDC` and the `oracle decimal` is `8`. At **line 178**, the `price` from the `oracle` is: `inputTokenPrice = 1000 * 1e8`. At **line 179**, the `oracle's decimal` precision is: `inputTokenOracleDecimals = 8`. Now, assume the conversion amount is **100 WETH**, which equals: `amount = 100 * 1e18`. At **line 195**, the converted amount is computed as:

```
convertedAmount = amount * inputTokenPrice
    / (10 ** inputTokenOracleDecimals)
                = 100 * 1e18 * 1000 * 1e8 / 1e8
                = 100000 * 1e18
```

This result is incorrect. The expected converted amount, in terms of the **underlying token's decimals (USDC, 6)**, should be `100000 * 1e6`.

Recommendations:

```
function convertToUnderlying(
    address _token,
    uint amount
 ) internal view returns (uint) {

 -    uint256 convertedAmount = amount.mulDiv(
 -        finalPriceForConversion,
 -        10 ** inputTokenOracleDecimals
 -    );
    uint256 convertedAmount = amount.mulDiv(

     finalPriceForConversion * 10 ** IERC20Metadata(underlyingToken).decimals(),
```

```
        10 ** (inputTokenOracleDecimals +
    IERC20Metadata(_token).decimals())
        );

        return convertedAmount;
    }
```

MORE Vaults: Resolved with @5185828...

Zenith: Verified.

4.2 High Risk

A total of 9 high risk findings were identified.

[H-1] In MorigamiGenericLpPool, the actual token amounts can exceed the specified minimum amounts

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- MorigamiGenericLpPool.sol

Description:

In the `execute` function of the `MorigamiGenericLpPool`, the specified `amounts` represent the `minimum expected withdrawal amounts` when `removing liquidity`.

- MorigamiGenericLpPool.sol#L104

```
function execute(
    IERC20 lpToken,
    IERC20[] memory tokensInPool,
    uint256[] memory amounts,
    uint256 amountLpTokenDesired,
    Action actionType, // add is true, remove is false
    bytes calldata lpPoolData
) external override {
    RouteData memory routeData = abi.decode(lpPoolData, (RouteData));
    if (!whitelistedRouters[routeData.router])
        revert InvalidRouter(routeData.router);

    if (actionType == Action.Add) {
        for (uint256 i = 0; i < tokensInPool.length; i++) {
            tokensInPool[i].safeTransferFrom(
                msg.sender,
                address(this),
                amounts[i]
```

```
            );
        }

        routeData.router.addLiquidity(
            lpToken,
            tokensInPool,
            amounts,
            amountLpTokenDesired,
            routeData.data
        );

        // Transfer back to the caller
        lpToken.safeTransfer(msg.sender, amountLpTokenDesired);
    } else {
        lpToken.safeTransferFrom(
            msg.sender,
            address(this),
            amountLpTokenDesired
        );

        routeData.router.removeLiquidity(
            lpToken,
            tokensInPool,
            amounts,
            amountLpTokenDesired,
            routeData.data
        );

        // Transfer back to the caller
        for (uint256 i = 0; i < tokensInPool.length; ) {
            tokensInPool[i].safeTransfer(msg.sender, amounts[i]);
            unchecked {
                ++i;
            }
        }
    }
}
```

These values are passed to the `removeLiquidity` function of the `LpTokenPool`, where they are used for `slippage checks`.

- LpTokenPool.sol#L99-L102

```
function removeLiquidity(
    address router,
    IERC20 lpToken,
```

```
    IERC20[] memory tokensInPool,
    uint256[] memory amountsOutMin,
    uint256 amountLpTokenToBurn,
    bytes memory removeLiquidityData
) internal {

    for (uint256 i = 0; i < tokensInPool.length; ) {
        if (
            tokensInPool[i].balanceOf(address(this)) - _initialBalances[i] <
            amountsOutMin[i]
        ) {
            revert InvalidRemoveLiquidity();
        }
        unchecked {
            ++i;
        }
    }
}
```

Naturally, the `actual withdrawal amounts` may be greater than these minimums.

However, instead of transferring the `actual withdrawn amounts` to the user, the function currently transfers only the `minimum specified amounts`, which is incorrect.

Similarly, during `liquidity deposits`, the logic may not `deposit` all the provided tokens if fewer tokens are needed to meet the `desired liquidity`, resulting in `some tokens remaining unutilized`.

Recommendations:

When `depositing liquidity`, any unused tokens should be returned to the user. Similarly, when `removing liquidity`, the actual `withdrawn` token amounts should be transferred to the user.

MORE Vaults: Resolved withs @55e5315... and @f7ea8e1...

Zenith: Verified.

[H-2] The MultiRewards contract does not include a getRewardTokens function

SEVERITY: High	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- MultiRewardsFacet.sol

Description:

Below is the confirmed source code of the `MultiRewards` contract.

- MultiRewards.sol

This contract **does not implement** the `getRewardTokens` function. However, the `getRewardTokens` function **is called** within the `MultiRewardsFacet` contract. MultiRewardsFacet.sol#L48

```
function accountingMultiRewardsFacet() external view returns (uint256 sum) {
    for (uint256 i = 0; i < stakings.length(); ) {
        IMultiRewards staking = IMultiRewards(stakings.at(i));
        address[] memory rewardTokens = staking.getRewardTokens();
        unchecked {
            ++i;
        }
    }
}
```

Recommendations:

Update the `IMultiRewards` interface to access the `rewardTokens` variable of the `MultiRewards`, instead of calling non-existent `getRewardTokens` function.

MORE Vaults: Acknowledged, the protocol won't interact with a multi reward contract that doesn't implement `getRewardTokens()`.

[H-3] The debtBalance function in the MorigamiAaveV3BorrowAndLendMultiBorrowTokens contract contains an error

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol

Description:

In the `MorigamiAaveV3BorrowAndLendMultiBorrowTokens`, there are multiple tokens that can be `borrowed` from the `Aave Pool`.

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol#L45

```
address[] public _borrowTokens;
```

For each `borrowed token`, `Aave` issues a corresponding **Debt Token**, whose `balance` reflects the **actual amount of that token currently borrowed**.

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol#L55

```
IERC20Metadata[] public _aaveDebtTokens;
```

The `_validateBorrowToken` function correctly checks whether a given token is among the allowed `borrowable tokens`.

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol#L581

```
function _validateBorrowToken(
    address _borrowToken
) internal view returns (bool) {
    for (uint256 i = 0; i < _borrowTokens.length; ) {
        if (_borrowToken == _borrowTokens[i]) return true;
        unchecked {
            ++i;
        }
```

```
    }
    return false;
}
```

However, the `debtBalance` function is implemented incorrectly. It retrieves the `balance` of the **underlying borrowable token**, rather than the `balance` of the corresponding **Aave Debt Token**.

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol#L393

```
function debtBalance(
    address debtToken_
) public view override returns (uint256) {
    _validateBorrowToken(debtToken_);
    return IERC20(debtToken_).balanceOf(address(this));
}
```

This is a problem because:

- When tokens are `borrowed` from `Aave`, the `balance` of the **Debt Token** increases—not the `underlying token`.
- The **Debt Token balance** accurately reflects the current `debt`.

As a result, the `debtBalance` function often returns **zero** or an incorrect value, since it's referencing the wrong `token balance`.

This miscalculation affects the `MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens` contract, as it uses the `debtBalance` function to compute **total liabilities**—leading to incorrect liability values.

- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol#L371

```
function liabilities(
    IMorigamiOracle.PriceType debtPriceType
)
    public
    view
    override(
        MorigamiAbstractLovTokenManagerMultiBorrowTokens,
        IMorigamiLovTokenManagerMultiBorrowTokens
    )
    returns (uint256)
{
    // Convert the [debtTokens] into the [reserveToken] terms
    uint256 totalDebtInReserveToken = 0;
    for (uint256 i = 0; i < _debtTokens.length; ) {
```

```
@->         uint256 debt = borrowLend.debtBalance(address(_debtTokens[i]));

        if (debt == 0) continue;
        totalDebtInReserveToken += _debtTokenToReserveTokenOracles[i]
            .convertAmount(
                address(_debtTokens[i]),
                debt,
                debtPriceType,
                MorigamiMath.Rounding.ROUND_UP
            );
        unchecked {
            ++i;
        }
    }

    return totalDebtInReserveToken;
}
```

Recommendations:

```
function debtBalance(
    address debtToken_
) public view override returns (uint256) {
    _validateBorrowToken(debtToken_);

    for (uint256 i = 0; i < _borrowTokens.length; ) {
        if (debtToken_ == _borrowTokens[i]) return _aaveDebtTokens[i].
            balanceOf(address(this));
        unchecked {
            ++i;
        }
    }

    return IERC20(debtToken_).balanceOf(address(this));
    return 0;
}
```

MORE Vaults: Resolved with @f02e4d9...

Zenith: Verified.

[H-4] `MoreVaultsLib::removeTokenIfnecessary()` doesn't take into account staked tokens

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MoreVaultsLib

Description:

The function MoreVaultsLib::removeTokenIfnecessary() removes tokens from the `ds.tokensHeld` mapping when the balance of the specified token is lower than `10e3`.

This doesn't take into account staked tokens (ie. `ds.staked[token] > 0`).

Let's take as example curve's LP tokens:

1. Vault curator stakes some assets in curve's pool and gets `100e18` LP tokens
2. Vault curator stakes `70e18` of the LP tokens in the gauge, this sets `ds.staked[token]` to `70e18`
3. Vault curator redeems `30e18` LP tokens from the pool for assets
4. The CurveFacet::_exchange() function executes MoreVaultsLib::removeTokenIfnecessary() which removes the LP tokens from the `ds.tokensHeld[CURVE_LP_TOKENS_ID]` as the balance in the vault is `0`
5. The function CurveFacet::accountingCurveFacet() doesn't account for the LP staked in the gauge anymore as the curve's LP token has been removed from `ds.tokensHeld[CURVE_LP_TOKENS_ID]`

This results in the vault losing value instantly and users receiving more shares than they should when depositing and less assets than they should when withdrawing.

Recommendations:

In MoreVaultsLib::removeTokenIfnecessary() don't remove tokens from `ds.tokensHeld` when `ds.staked[token]` is bigger than `0` or a small arbitrary value:

```
if (IERC20(token).balanceOf(address(this)) + ds.staked[token] < 10e3) {
   tokensHeld.remove(token);
}
```

MORE Vaults: Resolved with @639f2e...

Zenith: Verified.

[H-5] There is a bug in the liabilities function of the MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens contract

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol

Description:

If the `balance` of any `debt token` is zero, the `liabilities` function will revert due to an `infinite loop`.

- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol#L372

```
function liabilities(
    IMorigamiOracle.PriceType debtPriceType
)
    public
    view
    override(
        MorigamiAbstractLovTokenManagerMultiBorrowTokens,
        IMorigamiLovTokenManagerMultiBorrowTokens
    )
    returns (uint256)
{
    // Convert the [debtTokens] into the [reserveToken] terms
    uint256 totalDebtInReserveToken = 0;
    for (uint256 i = 0; i < _debtTokens.length; ) {
        uint256 debt = borrowLend.debtBalance(address(_debtTokens[i]));
@->         if (debt == 0) continue;
        totalDebtInReserveToken += _debtTokenToReserveTokenOracles[i]
            .convertAmount(
                address(_debtTokens[i]),
                debt,
                debtPriceType,
                MorigamiMath.Rounding.ROUND_UP
            );
```

```
        unchecked {
            ++i;
        }
    }

    return totalDebtInReserveToken;
}
```

Since this function is used across all operations, the impact of the bug is significant.

Recommendations:

```
function liabilities(
    IMorigamiOracle.PriceType debtPriceType
)
    public
    view
    override(
        MorigamiAbstractLovTokenManagerMultiBorrowTokens,
        IMorigamiLovTokenManagerMultiBorrowTokens
    )
    returns (uint256)
{
    // Convert the [debtTokens] into the [reserveToken] terms
    uint256 totalDebtInReserveToken = 0;
    for (uint256 i = 0; i < _debtTokens.length; ) {
        uint256 debt = borrowLend.debtBalance(address(_debtTokens[i]));
        if (debt == 0) continue;
         if (debt == 0) {
            unchecked {
                ++i;
            }
        }


        totalDebtInReserveToken += _debtTokenToReserveTokenOracles[i]
            .convertAmount(
                address(_debtTokens[i]),
                debt,
                debtPriceType,
                MorigamiMath.Rounding.ROUND_UP
            );
        unchecked {
```

```
            ++i;
        }
    }

    return totalDebtInReserveToken;
}
```

MORE Vaults: Resolved with @43830ed...

Zenith: Verified.

[H-6] The input token validation in CurveFacet is incorrect

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- CurveFacet.sol

Description:

The swap operation in the Curve Router can consist of multiple smaller steps. For example, to swap token A to token C, the path might go through A → B → C. In this case:

- A is the **input token**,
- C is the **final output token**,
- B is an **intermediate token**, serving as input of the second.
- Router.vy#L343

```
def exchange(
    _route: address[11],
    _swap_params: uint256[5][5],
    _amount: uint256,
    _min_dy: uint256,
    _pools: address[5]=empty(address[5]),
    _receiver: address=msg.sender
) → uint256:
    for i in range(5):
        # 5 rounds of iteration to perform up to 5 swaps
        swap: address = _route[i * 2 + 1]
        pool: address = _pools[i]  # Only for Polygon meta-factories
    underlying swap (swap_type == 6)
        output_token = _route[(i + 1) * 2]
        params: uint256[5] = _swap_params[i]  # i, j, swap_type, pool_type,
    n_coins

        input_token = output_token
```

Within the CurveFacet contract, the _getOutputTokenAddressAndIndexOfLastSwap function returns (1, C) — indicating that the last step produces token C.

- CurveFacet.sol#L197-L198

```
function _getOutputTokenAddressAndIndexOfLastSwap(
    address[11] calldata _route
) internal pure returns (uint256 i, address outputToken) {
    while (i < 4 && _route[i * 2 + 3] ≠ address(0)) i++;
    outputToken = _route[(i + 1) * 2];
}
```

In the `exchange` and `exchangeNg` functions, `_swap_params[1]` is incorrectly assumed to relate to `input token A`.

- CurveFacet.sol#L153-L155

```
function exchange(
    address curveRouter,
    address[11] calldata _route,
    uint256[5][5] calldata _swap_params,
    uint256 _amount,
    uint256 _min_dy,
    address[5] calldata _pools
) external payable returns (uint256) {
    AccessControlLib.validateDiamond(msg.sender);
    address inputToken = _route[0];
    (
        uint256 index,
        address outputToken
    ) = _getOutputTokenAddressAndIndexOfLastSwap(_route);

    // If not remove liquidity - validate input token
@->    if (_swap_params[index][2] ≠ 6) {
        MoreVaultsLib.validateAssetAvailable(inputToken);
    }

    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    if (_swap_params[index][2] == 4) {
        ds.tokensHeld[CURVE_LP_TOKENS_ID].add(outputToken);
@->    } else if (_swap_params[index][2] == 6) {
        MoreVaultsLib.removeTokenIfnecessary(
            ds.tokensHeld[CURVE_LP_TOKENS_ID],
            inputToken
        );
    }
    return receivedAmount;
}
```

In reality:

- `_swap_params[0]` corresponds to `token A` (the actual `input`),
- `_swap_params[1]` corresponds to `token B` (the `intermediate token`).

Because of this misinterpretation, the `input token validation` can be bypassed. Specifically, even if `_swap_params[0][2]` is **not** 6 (which it should be for `valid input tokens`), the check might still pass if `_swap_params[1][2]` **is** 6 — since that value is mistakenly used for validation. More critically, if `_swap_params[0][2]` is 0 and `_swap_params[1][2]` is non-zero, the operation is the `removing liquidity`. However, the `input token validation` will block this if the `LP token` is not listed as an available `asset`.

Recommendations:

```
function exchange(
    address curveRouter,
    address[11] calldata _route,
    uint256[5][5] calldata _swap_params,
    uint256 _amount,
    uint256 _min_dy,
    address[5] calldata _pools
) external payable returns (uint256) {
    AccessControlLib.validateDiamond(msg.sender);
    address inputToken = _route[0];
    (
        uint256 index,
        address outputToken
    ) = _getOutputTokenAddressAndIndexOfLastSwap(_route);

    // If not remove liquidity - validate input token
    if (_swap_params[index][2] ≠ 6) {
     if (_swap_params[0][2] ≠ 6) {
        MoreVaultsLib.validateAssetAvailable(inputToken);
    }

    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    if (_swap_params[index][2] == 4) {
        ds.tokensHeld[CURVE_LP_TOKENS_ID].add(outputToken);
    } else if (_swap_params[index][2] == 6) {
    }
    if (_swap_params[0][2] == 6) {
        MoreVaultsLib.removeTokenIfnecessary(
            ds.tokensHeld[CURVE_LP_TOKENS_ID],
```

```
            inputToken
        );
    }
    return receivedAmount;
}
```

The same fix should also be applied to the `exchangeNg` function.

MORE Vaults: Resolved with @9fa2e9f...

Zenith: Verified.

[H-7] `MorigamiUniswapV2LpTokenOracle::latestPrice()` result can be manipulated by an attacker

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MorigamiUniswapV2LpTokenOracle

Description:

The function MorigamiUniswapV2LpTokenOracle::latestPrice() uses reserves and assumes pools are always balanced when calculating the LP value:

```
price = quoteAsset == IUniswapV2Pair(baseAsset).token0()
    ? token0.mulDiv(balance, totalSupply, roundingMode) * 2
    : token1.mulDiv(balance, totalSupply, roundingMode) * 2;
```

Let's assume `quoteAsset` is `token0`. An attacker can imbalance the pool by swapping a huge amount of `token0` for `token1`. At this point the pool reserves will have a big amount of `token0` and a low amount of `token1`. MorigamiUniswapV2LpTokenOracle::latestPrice() will price the LP by doubling the value of `token0`, this will result in the LPs being overvalued as the pool is not balanced.

Recommendations:

Price UniswapV2 LPs using formulas that can't be manipulated. Here's a good resource on how to achieve that.

MORE Vaults: Resolved with @44fbe7

Zenith: Verified.

[H-8] `MorigamiCurveLpTokenOracle::latestPrice()` uses curve's `calc_withdraw_one_coin()` to price LPs, which can be manipulated

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MorigamiCurveLpTokenOracle::latestPrice()

Description:

The MorigamiCurveLpTokenOracle::latestPrice() oracle uses curve's calc_withdraw_one_coin() to price LP tokens.

calc_withdraw_one_coin() uses spot balances to price LPs and can be manipulated by an attacker.

Recommendations:

Use curve's get_virtual_price() to price LP tokens.

Keep in mind that get_virtual_price() can also be subject to manipulation via read-only reentrancy, in order to protect from this attack the protocol can call a function that triggers the reentrancy lock in the curve pool, such as remove_liquidity():

1. Call remove_liquidity() by passing `0` as `_amount`
2. Call get_virtual_price()

MORE Vaults: Resolved with @46d2666...

Zenith: Verified.

[H-9] The liquidity token is incorrect when depositing a native token into the Uniswap V2 pool

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- UniswapV2Facet.sol

Description:

In the `Uniswap V2 factory`, a `pool` cannot be created if either `token address` is `address(0)`.

- UniswapV2Factory.sol#L26

```
function createPair(address tokenA, address tokenB)
    external returns (address pair) {
    require(tokenA ≠ tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
    (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) :
    (tokenB, tokenA);
    require(token0 ≠ address(0), 'UniswapV2: ZERO_ADDRESS');
}
```

This introduces an issue when adding `liquidity` with `native tokens` using the `addLiquidityETH` function. In such cases, the `liquidityToken` returned is always `address(0)`, and this invalid address is mistakenly held instead of the correct `LP token`.

- UniswapV2Facet.sol#L136-L137

```
function addLiquidityETH(
    address router,
    address token,
    uint amountTokenDesired,
    uint amountETHDesired,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) external returns (uint amountToken, uint amountETH, uint liquidity) {
    address defaultUniswapFactory = IUniswapV2Router02(router).factory();
```

```
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, address(0));
    ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID].add(liquidityToken);
}
```

As a result, removing `liquidity` involving `native tokens` fails, because the transaction reverts due to the invalid `LP token`.

- UniswapV2Facet.sol#L454

```
function _removeLiquidityETH(
    address router,
    address token,
    uint liquidity,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) internal returns (uint amountToken, uint amountETH) {
    address defaultUniswapFactory = IUniswapV2Router02(router).factory();
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, address(0));

    IERC20(liquidityToken).approve(router, liquidity);  // liquidityToken =
    address(0)
}
```

Furthermore, if `address(0)` is held as the `LP token`, the `accountingUniswapV2Facet` function will always revert.

- UniswapV2Facet.sol#L54

```
function accountingUniswapV2Facet() public view returns (uint sum) {
    for (uint i = 0; i < tokensHeld.length(); ) {
        address lpToken = tokensHeld.at(i);
        if (ds.isAssetAvailable[lpToken]) {
            continue;
        }
        uint totalSupply = IERC20(lpToken).totalSupply(); // lpToken =
    address(0)
}
```

This causes all `deposits` and `withdrawals` to the `ERC4626 vault` to be paused.

Recommendations:

```
function addLiquidityETH(
    address router,
    address token,
    uint amountTokenDesired,
    uint amountETHDesired,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) external returns (uint amountToken, uint amountETH, uint liquidity) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    IERC20(token).approve(router, amountTokenDesired);

    address defaultUniswapFactory = IUniswapV2Router02(router).factory();

    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, address(0));
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, ds.wrappedNative);
    if (liquidityToken == address(0)) {
        liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .createPair(token, ds.wrappedNative);
    }

}

function removeLiquidityETHSupportingFeeOnTransferTokens(
    address router,
    address token,
    uint liquidity,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) external returns (uint amountETH) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    address defaultUniswapFactory = IUniswapV2Router02(router).factory();

    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, address(0));
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
```

```
        .getPair(token, ds.wrappedNative);

}

function _removeLiquidityETH(
    address router,
    address token,
    uint liquidity,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) internal returns (uint amountToken, uint amountETH) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    address defaultUniswapFactory = IUniswapV2Router02(router).factory();

    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, address(0));
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(token, ds.wrappedNative);
}
```

MORE Vaults: Resolved with @1c00e43...

Zenith: Verified.

4.3 Medium Risk

A total of 18 medium risk findings were identified.

[M-1] Vault owners can change a vault registry from permissioned to permissionless at any time

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AccessControlFacet.sol

Description:

Vaults can have two registries:

- A permissionless one that allows the owner to add any custom feature to a vault
- A permissioned one that allows the owner to add protocol-approved features to a vault

A vault with permissioned registry offers guarantees to the users regarding how the vault can be customized and it shouldn't be possible for an owner of a vault with a permissioned registry to add permissionless custom features.

This is not currently the case as the owner of a vault is allowed to change the vault registry via AccessControlFacet::setMoreVaultRegistry() from a permissioned one to a permissionless one, effectively breaking the guarantees provided by the permissioned registry.

Recommendations:

In AccessControlFacet::setMoreVaultRegistry() don't allow vault owners to change the registry from permissioned to permissionless.

MORE Vaults: Resolved with @d73e2f0...

Zenith: Verified.

[M-2] `VaultFacet::mint()` and `VaultFacet::redeem()` round in the wrong direction

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- VaultFacet.sol

Description:

The function VaultFacet::mint() converts the input amount of `shares` to the amount of `assets` the caller needs to pay by rounding down (ie. `Math.Rounding.Floor`). This favors the user instead of the protocol as the amount of assets the caller will pay is lower than expected.

The function VaultFacet::redeem() converts the input amount of `shares` to the amount of `assets` the caller will receive by rounding up (ie. `Math.Rounding.Ceil`). This favors the user instead of the protocol as the amount of assets the caller will receive is higher than expected.

Because of this users will pay less or receive more assets than intended, which can be problematic for valuable assets with a low amount of decimals.

Recommendations:

- In VaultFacet::mint() round up when converting shares to assets
- In VaultFacet::redeem() round down when converting shares to assets

MORE Vaults: Resolved with @6a2fa9d...

Zenith: Verified.

[M-3] `UniswapV2Facet` doesn't ensure `ETH` is an available asset when performing `ETH` operations

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- UniswapV2Facet

Description:

- addLiquidityETH()
- removeLiquidityETH()
- swapExactETHForTokens()
- swapTokensForExactETH()
- swapExactTokensForETH()
- swapETHForExactTokens()
- removeLiquidityETHSupportingFeeOnTransferTokens()
- swapExactETHForTokensSupportingFeeOnTransferTokens()
- swapExactTokensForETHSupportingFeeOnTransferTokens()

Don't ensure `ETH` is an available asset. This can have consequences on the vault total assets calculations. As an example swapping an available asset to a non-available asset will instantly decrease the total value of the pool which results in user depositing receiving more shares than they should and users withdrawing receiving less assets than they should.

Recommendations:

In the above functions ensure `ETH` is an available asset.

MORE Vaults: Resolved with @d895ba...

Zenith: Verified.

[M-4] Unbounded looping can lead to protocol DOS

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet::accountingAaveV3Facet()
- CurveFacet::accountingCurveFacet()
- CurveLiquidityGaugeV6Facet::accountingCurveLiquidityGaugeV6Facet()
- MORELeverageFacet::accountingMORELeverageFacet()
- MultiRewardsFacet::accountingMultiRewardsFacet()
- UniswapV2Facet::accountingUniswapV2Facet()

Description:

Unbounded looping is a scenario in which the protocol loops over too many elements and results in an out-of-gas error, reverting the call.

There are multiple instances of unbounded loops:

1. AaveV3Facet::accountingAaveV3Facet() loops over an unbounded amount of `mTokensHeld` and `debtTokensHeld`
2. CurveFacet::accountingCurveFacet() loops over an unbounded amount of curve LP tokens (ie. `ds.tokensHeld[CURVE_LP_TOKENS_ID]`)
3. CurveLiquidityGaugeV6Facet::accountingCurveLiquidityGaugeV6Facet() loops over an unbounded amount of curve gauges (ie. `ds.tokensHeld[CURVE_LIQUIDITY_GAUGES_V6_ID]`) and each gauge loops over an unbounded amount of reward tokens
4. MORELeverageFacet::accountingMORELeverageFacet() loops over an unbounded amount of morigami tokens (ie. `ds.tokensHeld[ORIGAMI_VAULT_TOKENS_ID]`)
5. MultiRewardsFacet::accountingMultiRewardsFacet() loops over an unbounded amount of multirewards staking contracts (ie. `ds.tokensHeld[MULTI_REWARDS_STAKINGS_ID]`) and each staking contract loops over an unbounded amount of reward tokens
6. UniswapV2Facet::accountingUniswapV2Facet() loops over an unbounded amount of UniswapV2 LP tokens (ie. `ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID]`)

Recommendations:

Limit the amount of possible loops (and internal loops) performed by VaultFacet::totalAssets(). This requires benchmarking each function gas consumption in order to know what the gas limits are.

MORE Vaults: Resolved with @1fe501... and @05fcda...

Zenith: Verified.

[M-5] `MoreVaultsLib::removeFunction()` doesn't remove facet address from `ds.facetsForAccounting` leading to a DOS

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- MoreVaultsLib

Description:

The function MoreVaultsLib::removeFunction() doesn't remove the facet address from `ds.facetsForAccounting` when removing a facet.

If a facet is removed from `ds.facetAddresses` but not from `ds.facetsForAccounting` the function VaultFacet::totalAssets(), which loops over `ds.facetsForAccounting`, will attempt to call the accounting function on the facet which doesn't exists anymore and as such will revert.

A revert in VaultFacet::totalAssets() will DOS all vault operations.

Recommendations:

In MoreVaultsLib::removeFunction() remove the facet address from `ds.facetsForAccounting` when a facet is being removed.

MORE Vaults: Resolved with @982329...

Zenith: Verified.

[M-6] `accountingMultiRewardsFacet()` doesn't work properly if a reward token that is not an available assets is added to a multirewards contract

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- MultiRewardsFacet

Description:

Rewards can be added to a MultiRewards instance at any point via MultiRewards::addReward() but the MultiRewardsFacet contract doesn't take it in consideration.

If a reward token that is not an available asset is added to MultiRewards:

1. The function accountingMultiRewardsFacet() will revert on the execution of `convertToUnderlying()` if the reward token doesn't have an oracle, leading to a DOS of vault operations
2. The function accountingMultiRewardsFacet() will account for the value of the reward tokens if it has an oracle (even if it's not an available asset)
3. The function stake() can't be called as it reverts
4. The function getReward() can't be called as it reverts

Recommendations:

1. In accountingMultiRewardsFacet() skip a reward token if it's not an available asset
2. In stake() revert if `staking` is not in `ds.stakingAddresses[MULTI_REWARDS_STAKINGS_ID]` and a reward token is not an available asset, otherwise ignore the reward token
3. In getReward() ignore the reward tokens that are not available assets

MORE Vaults: Resolved with @20a63e... and @45a3dc...

Zenith: Verified.

[M-7] `accountingCurveLiquidityGaugeV6Facet()` doesn't work properly if a reward token that is not an available assets is added to a gauge

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- CurveLiquidityGaugeV6Facet

Description:

The contract CurveLiquidityGaugeV6Facet doesn't consider that new reward tokens can be added to gauges at any point.

There are multiple issues:

1. If the reward token we are converting to underlying doesn't have an oracle the whole accountingCurveLiquidityGaugeV6Facet() function reverts, leading to a DOS of vault operations
2. If the reward token has an oracle but it's not an available asset the accountingCurveLiquidityGaugeV6Facet()

function will account for the non-available reward token as part of the vault total assets.

3. The function claimRewardsCurveGaugeV6 can't be executed if it encounters a reward token that is not an available asset, making it impossible to claim rewards
4. The function depositCurveGaugeV6() reverts if the gauge has a reward token that is not an available asset, making it impossible to deposit

Looping over the rewards tokens in the function depositCurveGaugeV6() to make sure they are available assets it's not enough as new reward tokens can be added to a gauge at any point.

Recommendations:

- In CurveLiquidityGaugeV6Facet::accountingCurveLiquidityGaugeV6Facet() only account for reward tokens that are available assets.

- In claimRewardsCurveGaugeV6 don't revert when a reward token is not an available asset but skip it instead
- In depositCurveGaugeV6() revert if the gauge has not been added to `ds.stakingAddresses[CURVE_LIQUIDITY_GAUGES_V6_ID]` yet and ignore the reward token otherwise

MORE Vaults: Resolved with @32da23...

Zenith: Verified.

[M-8] `accountingCurveLiquidityGaugeV6Facet()` doesn't account for `CRV` rewards

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- CurveLiquidityGaugeV6Facet

Description:

The function CurveLiquidityGaugeV6Facet::accountingCurveLiquidityGaugeV6Facet() doesn't account for `CRV` rewards coming from the `minterContract`, it only accounts for rewards coming from gauges.

They will be accounted for when mintCRV() as long as `CRV` is an available asset. Users that will withdraw from a vault before then will receive less tokens than they should.

Recommendations:

In CurveLiquidityGaugeV6Facet::accountingCurveLiquidityGaugeV6Facet() account for unclaimed `CRV` rewards.

MORE Vaults: Resolved with @b2d65d...

Zenith: Verified.

[M-9] `accountingCurveFacet()` and `accountingUniswapV2Facet()` can account for staked LP tokens even if the vault doesn't control them anymore

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- accountingCurveFacet()

Description:

The functions CurveFacet::accountingCurveFacet() and UniswapV2Facet::accountingUniswapV2Facet() accounts for staked LP tokens (either in a gauge or multirewards) by querying the `ds.staked[lpToken]` state variable:

```
function accountingCurveFacet() public view returns (uint sum) {
    ... SNIP ...
    uint256 lpTokenBalance = IERC20(lpToken).balanceOf(address(this)) +
        ds.staked[lpToken]; //<-- HERE
    sum += MoreVaultsLib.convertToUnderlying(
        ICurveViews(lpToken).coins(0),
        (
            ICurveViews(lpToken).calc_withdraw_one_coin(
                lpTokenBalance,
                int128(0)
            )
        )
    );
    ... SNIP ...
}
```

Consider the following scenario:

1. The vault curator stakes curve LPs into a gauge. The vault receives gauge tokens, which can be transferred, and increases the `ds.staked[lpToken]` variable by the amount of curve LPs staked.
2. The vault curator swaps or transfers the gauge LP tokens. Because the vault doesn't own gauge tokens anymore it can't redeem the curve LP tokens but the

`ds.staked[lpToken]` variable still accounts for them.

In case this happens the vault will account for the staked curve LP tokens when calculating the total assets, but the curve LP tokens don't belong to the vault anymore. This results in a situation where shares are overpriced, meaning depositors will receive less shares than they should and withdrawers will receive more assets than they should.

Recommendations:

Either:

- Query the gauges/multirewards contracts directly to know how many staked curve LP tokens the vault control
- Lower `ds.staked[lpToken]` in case the vault swaps gauge tokens

MORE Vaults: Resolved with @a57c10...

Zenith: Verified.

[M-10] Use the SafeERC20 library for handling token approvals

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- UniswapV2Facet.sol

Description:

The `SafeERC20` library was not used for handling `token approvals. For example`, in the `addLiquidity` function of the `UniswapV2Facet`, the desired token amounts are all approved to the `router`. UniswapV2Facet.sol#L96-L97

```
function addLiquidity(
    address router,
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    uint deadline
) external returns (uint amountA, uint amountB, uint liquidity) {
    IERC20(tokenA).approve(router, amountADesired);
    IERC20(tokenB).approve(router, amountBDesired);

    address defaultUniswapFactory = IUniswapV2Router02(router).factory();
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(tokenA, tokenB);
    ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID].add(liquidityToken);

    return
        IUniswapV2Router02(router).addLiquidity(
            tokenA,
            tokenB,
            amountADesired,
            amountBDesired,
            amountAMin,
            amountBMin,
```

```
            address(this),
            deadline
        );
}
```

However, the actual transferred amounts may be `less than the approved amounts`.

This can cause issues with certain tokens—like `USDT`—which `do not allow updating a non-zero allowance to another non-zero value`. In such cases, you must `first set the allowance to zero` before approving a new `non-zero amount`.

Recommendations:

Use `SafeERC20` library.

MORE Vaults: Resolved with @87295c8...

Zenith: Verified.

[M-11] `AaveV3Facet::flashloan()` can leave debt tokens whose underlying assets are not available assets

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AaveV3Facet.sol

Description:

The function AaveV3Facet::flashLoan() doesn't verify that `assets` are available assets. This is important in case AaveV3Facet::flashLoan() ends up leaving debt tokens in the vault.

If AaveV3Facet::flashLoan() adds debt tokens whose underlying assets are not available assets the function AaveV3Facet::accountingAaveV3Facet() will revert when calculating the value of the debt tokens leading to a DOS of vault operations.

Recommendations:

In AaveV3Facet::flashLoan() ensure the underlying asset of each debt token left in the vault is an available asset.

MORE Vaults: Resolved with @3eacef...

Zenith: Verified.

[M-12] `accountingAaveV3Facet()` doesn't account for unclaimed rewards

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet

Description:

The function AaveV3Facet::accountingAaveV3Facet() doesn't account for rewards accumulated by `rewardsController`. This results in AaveV3Facet::accountingAaveV3Facet() returning a lower underlying value than expected.

Recommendations:

In AaveV3Facet::accountingAaveV3Facet() account for the unclaimed rewards in `rewardsController`.

MORE Vaults: Resolved with @5f92fb...

Zenith: Verified.

[M-13] `accountingAaveV3Facet()` can revert for underflow leading to a DOS on vault operations

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet.sol

Description:

The function AaveV3Facet::accountingAaveV3Facet() calculates the value of debt tokens and aTokens in terms of underlying and then subtracts the value of the debt from the value of aTokens.

If the value of the debt is greater than the value of the underlying AaveV3Facet::accountingAaveV3Facet() will revert for underflow when performing the subtraction. This can happen in case of bad debt (ie. A position has not been liquidated in time).

If this happens the protocol vault operations will be DOSed as VaultFacet::totalAssets() internally calls AaveV3Facet::accountingAaveV3Facet() and VaultFacet::totalAssets() is used in all vault operations (`deposit()`, `mint()`, `withdraw()`, `redeem()`).

Recommendations:

The protocol should handle the scenario in which the variable `sum` in AaveV3Facet::accountingAaveV3Facet() is negative.

MORE Vaults: Resolved with @fc62c1...

Zenith: Verified.

[M-14] The AccessControlFacet should also be added alongside the DiamondCutFacet in the constructor of MoreVaultsDiamond

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MoreVaultsDiamond.sol

Description:

When a `MoreVaultsDiamond` is created, the `DiamondCutFacet` is initialized in the `constructor`.

- MoreVaultsDiamond.sol#L25-L30

```
constructor(
    address _diamondCutFacet,
    address _registry,
    address _wrappedNative,
    IDiamondCut.FacetCut[] memory _cuts
) payable {
    IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
    bytes4[] memory functionSelectors = new bytes4[](1);
    functionSelectors[0] = IDiamondCut.diamondCut.selector;
    cut[0] = IDiamondCut.FacetCut({
        facetAddress: _diamondCutFacet,
        action: IDiamondCut.FacetCutAction.Add,
        functionSelectors: functionSelectors,
        initData: ""
    });
@->    MoreVaultsLib.diamondCut(cut);
    if (_cuts.length > 0) {
        MoreVaultsLib.diamondCut(_cuts);
    }
}
```

This `facet` enables the addition of new `facets` to the `diamond` contract. However, adding new `facets` requires that `msg.sender` is the `owner`.

- DiamondCutFacet.sol#L35

```
function diamondCut(
    IDiamondCut.FacetCut[] calldata _diamondCut
) external override {
    AccessControlLib.validateOwner(msg.sender);
    MoreVaultsLib.diamondCut(_diamondCut);
}
```

If the `AccessControlFacet` is not set up in the `constructor`, there's no way to set the `ownership` — making it impossible to add any new `facet`.

Therefore, the constructor's `_cuts` parameter **must** include the `AccessControlFacet`. Since `MoreVaultsDiamonds` are deployed via `VaultsFactory`, and there's no guarantee that the `facets` input to the factory always includes `AccessControlFacet`, this requirement needs to be enforced explicitly.

- VaultsFactory.sol#L84-L89

```
function deployVault(
    IDiamondCut.FacetCut[] calldata facets
) external returns (address vault) {
    vault = address(
        new MoreVaultsDiamond(
            diamondCutFacet,
            address(registry),
            wrappedNative,
            cuts
        )
    );
}
```

Recommendations:

```
// MoreVaultsDiamond
constructor(
    address _diamondCutFacet,
    address _accessControlFacet,
    address _registry,
    address _wrappedNative,
    IDiamondCut.FacetCut[] memory _cuts
    IDiamondCut.FacetCut[] memory _cuts,
    bytes memory accessControlFacetInitData
```

```
) payable {
    IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
    IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](2);
    bytes4[] memory functionSelectors = new bytes4[](1);
    functionSelectors[0] = IDiamondCut.diamondCut.selector;
    cut[0] = IDiamondCut.FacetCut({
        facetAddress: _diamondCutFacet,
        action: IDiamondCut.FacetCutAction.Add,
        functionSelectors: functionSelectors,
        initData: ""
    });

     bytes4[] memory functionSelectors_1 = new bytes4[](1);
    functionSelectors_1[0] = IAccessControlFacet.setMoreVaultRegistry.
        selector;
    cut[1] = IDiamondCut.FacetCut({
        facetAddress: _accessControlFacet,
        action: IDiamondCut.FacetCutAction.Add,
        functionSelectors: functionSelectors_1,
        initData: accessControlFacetInitData
    });

    AccessControlLib.setMoreVaultsRegistry(_registry);

    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    ds.wrappedNative = _wrappedNative;

    MoreVaultsLib.diamondCut(cut);
    if (_cuts.length > 0) {
        MoreVaultsLib.diamondCut(_cuts);
    }
}

// VaultFactory
 address public accessControlFacet;

function initialize(
    address _registry,
    address _diamondCutFacet,
    address _accessControlFacet,
    address _wrappedNative
) external initializer {
```

```
    if (
        _registry == address(0) ||
        _diamondCutFacet == address(0) ||
        _accessControlFacet == address(0) ||
        _wrappedNative == address(0)
    ) revert ZeroAddress();
    _setDiamondCutFacet(_diamondCutFacet);
    _setAccessControlFacet(_accessControlFacet);
    wrappedNative = _wrappedNative;

    registry = IMoreVaultsRegistry(_registry);

    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
}

function setAccessControlFacet(
    address _accessControlFacet
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    _setAccessControlFacet(_accessControlFacet);
}

function _setAccessControlFacet(address _accessControlFacet) internal {
    if (_accessControlFacet == address(0)) revert ZeroAddress();
    accessControlFacet = _accessControlFacet;
}

function deployVault(
    IDiamondCut.FacetCut[] calldata facets
    IDiamondCut.FacetCut[] calldata facets,
    bytes memory accessControlFacetInitData
) external returns (address vault) {
    IDiamondCut.FacetCut[] memory cuts = new IDiamondCut.FacetCut[](
        facets.length
    );
    for (uint256 i = 0; i < facets.length; ) {
        cuts[i] = IDiamondCut.FacetCut({
            facetAddress: facets[i].facetAddress,
            action: facets[i].action,
            functionSelectors: facets[i].functionSelectors,
            initData: facets[i].initData
        });
        unchecked {
            ++i;
```

```
        }
    }
    // Deploy new MoreVaultsDiamond (vault)
    vault = address(
        new MoreVaultsDiamond(
            diamondCutFacet,
            accessControlFacet,
            address(registry),
            wrappedNative,
            cuts
            cuts,
            accessControlFacetInitData
        )
    );
    isFactoryVault[vault] = true;
    deployedVaults.push(vault);
    emit VaultDeployed(vault, address(registry), wrappedNative, facets);
}
```

MORE Vaults: Resolved with @3100e56...

Zenith: Verified.

[M-15] `MORELeverageFacet` is incompatible with `MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens`

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- MORELeverageFacet

Description:

The following functions can't be used with MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens as a manager:

- MORELeverageFacet::rebalanceUp()
- MORELeverageFacet::forceRebalanceUp()
- MORELeverageFacet::rebalanceDown()
- MORELeverageFacet::forceRebalanceDown()

This is because both RebalanceUpParams and RebalanceDownParams used by MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens expect a `uint256[] flashLoanAmounts` parameter as one of the inputs while the MORELeverageFacet contract passes a single `uint256 flashLoanAmount` parameter, resulting in a reverted call.

Recommendations:

In MORELeverageFacet update the aforementioned functions so they call the possible different types of managers with the correct interface.

MORE Vaults: Acknowledged, client is not looking to integrate with the `MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens` manager.

[M-16] `_rebalanceUpFlashLoanCallback()` uses single `params.repaySurplusThreshold` parameter for every token

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens

Description:

The function MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol#L460 uses the same `params.repaySurplusThreshold` parameter for every token to determine if it's necessary to repay residual debt.

The function should use an array of `params.repaySurplusThreshold`, one for each token, because different tokens have different amount of decimals and different market values.

Recommendations:

The function MMorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol#L46should take an array of `params.repaySurplusThreshold` as input.

To achieve this it's necessary to update the RebalanceUpParams struct and adjust the upstream calls that lead to calling MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens.sol#L460.

MORE Vaults: Resolved with @3040ad...

Zenith: Verified.

[M-17] When adding liquidity to a non-existent Uniswap V2 pool, `address(0)` may be mistakenly held as the LP token

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- UniswapV2Facet.sol

Description:

When adding `liquidity` to a non-existent `Uniswap V2 pool`, the `factory`'s `getPair` function returns `address(0)`, which will mistakenly be held as the `LP token`.

- UniswapV2Facet.sol#L100-L102

```
function addLiquidity(
    address router,
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    uint deadline
) external returns (uint amountA, uint amountB, uint liquidity) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();

    IERC20(tokenA).approve(router, amountADesired);
    IERC20(tokenB).approve(router, amountBDesired);

    address defaultUniswapFactory = IUniswapV2Router02(router).factory();
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(tokenA, tokenB);
    ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID].add(liquidityToken);

    return
        IUniswapV2Router02(router).addLiquidity(
            tokenA,
```

```
            tokenB,
            amountADesired,
            amountBDesired,
            amountAMin,
            amountBMin,
            address(this),
            deadline
        );
}
```

Since the `pool` doesn't exist yet, the `addLiquidity` function in the `Uniswap V2 router` will create the `pool` as part of the transaction. As a result, the transaction still succeeds.

If `address(0)` is held as the `LP token`, the `accountingUniswapV2Facet` function will always revert.

- UniswapV2Facet.sol#L54

```
function accountingUniswapV2Facet() public view returns (uint sum) {
    for (uint i = 0; i < tokensHeld.length(); ) {
        address lpToken = tokensHeld.at(i);
        if (ds.isAssetAvailable[lpToken]) {
            continue;
        }
        uint totalSupply = IERC20(lpToken).totalSupply(); // lpToken =
    address(0)
}
```

This causes all `deposits` and `withdrawals` to the `ERC4626 vault` to be paused.

Recommendations:

```
function addLiquidity(
    address router,
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    uint deadline
) external returns (uint amountA, uint amountB, uint liquidity) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
```

```
    IERC20(tokenA).approve(router, amountADesired);
    IERC20(tokenB).approve(router, amountBDesired);

    address defaultUniswapFactory = IUniswapV2Router02(router).factory();
    address liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
        .getPair(tokenA, tokenB);

    if (liquidityToken == address(0)) {
        liquidityToken = IUniswapV2Factory(defaultUniswapFactory)
            .createPair(tokenA, tokenB);
    }

    ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID].add(liquidityToken);
}
```

MORE Vaults: Resolved with @51ec4a1...

Zenith: Verified.

[M-18] Vaults allow to front-run and back-run transactions to either profit or avoid losses

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- VaultFacet.sol

Description:

The VaultFacet contract allows for instant deposits and instant withdrawals.

Because of this is possible for users to front-run and back-run transactions that will change the amount of `totalAssets()` in the vault to either avoid losses or earn profits.

Let's assume the `curator` of the vault wants to swap some assets in the vault, this will generally cost a fee and will result in the vault `totalAssets()` to diminish. To avoid incurring the loss an user can:

1. Monitor the mempool for the swap transaction
2. Withdraw his deposited funds in the vault
3. The swap transaction goes through and `totalAssets()` gets lowered
4. Deposit the funds again

This results in the user avoiding the losses.

As another example let's assume an oracle is about to increase the price of an asset that's in the vault, an user can:

1. Monitor the mempool for oracle updates
2. Deposit assets in the vault before the oracle update is processed by the blockchain
3. The oracle update is processed by the blockchain, which increases `totalAssets()`
4. Withdraw assets from the vault which results in a profit

This results in the user making a profit.

Recommendations:

Enforce a delay between deposits and withdrawals. Charging a fee on deposits and/or withdrawals is also an option.

MORE Vaults: Resolved with @28ebe2...

Zenith: Verified.

4.4 Low Risk

A total of 8 low risk findings were identified.

[L-1] Inflation attack is still possible in case of multiple deposits from users

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultFacet.sol

Description:

Vaults use a virtual offset returned by `_decimalsOffset()` when performing shares-to-assets conversions to prevent inflation attacks.

The VaultFacet doesn't overwrite the internal `_decimalsOffset()` function, which currently returns `0`.

The idea of an inflation attack is the following:

1. An empty vault exists
2. The attacker deposits `1 wei` of token in the vault in receives `1 share`
3. The attacker transfers `X` amount of tokens directly to the vault (not a deposit), this doesn't mint any share and it increases the value of a single share as the total amount of assets increased but the amount of shares stayed the same
4. An user deposits in the vault `Y` amount of tokens where `Y` is an amount such that the minted amount of shares will round down to `0`. If this happens the user will receive no shares while still depositing assets, which are effectively lost.

While `_decimalsOffset()` returning `0` still lowers the profitability of the attack as the attacker will lose 50% of his direct transfer it can still be profitable if multiple users deposit low amount of tokens.

Here's a POC that can be copy-pasted in `VaultFacet.t.sol` to illustrate the issue:

```
function test_inflationAttack() public {
    // Mock oracle call
    vm.mockCall(
        registry,
        abi.encodeWithSignature("oracle()"),
        abi.encode(aaveOracleProvider)
    );
    vm.mockCall(
        registry,
        abi.encodeWithSignature("getDenominationAsset()"),
        abi.encode(asset)
    );
    vm.mockCall(
        aaveOracleProvider,
        abi.encodeWithSignature("getSourceOfAsset(address)"),
        abi.encode(oracle)
    );
    vm.mockCall(
        oracle,
        abi.encodeWithSignature("latestRoundData()"),
        abi.encode(0, 1 ether, block.timestamp, block.timestamp, 0)
    );
    vm.mockCall(
        oracle,
        abi.encodeWithSignature("decimals()"),
        abi.encode(8)
    );
    vm.mockCall(
        registry,
        abi.encodeWithSignature("protocolFeeInfo(address)"),
        abi.encode(address(0), 0)
    );

    address attacker = address(2);
    uint256 initialAttackerBalance = 1000 ether;
    MockERC20(asset).mint(attacker, initialAttackerBalance);
    vm.prank(attacker);
    IERC20(asset).approve(facet, type(uint256).max);

    //1. Attacker first deposits 1 wei in order to mint 1 share to himself
    and then transfers 1ETH directly to the vault
    //1 share is now worth 5e17ETH
    vm.startPrank(attacker);
    uint256 shares = VaultFacet(facet).deposit(1, attacker);
    IERC20(asset).transfer(address(facet), 1 ether);
    console.log("Attacker #shares: ",
```

```
    VaultFacet(facet).balanceOf(attacker));
    vm.stopPrank();

    //2. Multiple users deposit an amount of ETH (in this case 5e17) that
    rounds down the amount
    //of shares received (in this case 0)
    vm.startPrank(user);
    VaultFacet(facet).deposit(5e17, user);
    VaultFacet(facet).deposit(5e17, user);
    VaultFacet(facet).deposit(5e17, user);
    console.log("User #shares: ", VaultFacet(facet).balanceOf(user));
    vm.stopPrank();

    //3. Attacker redeems his share for profit
    vm.prank(attacker);
    VaultFacet(facet).redeem(1, address(attacker), address(attacker));
    console.log("Attacker balance: ", IERC20(asset).balanceOf(attacker));
    console.log("Attacker profit: ", IERC20(asset).balanceOf(attacker)
    - initialAttackerBalance);
}
```

Recommendations:

In VaultFacet.sol override the `_decimalsOffset()` and return a higher number than 0. Returning `1` or `2` should be enough.

MORE Vaults: Resolved with @cfe1136...

Zenith: Verified.

[L-2] The VaultFacet currently does not support native token deposits

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultFacet.sol

Description:

By design, the `vault` is capable of holding `native tokens`. For instance, `native tokens` can be `deposited` into a `Uniswap V2 pool` via the `addLiquidityETH` function.

- UniswapV2Facet.sol#L120

```
function addLiquidityETH(
    address router,
    address token,
    uint amountTokenDesired,
    uint amountETHDesired,
    uint amountTokenMin,
    uint amountETHMin,
    uint deadline
) external returns (uint amountToken, uint amountETH, uint liquidity) {
    return
        IUniswapV2Router02(router).addLiquidityETH{value: amountETHDesired}(
            token,
            amountTokenDesired,
            amountTokenMin,
            amountETHMin,
            address(this),
            deadline
        );
}
```

However, the `VaultFacet` currently lacks functionality to support direct `native token deposits`.

- VaultFacet.sol#L325-L329

```
function deposit(
    address[] calldata tokens,
    uint256[] calldata assets,
    address receiver
) external whenNotPaused returns (uint256 shares) {

}
```

Recommendations:

If this functionality was unintentionally omitted, the `deposit` function should be updated to accept `native tokens`.

MORE Vaults: Resolved with @1f94a2b... and @935891a...

Zenith: Verified.

[L-3] `CurveFacet::_exchange()` doesn't take in consideration adding/removing liquidity with `swap_type` either `5` and `7`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- CurveFacet

Description:

The function CurveFacet::_exchange():

- validates `inputToken` in case the `swap_type` is not `6` (ie. removing liquidity)
- validates `outputToken` in case the `swap_type` is not `4` (ie. adding liquidity)
- adds `outputToken` to `ds.tokensHeld[CURVE_LP_TOKENS_ID]` in case the `swap_type` is `4` (ie. adding liquidity)
- removes `inputToken` from `ds.tokensHeld[CURVE_LP_TOKENS_ID]` in case the `swap_type` is `6` (ie. removing liquidity)

The issue is that in curve's exchange function there are other `swap_type` types that allow to add/remove liquidity: `5` to add liquidity and `7` to remove liquidity.

At the current state using `5` or `7` as `swap_type` would revert anyway unless both `inputToken` and `outputToken` are available assets.

Recommendations:

In CurveFacet::_exchange() either:

- revert when `swap_type` is `5` or `7`
- correctly validate input/output tokens and addition/removal from `ds.tokensHeld[CURVE_LP_TOKENS_ID]` when `swap_type` is `5` or `7`

MORE Vaults: Resolved with @1b1a6e...

Zenith: Verified.

[L-4] Hardcoded `3 hours` staleness check for all oracles

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MoreVaultsLib

Description:

The function MoreVaultsLib::verifyPrice() uses a hardcoded `3 hours` staleness check for all oracles.

Different oracles might have different update frequencies (heartbeats). Using a fixed time window leads to two issues:

1. For oracles with shorter heartbeats stale prices might be used by the protocol.
2. For oracles with longer heartbeats the MoreVaultsLib::verifyPrice() function can revert on normal conditions. This would lead to a temporary DOS of vault operations.

Recommendations:

Use time windows tailored to each oracle update frequency when checking for stale prices.

MORE Vaults: Resolved with @4d74d6...

Zenith: Verified.

[L-5] The validity of individual function selectors is not checked when adding a new facet to the vault

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MoreVaultsLib.sol

Description:

In the `Vault Registry`, the `facet address` and its `function selectors` are registered. BaseVaultsRegistry.sol#L24

```
/// @dev Mapping selector ⇒ facet address
mapping(bytes4 ⇒ address) public selectorToFacet;

/// @dev Mapping of facet address ⇒ all selectors
mapping(address ⇒ bytes4[]) public facetSelectors;

/// @dev List of all allowed facets
address[] public facetsList;
```

When adding a new `facet`, there is a check to ensure the `facet` is registered in the `Vault Registry`. MoreVaultsLib.sol#L321-L324

```
function diamondCut(IDiamondCut.FacetCut[] memory _diamondCut) internal {
    AccessControlLib.AccessControlStorage storage acs = AccessControlLib
        .accessControlStorage();
    IMoreVaultsRegistry registry = IMoreVaultsRegistry(
        acs.moreVaultsRegistry
    );

    for (uint256 facetIndex; facetIndex < _diamondCut.length; ) {
        IDiamondCut.FacetCutAction action = _diamondCut[facetIndex].action;
        address facetAddress = _diamondCut[facetIndex].facetAddress;

        // Validate facet and selectors for Add and Replace actions
        if (
```

```
            action == IDiamondCut.FacetCutAction.Add ||
            action == IDiamondCut.FacetCutAction.Replace
        ) {
            // Check if facet is allowed in registry
            bool isAllowed = registry.isFacetAllowed(facetAddress);
            if (!isAllowed) {
                revert FacetNotAllowed(facetAddress);
            }
        }
    }
}
```

However, there is no check for the validity of the individual `function selectors` being added.

Recommendations:

A check for `function selectors` can be added.

MORE Vaults: Resolved with @b259032...

Zenith: Verified.

[L-6] The mTokens should be validated for removability within the repayWithATokens function

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet.sol

Description:

When `withdrawing` supply tokens, the `mTokens` are checked to determine if they can be removed.

- AaveV3Facet.sol#L425

```
function _withdraw(
    address pool,
    address asset,
    uint256 amount,
    address to
) internal returns (uint256 withdrawnAmount) {
    MoreVaultsLib.MoreVaultsStorage storage ds = MoreVaultsLib
        .moreVaultsStorage();
    address mToken = IPool(pool).getReserveData(asset).aTokenAddress;

    IERC20(mToken).forceApprove(pool, amount);
    withdrawnAmount = IPool(pool).withdraw(asset, amount, to);

    MoreVaultsLib.removeTokenIfnecessary(ds.tokensHeld[MTOKENS_ID], mToken);
}
```

However, in the `repayWithATokens` function, this check is missing—even though the tokens are actually burned.

- AaveV3Facet.sol#L204

```
function repayWithATokens(
    address pool,
```

```
    address asset,
    uint256 amount,
    uint256 interestRateMode
) external returns (uint256 repaidAmount) {
    address debtToken;
    if (interestRateMode == 1)
        debtToken = IPool(pool)
            .getReserveData(asset)
            .stableDebtTokenAddress;
    else
        debtToken = IPool(pool)
            .getReserveData(asset)
            .variableDebtTokenAddress;

    MoreVaultsLib.removeTokenIfnecessary(
        ds.tokensHeld[MORE_DEBT_TOKENS_ID],
        debtToken
    );
}
```

Recommendations:

```
function repayWithATokens(
    address pool,
    address asset,
    uint256 amount,
    uint256 interestRateMode
) external returns (uint256 repaidAmount) {
    address mToken = IPool(pool).getReserveData(asset).aTokenAddress;

    address debtToken;
    if (interestRateMode == 1)
        debtToken = IPool(pool)
            .getReserveData(asset)
            .stableDebtTokenAddress;
    else
        debtToken = IPool(pool)
            .getReserveData(asset)
            .variableDebtTokenAddress;

    MoreVaultsLib.removeTokenIfnecessary(
        ds.tokensHeld[MORE_DEBT_TOKENS_ID],
        debtToken
    );
```

```
        MoreVaultsLib.removeTokenIfnecessary(ds.tokensHeld[MTOKENS_ID], mToken);
    }
```

MORE Vaults: Resolved with @689efe...

Zenith: Verified.

[L-7] `flashLoanCallback()` forces to flashloan all available tokens when rebalancing

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens

Description:

The MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens::flashLoanCallback() function ensures each token in the `tokens` array is a valid debt token by requiring the order and amount of tokens in `tokens[i]` to be equal to `_debtTokens[i]`:

```
for (uint256 i = 0; i < tokens.length; ) {
    if (address(tokens[i]) ≠ address(_debtTokens[i]))
        revert CommonEventsAndErrors.InvalidToken(address(tokens[i]));
    unchecked {
        ++i;
    }
}
```

This implies MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens::flashLoanCallback() will fail unless all the tokens in the `_debtTokens` array are borrowed, this limits flexibility on being able to borrow only a sub-set of tokens.

Recommendations:

In MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens::flashLoanCallback() revert only if at least one token in the `tokens` input parameter is not in the `_debtTokens` array.

MORE Vaults: Resolved with @cef32a...

Zenith: Verified.

[L-8] Morigami protocol attempts to set elements of uninitialized arrays

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MorigamiAaveV3BorrowAndLendMultiBorrowTokens.sol
- MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens

Description:

- The constructor of the MorigamiAaveV3BorrowAndLendMultiBorrowTokens contract attempts to access the elements of the `_aaveDebtTokens` array without initializing it with the correct amount of elements first. This results in the constructor always failing making it impossible to deploy the contract.
- The MorigamiLovTokenFlashAndBorrowManagerMultiBorrowTokens::setOracles() function attempts to access the elements of the `_debtTokenToReserveTokenOracles` array without initializing it with the correct amount of elements first. This makes it impossible to set oracles.

Recommendations:

Initialize the arrays with the correct number of elements before setting values.

As an example in the constructor initialize the `_aaveDebtTokens` array before accessing its indexes:

```
... SNIP ...
_aaveDebtTokens = new address[](borrowTokens_.length);
 for (uint256 i = 0; i < borrowTokens_.length; ) {
... SNIP ...
```

MORE Vaults: Resolved with @dfa9f7...

Zenith: Verified.

4.5 Informational

A total of 7 informational findings were identified.

[I-1] It's impossible to add selectors to an already existing facet

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MoreVaultsLib.sol

Description:

The function DiamondCutFacet::diamondCut() allows to add new facets and new selectors to the diamond proxy.

It internally calls MoreVaultsLib::diamondCut() which always tries to initialize the relative facet when a vault owner wants to add a new selector:

```
function diamondCut(IDiamondCut.FacetCut[] memory _diamondCut) internal {
    ... SNIP ...
    if (action == IDiamondCut.FacetCutAction.Add) {
        initializeAfterAddition(
            _diamondCut[facetIndex].facetAddress,
            _diamondCut[facetIndex].initData
        );
    }
}
```

This will fail for existing facets as they are already initialized.

Recommendations:

In MoreVaultsLib::diamondCut() don't re-initialize the facet if it's been initialized already.

MORE Vaults: Resolved with @8c9b3d3...

Zenith: Verified.

[I-2] `ConfigurationFacet::setFee()` allows to change fees at any point with immediate effect

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ConfigurationFacet.sol

Description:

The function ConfigurationFacet::setFee() allows to owner of a vault to change the fee at any point with immediate effect.

This doesn't give enough time to users to withdraw their funds in case they don't agree with the fee change.

Recommendations:

If the owner wants to increase the fee implement functionality where the owner declares what the new fee will be and have the new fee take effect after a set amount of time.

This gives users time to react to the fee change.

MORE Vaults: Resolved with @f8e3c78b97...

Zenith: Verified.

[I-3] Lack of 2-step ownership transfer

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- AccessControlFacet.sol

Description:

The function AccessControlFacet::transferOwner() allows to transfer ownership to a new `_newOwner` address.

In case the old owner passes a wrong `_newOwner` address there is no way to get the ownership back.

Recommendations:

Implement a 2-step ownership transfer:

1. The current owner calls `transferOwner` which sets a new `pendingOwner` state variable to the specified new owner
2. The new owner calls a new function `acceptOwnership` which transfer the ownership in case `msg.sender == pendingOwner` and resets `pendingOwner` to `address(0)`

By doing this the possibility of setting the wrong address as owner is nullified.

MORE Vaults: Resolved with @2f1cfe... and @0436bd...

Zenith: Verified.

[I-4] `accountingAaveV3Facet()` rounds debt value in the wrong direction

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet.sol

Description:

The value of debt tokens is calculated in the function: AaveV3Facet::accountingAaveV3Facet()

```
for (uint i = 0; i < debtTokensHeld.length(); ) {
    address debtToken = debtTokensHeld.at(i);
    uint balance = IERC20(debtToken).balanceOf(address(this));
    address underlyingOfDebtToken = IATokenExtended(debtToken)
        .UNDERLYING_ASSET_ADDRESS();

    sum -= MoreVaultsLib.convertToUnderlying(
        underlyingOfDebtToken,
        balance );
    unchecked {
        ++i;
    } }
```

The function MoreVaultsLib::convertToUnderlying() rounds down the returned value. However, we are doing operations with a debt so it should round-up to favor the vault.

Recommendations:

In AaveV3Facet::accountingAaveV3Facet() round-up the value returned by MoreVaultsLib::convertToUnderlying() when subtracting the debt from `sum`.

MORE Vaults: Resolved with @205c49...

Zenith: Verified.

[I-5] `accountingAaveV3Facet()` double-counts the value of an `aToken` that's an available asset

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- AaveV3Facet

Description:

The function AaveV3Facet::accountingAaveV3Facet() double-counts the value of an `aToken` if the `aToken` is also in the `ds.isAssetAvailable` mapping and `ds.availableAssets` array. This leads to shares being overvalued and users being able to withdraw more than they should.

This is only an issue if any `aToken` is also an available asset with a valid oracle, which won't be the case according to the client.

Recommendations:

Don't add the value of an `aToken` to `sum` in case the `aToken` is in the `ds.isAssetAvailable` mapping:

```
for (uint i = 0; i < mTokensHeld.length(); ) {
    address mToken = mTokensHeld.at(i);

    if (ds.isAssetAvailable[mToken]) {
        continue;
    }
... SNIP ...
}
```

MORE Vaults: Resolved with @4b3b6c...

Zenith: Verified.

[I-6] The `vetoActions()` function can be improved by allowing to veto multiple actions at once

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MulticallFacet

Description:

Vaults have a `guardian` role that can veto a curator actions via the MulticallFacet::vetoActions() function.

The MulticallFacet::vetoActions() function could be improved by allowing the guardian to veto multiple actions at once, this would lower potential gas costs in case the guardian has to veto a high amount of actions.

Recommendations:

Rewrite MulticallFacet::vetoActions() to take as input an array of `actionsNonce` and allow to cancel multiple actions at once.

MORE Vaults: Resolved with @6b6dd3...

Zenith: Verified.

[I-7] Vault curator has multiple ways to steal funds from a vault

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Files in description

Description:

1. In AaveV3Facet::supply() and AaveV3Facet::borrow()the `curator` can pass a malicious contract as `pool` in order to add a malicious `mToken` to the `ds.tokensHeld[MTOKENS_ID]` array. When the vault is calculating the total assets the malicious `mToken` can return an arbitrary number that would allow the `curator` to steal funds.
2. In AaveV3Facet::withdraw() the `curator` can pass an arbitrary `pool` contract in order to remove any
3. In AaveV3Facet::supply(), AaveV3Facet::borrow(), AaveV3Facet::repay() and AaveV3Facet::repayWithATokens()the `curator` can pass a malicious contract as `pool` parameter that will be approved to move assets from the vault. He can than transfer assets from the vault via the malicious contract.
4. In AaveV3Facet::swapBorrowRateMode() the `curator` can pass an arbitrary malicious `pool` in order swap the debt tokens of a particular `mToken`. By doing this he can increase the vault total assets and profit from it.
5. In AaveV3Facet::flashLoan() and AaveV3Facet::flashLoanSimple() the `receiverAddress` is not validated. A curator can set himself as receiver and repay the flashloan using assets already in the vault, effectively stealing them. He can also withdraw from the vault in the middle of a flashloan in which case the vault would include the loaned tokens in total assets.
6. In AaveV3Facet::flashLoan() the curator can pass a malicious contract as `pool` and add a custom malicious token to the `ds.tokensHeld[MORE_DEBT_TOKENS_ID]` array. When the vault is calculating the total assets the malicious `mToken` can return an arbitrary number that would allow the `curator` to steal funds.
7. In AggroKittySwapFacet::swapNoSplit() and AggroKittySwapFacet::swapNoSplitToNative() the curator can have the vault approve an arbitrary address `_router`. This would allow him to transfer funds out of the vault, stealing them.

8. The `curator` has the ability to set the slippage during swaps. Because of this he can set no slippage and sandwich the vault swap transaction in order to steal funds from the vault.
9. The `curator` has the ability to swap on arbitrary pools, because of this he create a pool where he owns the whole liquidity and swap assets back and forth in order to steal funds via fees.
10. In CurveLiquidityGaugeV6Facet::depositCurveGaugeV6() can pass an arbitrary contract as `gauge` and have the vault approve any token to it, allowing him to steal funds. He can also add an arbitrary malicious `gauge` to `ds.stakingAddresses[CURVE_LIQUIDITY_GAUGES_V6_ID]` and an arbitrary amount of assets to `ds.staked[address(lpToken)]` where `lpToken` is any asset of his choice.
11. In CurveLiquidityGaugeV6Facet::withdrawCurveGaugeV6() can lower the `ds.staked[address(lpToken)]` variable of any asset of his choice by passing a malicious `gauge` as input.
12. In IzumiSwapFacet::swapAmount() and IzumiSwapFacet::swapDesire() can pass an arbitrary `swapContract` as input to which the vault will approve `inputToken`. This allows the curator to steal funds.
13. In MORELeverageFacet::investWithToken() the`curator` can approve an arbitrary `lovToken` contract to transfer the vaults assets. He can also add any arbitrary `lonToken` contract to the `ds.tokensHeld[ORIGAMI_VAULT_TOKENS_ID]` array.
14. In MultiRewardsFacet::stake() the `curator` can approve tokens to an arbitrary malicious `staking` contract. He can also add any arbitrary malicious contract to `ds.stakingAddresses[MULTI_REWARDS_STAKINGS_ID]` and any arbitrary amount to `ds.staked[address(stakingToken)]` on a `stakingToken` of his choice.
15. In MultiRewardsFacet::withdraw() the `curator` can remove an arbitrary amount of a token of his choice from `ds.staked[address(stakingToken)]`.
16. The `curator` can have the diamond proxy call `deposit/mint/withdraw/redeem` VaultFacet in order to change the value of each share. As an example calling `deposit()` would dilute the value of shares as new shares are minted but no assets are added.
17. In CurveFacet::_exchange() the `curator` can approve any token to an arbitrary malicious `curveRouter`. He can also add any `outputToken` to the `ds.tokensHeld[CURVE_LP_TOKENS_ID]` array or remove any `inputToken` from the `ds.tokensHeld[CURVE_LP_TOKENS_ID` array.
18. In UniswapV2Facet::addLiquidity(), UniswapV2Facet::addLiquidityETH(), UniswapV2Facet::removeLiquidity, UniswapV2Facet::removeLiquidityETH the `curator` can approve any malicious `router` contract to spend the vault assets. He can also add/remove any arbitrary malicious `liquidityToken` contract to `ds.tokensHeld[UNISWAP_V2_LP_TOKENS_ID]`.
19. In the swapping functions in UniswapV2Facet the `curator` can approve any arbitrary malicious `router` to transfer the vault tokens.

20. In exactInputSingle(), exactInput(), exactOutputSingle(), exactOutput() the `curator` can approve any malicious `router` to transfer the vault tokens.
21. The `curator` can supply `assets` directly to the `Aave pool` outside of the `vault` and then transfer the resulting `aTokens` to the `vault`. This allows them to `borrow` more tokens than the `vault` has tracked as `supplied`. As a result, the `AccountingAaveV3Facet` may revert due to inconsistent accounting.

Recommendations:

Most of the issues derive from improper input validation. The protocol should have a list of allowed AAVEV3 pools, Curve pools, Uniswap pools, Curve Gauges, Routers, LOV tokens, and MultiRewards instances.

Not having such a list should be fine on permissionless vaults, as the `curator` has full power anyway and can steal funds by adding any custom facet, but options should be limited when it comes to permissioned vaults.

MORE Vaults: Resolved with @18f8ed..., @37e3a5... and @bf654b...

Zenith: Verified.