FILAMENT

SMART CONTRACTS REVIEW

zokyo

August 15th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING FILAMENT

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 9 Critical issues: 9 resolved = 0 points deducted
- 8 High issues: 7 resolved and 1 acknowledged = - 4 points deducted
- 20 Medium issues: 18 resolved and 2 acknowledged = - 2 points deducted
- 9 Low issues: 9 resolved = 0 points deducted
- 12 Informational issues: 11 resolved and 1 acknowledged = 0 points deducted

Thus, 100 - 4 - 2 = 94

TECHNICAL SUMMARY

This document outlines the overall security of the Filament smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Filament smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 9 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Filament team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Filament repository:
Repo: https://github.com/FilamentFinance/Filament-v1

Last commit - 1b305a547f6356f42f35ce615d4a69c5ba165e44

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./Deposit.sol
  ./facets/DiamondCutFacet.sol
  ./facets/OwnershipFacet.sol
  ./facets/TradeFacet.sol
  ./facets/ViewFacet.sol
  ./facets/DiamondLoupeFacet.sol
  ./facets/VaultFacet.sol
  ./upgradeInitializers/DiamondInit.sol
  ./Governor.sol
  ./libraries/LibDiamond.sol
  ./Formula.sol
  ./Escrow.sol
  ./LpToken.sol
  ./MMTrade.sol
  ./Diamond.sol
  ./Router.sol
  ./Referrals.sol
  ./Keeper.sol
  ./AppStorage.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Filament smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Thorough manual review of the codebase line by line.

Executive Summary

Filament introduces an innovative architecture in the perpetual decentralized exchange (Perp DEX) domain by integrating an off-chain order book with on-chain liquidity pools. This hybrid system is designed to address key challenges in the Perp DEX market, such as optimizing liquidity in low-volume markets, enhancing capital efficiency for liquidity providers, and expanding the range of tradable assets.

The Filament protocol is organized using the Diamond upgradeability pattern, dividing its functionalities into facets. Key facets include:

- OwnershipFacet: Manages permissioned accounts within the protocol.
- ViewFacet: Retrieves critical protocol variables.
- TradeFacet: Handles position creation and reduction, calculating profits/losses, and managing collateral.
- VaultFacet: Oversees protocol state, including rebalancing, ADL, global PNL, and LP balances.

Additional contracts, like Deposit.sol, manage user deposits, withdrawals, and liquidity transfers. The Keeper contract assists with fee distribution, liquidation, and treasury management. The protocol also features a referral system, allowing users to earn rewards by inviting others to the platform.

The Filament Protocol's architecture addresses key challenges in the perpetual DEX market, aiming to optimize liquidity in low-volume markets, enhance capital efficiency for liquidity providers, and broaden the range of tradable assets. The use of the Diamond standard and traditional proxies enhances interactions with the contracts while maintaining a clear separation of functionalities.

Zokyo conducted a security review of the Filament protocol, identifying various issues from critical findings to architectural suggestions. While the code is well-structured, inadequate testing led to several business logic errors. The Filament Development Team has been diligent in addressing these issues. Further recommendations include implementing more rigorous unit testing and conducting fuzz testing on core components to ensure the highest level of security before deployment.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Filament team and the Filament team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	FundingFee is incorrectly accrued, leading it to always be 0	Critical	Resolved
2	Position average price is wrongly calculated leading to users paying more/less for their position than the actual amount required	Critical	Resolved
3	User's funds are locked and lost when decreasing a position	Critical	Resolved
4	`getGlobalPnL` is wrongly accrued resulting on a critical computation	Critical	Resolved
5	Fees are wrongly accrued when reducing collateral	Critical	Resolved
6	Liquidation is not validated at the right time	Critical	Resolved
7	Referrer is not valid for earning for more time than expected	Critical	Resolved
8	Collateral is reduced twice when a position is liquidated by a user different from `protocolLiquidator`	Critical	Resolved
9	Incorrect movement of funds in TradeFacet from Deposit contract may cause permanent locking of funds	Critical	Resolved
10	Users are paying an incorrect amount of fees when decreasing a position leading to loss of funds	High	Resolved
11	Interest Could Be Calculated As 0 Due To Rounding	High	Resolved
12	addNewAsset Does Not Mark The Previous Assets As False	High	Resolved
13	When updateCompartmentsReduceLiq Is Called It Is Possible The Compartment Becomes Invalid	High	Acknowledged
14	Position Might Still Be Liquiditable Even When User Increases Position	High	Resolved

#	Title	Risk	Status
15	updateBalances Should Be Called Before collectFees()	High	Resolved
16	Anyone can set a malicious `diamond` address	High	Resolved
17	The minimum time for earning as a referrer can be bypassed resulting in user claiming earnings before the minimum time	High	Resolved
18	Fees keep accumulating when the protocol is paused	Medium	Acknowledged
19	A malicious user can add themselves as a referrer in the Referral contract to aid phishing attacks	Medium	Resolved
20	>= Instead Of >	Medium	Resolved
21	Multiple Functionalities Will Break If USDC Turns On Fee-On-Transfer	Medium	Resolved
22	Improper use of the whenNotPaused modifier for liquidations may result in the protocol taking on bad debt or resulting in users being liquidated as soon as unpause is executed	Medium	Resolved
23	onlySequencer modifier in processTransaction can be bypassed in the Router contract by processing since batched transactions	Medium	Resolved
24	VaultFacet contains a rug vector where a rogue team member may be able to cause mass liquidations	Medium	Resolved
25	Positions can not get liquidated when protocol is paused	Medium	Resolved
26	Back-running attacks can liquidate transactions without any option to repay the debt after changing the collateralization ratio	Medium	Resolved
27	Missing access restrictions modifier	Medium	Resolved
28	Transaction displacement can lead to the referrer not being able to earn	Medium	Acknowledged

#	Title	Risk	Status
29	It is not possible to add only 1 asset to the vault	Medium	Resolved
30	Vault and diamond assigned to the same address	Medium	Resolved
31	`maxReferralEarningTime` should be within a defined threshold to avoid excessive waiting for earning	Medium	Resolved
32	`collateralizationRatio` can be set higher than 100% which would result in no positions getting liquidated	Medium	Resolved
33	DOS for withdrawing funds	Medium	Resolved
34	`allowUSDCtoVault` increases allowance by X instead of setting it to X	Medium	Resolved
35	`reservedSizeShort` can not reach its maximum allowed value	Medium	Resolved
36	Loss of decimal precision for `currentFundingTime`	Medium	Resolved
37	`updateBalancesForAll()` and `updateBalances()` functions are callable when the contract is paused.	Medium	Resolved
38	Transaction displacement lead to force users to stake more time than expected	Low	Resolved
39	Optimal utilization percentage can be set to more than 100%	Low	Resolved
40	`init` function can be called with malicious values by attacker	Low	Resolved
41	Missing Referrer Existence Check in generateMyCode	Low	Resolved
42	Non-zero Amount Validation in addEarnings	Low	Resolved
43	Position leverage is hardcoded when transferring position as a result of a liquidation	Low	Resolved
44	Unlimited Registrations with Same Referral Code	Low	Resolved

#	Title	Risk	Status
45	Lack of Self-referral Prevention	Low	Resolved
46	Duplicated function	Low	Resolved
47	minUSDC Can Be Bypassed While Adding Liquidity	Informational	Acknowledged
48	Incorrect NatSpec comment about decimals	Informational	Resolved
49	updateBalances Can Return Early	Informational	Resolved
50	_transferIn Function Can Be Removed	Informational	Resolved
51	_validateOrder Should Have A Sanity Check For Collateral Too	Informational	Resolved
52	Contract not pausable	Informational	Resolved
53	Missing events	Informational	Resolved
54	Unnecessary if keyword	Informational	Resolved
55	Reading the array length from storage costs more gas	Informational	Resolved
56	Unused variables	Informational	Resolved
57	Unnecessary assignment	Informational	Resolved
58	Not only the keeper can call `addEarnings` but anybody	Informational	Resolved

CRITICAL-1 | RESOLVED

FundingFee is incorrectly accrued, leading it to always be 0.

The `netFundingAmount`, which represents the funding fee amount for a trader based on the size and funding rates, is calculated and returned by the `getFundingFee()` function withing the `Formula.sol` smart contract. This function will always return that the fundingFee is 0 because it is incorrectly calculated.

```
function getFundingFee(address _asset, uint256 _size, FundingRate memory _entryFundingRate,
bool _islong)
    external
    view
    returns (uint256 netFundingAmount, bool tobePaid)
{
    // @note-rajeeb: fundingRate decimals 10^4
    FundingRate memory currentFundingRate = cumulativeFundingRates[_asset];
    FundingRate memory netFundingRate = subtractFundingRates(_entryFundingRate,
currentFundingRate);

     netFundingAmount *= (_size * netFundingRate.value) / 1e6;
    // fundingRate decimals 4
    // @note-rajeeb: size (100) = 100_000_000, fundingRate (0.1%) = 1000
    // funding amount = (100_000_000 * 1000)/1_000_000 = 100_000 (i.e. = 0.1)
    if (
        (_islong && netFundingRate._direction == Direction.LongToShort)
            || (!_islong && netFundingRate._direction == Direction.ShortToLong)
    ) {
        tobePaid = true;
    } else {
        tobePaid = false;
    }
```

As it can be seen the `netFundingAmount` is calculated as:

- `netFundingAmount *= (_size * netFundingRate.value) / 1e6;`
- `netFundingAmount` is initialized above to 0.
- As a result it will always be a multiplication by 0 resulting in `netFundingAmount = 0`.

The described scenario leads to `netFundingAmount` being always 0 and as a result users avoid paying funding fees.

Recommendation:

Fix how `netFundingAmount` is calculated:

```
netFundingAmount = (_size * netFundingRate.value) / 1e6;
```

CRITICAL-2 | RESOLVED

Position average price is wrongly calculated leading to users paying more/less for their position than the actual amount required.

When a user opens a position calling `increasePosition()` within the `TradeFacet.sol` smart contract the position's average price is calculated by `getNextAveragePrice()`. This function calculates the average price considering `reservedSizeLong` within others.

The value for `reservedSizeLong` is updated by `updateReserve()` every time a new position is created:

```
function updateReserve(IncreaseParamaters memory _increasePositionParams, uint256 price)
internal {
        if (_increasePositionParams.isLong) {
            if (s.reservedSizeLong[_increasePositionParams.indexToken] == 0) {
                s.averagePriceLong[_increasePositionParams.indexToken] = price;
                s.reservedSizeLong[_increasePositionParams.indexToken] =
(_increasePositionParams.indexDelta) / price;
```

However, the calculation is wrongly accrued:

` s.reservedSizeLong[_increasePositionParams.indexToken] = (_increasePositionParams.indexDelta) / price;`

A `*1e6` factor is missed. This leads to a wrong calculation of reservedSizeLong.

The same error takes place later in the same function when passing parameters to `_increaseGlobalShortSize` leading to a wrong calculation of `reservedSizeShort`:

```
} else {
                updateAveragePriceShort(_increasePositionParams.indexToken, price,
_increasePositionParams.indexDelta);
                _increaseGlobalShortSize(
                    _increasePositionParams.indexToken, (_increasePositionParams.indexDelta)
/ (price)
                );
            }
```

Recommendation:

Fix the calculation:

`s.reservedSizeLong[_increasePositionParams.indexToken] = (_increasePositionParams.indexDelta) * 1e6 / price;`

And

`_increaseGlobalShortSize( _increasePositionParams.indexToken, (_increasePositionParams.indexDelta) * 1e6 / (price));`

CRITICAL-3 | RESOLVED

User's funds are locked and lost when decreasing a position

When a user decreases a position calling `decreasePosition()` within the `TradeFacet.sol` smart contract, a call to `lockForAnOrder` is executed. `lockForAnOrder` is locking user's fundings which can later not get unlocked:

```
/// @notice Lock an amount of USDC for an order
    /// @param _account The address of the user whose funds will be locked
    /// @param _amount The amount of USDC to lock
    /// @dev Can only be called by the diamond contract
    function lockForAnOrder(address _account, uint256 _amount) external onlyDiamond
whenNotPaused {
        if (_amount > balances[_account]) {
            revert LockForOrderFailed(_account);
        }
        locked[_account] += _amount;
        balances[_account] = balances[_account] - _amount;
    }
```

Locking funds is only needed when increasing positions, not when decreasing them.

Recommendation:

Replace lockForAnOrder with unlockForAnOrder.

`getGlobalPnL` is wrongly accrued resulting on a critical computation.

The function `getGlobalPnL()` within the `VaultFacet.sol` smart contract is used to calculate the profit and losses of the whole protocol:

```
function getGlobalPnL(address _address, uint256 price) public view returns (int256) {
       int256 ProfitLong =
           ((int256(price) - int256(s.averagePriceLong[_address])) *
(int256(s.reservedSizeLong[_address]))) / 1e6;
       int256 ProfitShort =
           (int256(price) - int256(s.averagePriceShort[_address])) *
(int256(s.reservedSizeShort[_address])) / 1e6;
       return ProfitLong + ProfitShort;
   }
```

However, Profit short is wrongly accrued as the subtraction should be the other way around.

Recommendation:

Fix the calculation:
int256 ProfitShort =
(int256(s.averagePriceShort[_address])) * (int256(s.reservedSizeShort[_address]) - int256(price)) / 1e6;

Fees are wrongly accrued when reducing collateral.

The function `_reduceCollateral()` within the `TradeFacet.sol` smart contract is called when a user decreases a position, this function gets the `fundingFee` and if it needs to be paid:

```
(uint256 fundingfee, bool tobePaid) = IFormula(s.formula).getFundingFee(
            position._indexToken, position.size, position.entryFundingRate, position.isLong
        );

        if (tobePaid) {
            fee = int256(borrowingInterest + fundingfee);
        } else {
            if (borrowingInterest > fundingfee) {
                fee = int256(borrowingInterest - fundingfee);
            } else {
                fee = toNegativeInt256(fundingfee + borrowingInterest);
            }
        }
```

The function differentiates several cases, however, the case when the fee does not have to be paid and the fundingFee is greater or equal to the borrowingInterest is wrongly implemented.

Recommendation:

Fix the last case with the correct implementation:
`fee = toNegativeInt256(fundingfee - borrowingInterest);`

Liquidation is not validated at the right time.

When a user calls `increasePosition()` within the TradeFacet.sol it is validated if the position is liquidable or not before the updates of the position have been applied.

Consider the following scenario:,
1. Position is healthy.
2. Checks that positions is healthy.
3. Updates positions with an unhealthy result.
4. No checks to ensure that the new position is not healthy.

The new position will be directly updated to an unhealthy state.

The same case happens when `decreasePosition()` a user could decrease its positions directly to an unhealthy state.

Recommendation:

Implement a liquidation validation check after the position has been updated and do not allow updating positions to an unhealthy state.

Referrer is not valid for earning for more time than expected.

The `isValidForEarning()` function in `Referrals.sol` is used for checking if a referrer is able to earn funds or not. This restriction is checked considering that a referrer is available for earning only if `maxDaysForEarning` have passed. The issue is that the check is wrongly implemented:

```
if (regis.hasRegistered && regis.registrationTime + (maxDaysForEarning * 1 days * 86_400) >=
block.timestamp) {
            return true;
        }
```

maxDaysForEarning * 24 hours * 24 hours are multiplied instead of maxDaysForEarning * 24 hours.

Recommendation:

Remove `1 days` from the multiplication.

Collateral is reduced twice when a position is liquidated by a user different from `protocolLiquidator`.

The function `liquidatePosition()` within the `Escrow.sol` smart contract is used to liquidate a position. This function differentiates if the caller is `protocolLiquidator` or any other user. However, there is an `else` statement missing:

```
if (msg.sender != protocolLiquidator) {
            require(actualCollateralRequired <= _collateralAmount, "less collateral");
            uint256 positionSize = position.size;
            require(positionSize >= _collateralAmount, "position size should be greater than
collateral");
            uint256 userleverage = positionSize / _collateralAmount;
            require(userleverage <= 20, "exceeds 20X leverage");
            position.collateral = positionSize / userleverage;

            IERC20(usdc).transferFrom(msg.sender, address(this), _collateralAmount);
            IERC20(usdc).approve(diamond, _collateralAmount);

            ITrade(diamond).updateCollateralFromLiquidation(
                _decreasedCollateralValue, _collateralAmount, position, newKey
            );
        }

       ITrade(diamond).updateCollateralFromLiquidation(_decreasedCollateralValue, 0,
position, newKey);
```

It can be observed that is caller is not `protocolLiquidator` `updateCollateralFromLiquidation() is executed twice:

1st: ` ITrade(diamond).updateCollateralFromLiquidation(
            _decreasedCollateralValue, _collateralAmount, position, newKey``
2nd: `ITrade(diamond).updateCollateralFromLiquidation(_decreasedCollateralValue, 0, position, newKey);`

This means that `_decreasedCollateralValue` is decreased twice from the global reserves:

```
function updateCollateralFromLiquidation(
        uint256 _decreaseCollateralValue,
        uint256 _collateralAmount,
        Position memory _position,
        bytes32 _key
    ) external onlyEscrow {
        if (_position.isLong) {
            s.longCollateral[_position._indexToken] -= _decreaseCollateralValue;
            s.longCollateral[_position._indexToken] += _collateralAmount;
        } else {
            s.shortCollateral[_position._indexToken] -= _decreaseCollateralValue;
            s.shortCollateral[_position._indexToken] += _collateralAmount;
        }
        s.positions[_key] = _position;
    }
```

Reducing two times the collateral value leads to a wrong accounting of the global reserves for the long and short positions. This has a critical impact on other functions of the protocol when users are decreasing positions and the reserves are checked.

Recommendation:

Add an ´else´ statement to only reduce collateral once.

CRITICAL-9 | RESOLVED

Incorrect movement of funds in TradeFacet from Deposit contract may cause permanent locking of funds

The TradeFacet contract facilitates the use of the decrease and increase position functions which handles these operations, adjusts collateral, the size and reserved amounts. In addition to this it also handles the assessing of liquidation conditions. When the user attempts to increase their position, the Deposit contract is called to lock funds for an order through the `lockForAnOrder` function which modifies the users balances to remove free funds and saves them to locked funds however, the decrease position operation is also locking for an order which may cause a denial of service condition and preventing users funds from being released from a locked state as we are locking funds that the user does not have.

Recommendation:

It's recommended that user funds are released in the Deposit contract when decreasing a position via the `unlockForAnOrder` function.

Users are paying an incorrect amount of fees when decreasing a position leading to loss of funds.

When a user decreases one of its positions by calling `decreasePosition()` within the `TradeFacet.sol` smart contract, `tradeFees` are calculated and decreased from position.collateral, these feeds vary depending if the traderType is Maker or Taker.

```
if (_decPosParams.traderType == TraderType.Maker) {
            (, uint256 makerincreaseFees,,,) =
s.keeper.orderFeesByCompartment(position._indexToken);
            uint256 tradeFees = (makerincreaseFees * _decPosParams.indexDelta) / 10_000;
            s.keeper.distributeTradeFees(
                tradeFees, position._indexToken, _decPosParams.account,
_decPosParams.traderType
            );
        } else if (_decPosParams.traderType == TraderType.Taker) {
            (,,, uint256 takerIncreaseFees,) =
s.keeper.orderFeesByCompartment(position._indexToken);
            uint256 tradeFees = (takerIncreaseFees * _decPosParams.indexDelta) / 10_000;
            IERC20(s.USDC).approve(address(s.keeper), tradeFees);
            s.keeper.distributeTradeFees(
                tradeFees, position._indexToken, _decPosParams.account,
_decPosParams.traderType
            );
            position.collateral -= tradeFees;
        }
```

However, the fees used are `makerincreaseFees` and `takerincreaseFeeds` instead of decreasing ones.

Recommendation:

Fix the fees used when decreasing the position, use the decreasing ones.

Interest Could Be Calculated As 0 Due To Rounding

In the updateBalances function the formula to calculate interest (short or long) is as follows →

```
uint256 shortInterest = (borrowingRate * interval *
s.borrowedAmountFromPool[_indexToken].short)
            / (INTERESTRATE_DECIMALS * 365 * 86_400);
```

Consider the following scenario ->

Borrow rate for the index token = 3e3

Borrow amount from pool = 1e10 (10k USD)

Since the contracts would be deployed on SEI , the block time would be < 1 , let's assume interval to be 1 , and consider attacker is calling this via a bot every second (cheap gas on L2)

Therefore the equation becomes →

Interest = (3e3)(1)(1e10) / (1e6)(31536000) = 30000 / 31536

Which rounds down to 0 , therefore the interest that would be applied is 0 on the index token.

Recommendation:
Have a minimum interval or increase precision of the formula

addNewAsset Does Not Mark The Previous Assets As False

When adding new assets in the vault facet all the index tokens are removed first (the old ones) and new ones are added (deletion of allIndexTokens) and then new tokens are pushed in the allIndexTokens.

```
delete s.allIndexTokens;
        uint256 length = _address.length;
        for (uint256 i = 0; i < length; ++i) {
            s.allIndexTokens.push(_address[i]);
            s.IndexToken[_address[i]] = true;
            assignCompartment(_address[i], _percentage[i]);
            emit NewAssetAdded(_address[i], _percentage[i]);
        }
```

But , s.indexTokens[] is not being marked as false for the old tokens , i.e. all the old index tokens will still point to true even when they are removed from the allIndexTokens array , functions like increasePosition require the asset to be in the indexToken mapping but in this case a user might open a new position with an outdated index asset.

Recommendation:

Assign the s.indexToken[address] = false for old assets.

When updateCompartmentsReduceLiq Is Called It Is Possible The Compartment Becomes Invalid

When liquidity is being removed (removeLiquidity() in trade facet) , every compartment's balance is adjusted depending on the assigned percentage in the updateCompartmentsReduceLiq function . When liquidity is removed from a compartment it is possible that the balance of the compartment goes below the valid threshold and therefore should be marked as invalid since the available balance is less than the required balance.

```
for (uint256 i = 0; i < length; ++i) {
            address asset = s.allIndexTokens[i];
            Compartment memory compartment = s.compartments[asset];
            uint256 liquidityPerAsset = (amount *
(compartment.assignedPercentage)) / (BASIS_POINTS_DIVISOR);
            compartment.balance -= liquidityPerAsset;
```

The same can be argued for add Liquidity function , compartments which were invalid could have now become valid .

Recommendation:

From the vault facet compartmentalize() should be called so that the compartments are adjusted accordingly.

Position Might Still Be Liquiditable Even When User Increases Position

The user might realise that his position is subject to liquidation and decide to increase the collateral in the position , he calculates how much collateral is needed to make the position healthy and submits the increased Position request which is executed by the router/ sequencer .
Inside increasePosition() total fee is calculated which comprises of → borrow rate and taker maker fee which is then deducted from the position's collateral. Therefore , it is possible that even though the provided collateral was enough to make the position healthy but due to fee deductions the position still remains unhealthy and gets liquidated.

Recommendation:

Check if the position is liquiditable or not at the end of the increase position function.

updateBalances Should Be Called Before collectFees()

When increasePosition is invoked in the trade facet , the borrow interest is accumulated using the collectFees() function

```
uint256 fee = collectFees(borrow);
```

The collectFees() function calculates the interest which is dependent on the interestFactor which in turn is dependent on the cumulativeInterestRate .

```
function collectFees(Position memory _position) internal returns (uint256)
{
        uint256 interestFactor =
s.cumulativeInterestRate[_position._indexToken] -
_position.entryCumulativeInterestRate;
        uint256 interest = (interestFactor * _position.reserveAmount) /
(INTERESTRATE_DECIMALS * 365 * 86_400);
        emit BalanceOutDone(interest);
        return interest;
    }
```

The problem here is that the value of the cumulativeInterestRate might be outdated for which updateBalances(pos.indexToken) should be called before the collectFees() so that the correct interest gets accumulated.

Recommendation:

updateBalances(pos.indexToken) should be called before the collectFees()

Anyone can set a malicious `diamond` address.

The `Referrals.sol` smart contract implements an `addDiamond` function which set a specific address to the `diamond` which is used by the `onlyDiamond` modifier. The function `addDiamond` does not implement any access restrictions so anyone can set a new address for the diamond.

```
function addDiamond(address _address) external {
        diamond = _address;
}
```

The `onlyDiamond` modifier should be used for calling critical functions like `generateMyCode()` and `registerReferral()`. Due to the above description anybody would be able to call them without access restriction.

Scenario:

1. Any external user calls `generateMyCode` to generate a referral code without any restriction.
2. Unauthorized users can call registerReferral to register themselves using any referral code.
3. This can lead to unauthorized and fraudulent registrations and referral code generation, potentially manipulating referral rewards.

Recommendation:

Implement an onlyOwner modifier that checks if the caller is the owner, if not, revert the transaction.

HIGH-8 | RESOLVED

The minimum time for earning as a referrer can be bypassed resulting in user claiming earnings before the minimum time.

For a referrer to be valid to earn, a minimum of X days since referring should have passed. This is checked by the `isValidForEarning()` function in the `Referrals.sol` smart contract. The issue is that the function `addEarnings` which is used to set the earnings for a referrer does not check if a referrer is valid to get these earnings.

Recommendation:

Call the `isValidForEarnings` function inside the `addEarnings` function in order to assert that the referrer is a valid earner.

MEDIUM-1 | ACKNOWLEDGED

Fees keep accumulating when the protocol is paused.

The function `updateBalances()` within the `TradeFacet.sol` smart contract is used to update balances and distribute borrowing fees. This function is public and has no access restrictions or any other modifier so that it can be called at any point in time even when the rest of the protocol is paused.

The main logic of this function is working around the elapsed time since last execution. For calculating this elapsed time the `lastBalanceUpdateTime` is updated every time this function is executed so that an 'interval' is calculated.

```
if (s.lastBalanceUpdateTime[_indexToken] == 0) {
            s.lastBalanceUpdateTime[_indexToken] = block.timestamp;
        }
```

```
uint256 interval = (block.timestamp - s.lastBalanceUpdateTime[_indexToken]);
```

However, this elapsed time does not consider the pause of the protocol.
Consider the following scenario:

1. The protocol is unpaused, users create positions.
2. The protocol is paused for X blocks.
3. Users can not modify their positions but fees are being accumulated by these X blocks.

The interest calculated is used for decreasing long and short collateral which leads to users positions being decreased while the protocol gets fees:

```
s.longCollateral[_indexToken] -= longInterest;
        s.shortCollateral[_indexToken] -= shortInterest;
        uint256 totalInterest = shortInterest + longInterest;

        IERC20(s.USDC).approve(address(s.keeper), totalInterest);
        s.keeper.distributeBorrowingFees(address(this), totalInterest, _indexToken);
```

Recommendation:

Implement a mechanism to not consider the blocks while the protocol were paused to be used for generating fees.

Client comment: Borrowing fee should still be calculated if protocol is paused because funds are still borrowed.

MEDIUM-2 | RESOLVED

A malicious user can add themselves as a referrer in the Referral contract to aid phishing attacks

The Referral contract is responsible for the referral program which allows users to refer others to earn rewards for introducing users to the protocol. The issue lies within the `registerReferral` function where a malicious user can add themselves as a referee which will in turn cause `getReferrer` to return true as a valid referral. The malicious payload would also add earnings against the referee. Once this has been executed, the phisher is free to claim earnings.

Recommendation:

it's recommended that register referral is modified to not allow EOA's to add themselves as a referral.

>= Instead Of >

Auto deleveraging is a feature where if the compartment's Pnl is higher than the ADL ratio then the balance is auto deleveraged , ADLValue is calculated as follows →

```
uint256 ADLValue = (s.ADLPercentage[_address] * (compartment.balance)) /
BASIS_POINTS_DIVISOR;
```

According to the AppStorage definition of ADLPercentage , it is the percentage at which the ADL will hit , meaning when global Pnl equals (or above) the ADLValue then ADL should take place , therefore instead of

```
if (uint256(globalPnL) > ADLValue) {
            return true;
        }
```

It should be

```
if (uint256(globalPnL) >= ADLValue) {
            return true;
```

Recommendation:

Make the suggested change.

Multiple Functionalities Will Break If USDC Turns On Fee-On-Transfer

In the deposit contract , when depositing we perform the following →

```
IERC20(usdc).safeTransferFrom(msg.sender, address(this), _amount);
        balances[msg.sender] += _amount;
        emit Deposited(msg.sender, _amount);
```

But if USDC (which is upgradeable) turns on its feature for fee-on-transfer , the amount which is transferred into the contract would be less than the amount in the balances[] mapping , this accounting can break the entire deposit contract , since there are lesser funds in the contract as compared to the balances[] stored in the state it might lead to revert while withdrawing.

Recommendation:

Calculate the amount of USDC prior to the transfer and calculate the amount after the transfer and only account for the difference between the two.

Improper use of the whenNotPaused modifier for liquidations may result in the protocol taking on bad debt or resulting in users being liquidated as soon as unpause is executed

The Escrow contract contains the function `liquidatePosition` which is responsible for liquidating users from the protocol when they haven't maintained a healthy state of over collateralisation as a result of the markets moving unfavorably. The `whenNotPaused` modifier effectively prevents the liquidation functionality from executing when the contract is paused. This may cause issues for both the protocol and the users depending on how long maintenance or security investigations take. Firstly, a drastic market turn may cause the protocol to take on bad debt with the lack of liquidations to keep the protocol's value in a healthy position. Secondly, user positions may be sunk and are out of the control of users to bring back into a healthy state since they cannot modify their position when the protocol is paused. Once the protocol comes out of a paused state, users may be eligible for forced liquidation.

Recommendation:

There are two ways to go about fixing this issue:

- First: Remove the `whenNotPaused` modifier on liquidation functionalities to continuously allow users to be liquidated, thus keeping the protocol in a healthy state. Continuously allow users to keep their positions in a healthy state even while the contract is paused. Opening new positions will still be blocked while the protocol is paused.
- Second (and most recommended): If the protocol owners insist on having a pausable modifier for the liquidations, consider setting a grace period of a few hours to allow users to bring their positions back in a healthy state. This is what Aave 3.1's protocol upgrade did to work with pausable modifiers on liquidation.

MEDIUM-6 | RESOLVED

onlySequencer modifier in processTransaction can be bypassed in the Router contract by processing since batched transactions

The Router contract acts as the external facing interface for users to process long and short positions using signatures. The `processTransaction` function accepts a single transaction which is guarded by the `onlySequencer` modifier (client signer, requiring users to go through the frontend of the protocol to process transactions) however, the `batchProcessTransaction` is not guarded by any modifier other than `whenNotPaused`. A user can bypass the `onlySequencer` modifier by using `batchProcessTransaction` to process a single transaction.

Recommendation:

It's recommended that `onlySequencer` and `nonReentrant` is implemented for the `batchProcessTransaction` function.

MEDIUM-7 | RESOLVED

VaultFacet contains a rug vector where a rogue team member may be able to cause mass liquidations

The VaultFacet portion of the Diamond contract is responsible for various setters and configuration functions which are required to be called and set in order for the protocol to function properly. One of these functions is the `updateCollateralizationRatio` which sets the amount of collateral required in order to trade a derivative by setting a value for `s.CollateralizationRatio[_indexToken]` which the users deposited collateral must be greater than this threshold. The issue lies in the overstepping of privileges whilst users are still invested in the protocol. A malicious owner could update the collateralization ratio for an asset to cause mass liquidations opening users up to a significant centralisation risk.

Recommendation:

It's recommended that the updateCollateralizationRatio is updated through a two step function where we set and notify users of the new updated collateralization ratio into a pending state. After a reasonable period of time, the collateralization ratio can be "confirmed" and set allowing users to do their proper due diligence. In addition to this, a grace period can optionally (in the context of this issue) be introduced (perhaps a 3 or 4 hours) to allow users to bring their positions back to a healthy state or else be liquidated.

MEDIUM-8 | RESOLVED

Positions can not get liquidated when protocol is paused.

When a position becomes unhealthy a user can call `increasePosition()` within the `TradeFacet.sol`smart contract to pay the debt. However, this `increasePosition()` calls to `lockForAnOrder()` which only callable then the contract is not paused:

```
function lockForAnOrder(address _account, uint256 _amount) external onlyDiamond
whenNotPaused {
        if (_amount > balances[_account]) {
            revert LockForOrderFailed(_account);
        }
        locked[_account] += _amount;
        balances[_account] = balances[_account] - _amount;
    }
```

Consider the following scenario:

1. Alice opens a positions
2. Alice's position is healthy.
3. The protocol is paused for any reason (maintenance/security).
4. Alice's position becomes unhealthy.
5. She can not pay the debt back and will get liquidated.

Recommendation:

Make sure to add all parameters of the order struct in the orderType

MEDIUM-9 | RESOLVED

Back-running attacks can liquidate transactions without any option to repay the debt after changing the collateralization ratio.

For a position to become liquidable the following conditions must be met:

`currentCollateral <= (s.CollateralizationRatio[_indexToken] * position.collateral) / BASIS_POINTS_DIVISOR`.

This is validated by the `_validateliquidation()` function within the `TradeFacet.sol` smart contract.

Regarding the above mentioned condition, it can be observed that `s.CollateralizationRatio[_indexToken]` is a key factor for determining if the position is liquidable or not. This factor can be changed without any grace period by calling `updateCollateralizationRatio()` by a new value by the owner. If an attacker executes a back-running attack for liquidating the position just after it became liquidable the user will not be able to pay the debt to avoid the liquidation.

Consider the following scenario:

1. Position is healthy.
2. Collateralization ratio is changed.
3. Position suddenly becomes unhealthy
4. Protocol liquidator calls `transferPosition` in order to get liquidated in the Escrow.sol contract.
5. Attacker executes a back-running attack placing a transaction calling `liquidatePosition.sol` just after the position has been transferred to the Escrow.sol contract.
6. User's position gets liquidated without the possibility of paying the debt as a result of the collateralization ratio change.

Recommendation:

Add a mechanism to not change the collateralization ratio without a previous warning or grace period.
For example, use a 2 step mechanism:

1. Set the new value for collateralization ratio but it is not applied after X time.
2. After X time call to the second function to apply the change.

An alternative solution: Add a grace period for existing positions, give users some time to get their positions back to health and new positions created after the change are subject to the new ratio

Missing access restrictions modifier.

The functions `registerReferral()` and `generateMyCode()` within the `Referral.sol` smart contract are not implementing any access restriction modifier which would allow any user to call this functions while they should only be callable by the diamond.

Scenario:

1. Any external user calls `generateMyCode` to generate a referral code without any restriction.
2. Unauthorized users can call `registerReferral` to register themselves using any referral code.
3. This can lead to unauthorized and fraudulent registrations and referral code generation, potentially manipulating referral rewards.

Recommendation:

Add `onlyDiamond()` modifier to these functions.

Transaction displacement can lead to the referrer not being able to earn.

The `registerReferral()` function in the Referrals.sol smart contract is used to assign a referral code to a new trader, or in other words, register a referred user by a referrer. The registration involves several parameters of information, such as the registration time. This registration time is later used by the `isValidForEarning()` function to check if enough time has passed since the user was referred. The issue is related to the registration time, which is set to block.timestamp:

```
Registration memory registration =
            Registration({ registrationTime: block.timestamp, referralCode: referralCode,
hasRegistered: true });
```

When the `registerReferral` transaction is waiting in the mempool to be added to a block, it is visible to anyone. An attacker could initiate a displacement transaction attack, causing the transaction to be delayed and added to later blocks. This delay can be prolonged for several blocks, leading to the referrer being registered much later than expected. This directly affects the validity of the referrer to earn funds because the time passed since the referral will be shorter, potentially causing the referrer to become invalid for earning due to this time displacement.

Recommendation:

Add a `deadline` parameter to the function and check if the transaction is executed after the deadline, if so, revert the transaction.

It is not possible to add only 1 asset to the vault.

The `addNewAsset()` function within the `VaultFacet.sol` smart contract is used to set compartment percentages to assets. The total sums of percentages should always hold to be 100%, this is checked by the function.

```
require(totalPercent == 10_000, "percentages not equals 100%");
```

The `addNewAsset()` function calls `assignCompartment()`. This second function implements a new `require` statement which is not compatible with the last one when only 1 asset is wanted to be added:

```
function assignCompartment(address _asset, uint256 _percentage) internal {
        require(_percentage < BASIS_POINTS_DIVISOR, "invalid value");        Compartment
storage compartment = s.compartments[_asset];
        compartment.assignedPercentage = _percentage;
        s.compartments[_asset] = compartment;
        emit CompartmentAssigned(_asset, _percentage);
    }
```

Consider the following scenario:

1. AssetA is added with _percentage = 10_000. The first require is met.
2. As _percentage is 10_000 the second `require` reverts.

Recommendation:

Change the require statement within the `assignCompartment` function:

`require(_percentage <=  BASIS_POINTS_DIVISOR, "invalid value");`

MEDIUM-13 | RESOLVED

Vault and diamond assigned to the same address

`vault` and `diamondAddress` within `LpToken.sol` are being assigned to the same address using the `setDiamondAddress` function. These addresses are used for important logic like modifiers.

```
function setDiamondAddress(address _diamondAddress) public onlyGov {
        require(_diamondAddress != address(0), "Invalid address");
        vault = _diamondAddress;
        diamondAddress = _diamondAddress;
        emit SetVault(_diamondAddress);
    }
```

Recommendation:

Define each address separately or if they should be threatened equally, remove one variable.

MEDIUM-14 | RESOLVED

`maxReferralEarningTime` should be within a defined threshold to avoid excessive waiting for earning

The function `setMaxReferralEarningTime` within the `Keeper.sol` smart contract is used for setting `maxReferralEarningTime`'s value. This variable is used in the `isValidForEarning` function within the `Referral.sol` smart contract to check if enough time has passed for a referrer to earn. The required time is computed in the following way:

```
registeredUser.registrationTime + (maxDaysForEarning * 1 days) >= block.timestamp
```

As it can be seen, the variable `maxDaysForEarning` is multiplied by `1 days`, this means that the `maxDaysForEarning` should be set in terms of single number, e.g 180, avoiding using the ' days' notation, e.g 180 days.

If the `setMaxReferralEarningTime` does not define a threshold for the new `maxReferralEarningTime` value, the above mentioned requirement can be bypassed.

Recommendation:

Define a threshold within the `setMaxReferralEarningTime` function to not allow setting a new value outside the threshold.
The threshold can also be defined as global variables set by other 'set functions', so that the threshold is dynamic.

MEDIUM-15 | RESOLVED

`collateralizationRatio` can be set higher than 100% which would result in no positions getting liquidated.

The `updateCollateralizationRatio()` within the `VaultFacet.sol` function is used by the owner to change the collateralizationRatio of an asset. It is possible to set the value higher than 100% by error because there are no restrictions.

The `CollateralizationRatio` value is used for checking if a position is in a liquidable state or not. As a result of setting its value higher than 100% would result in no positions getting liquidated:

```
if (currentCollateral <= (s.CollateralizationRatio[_indexToken] * position.collateral) /
BASIS_POINTS_DIVISOR) {
            return (true, currentCollateral);
        }
```

The contract owner sets the collateralization ratio for a particular token to zero using the `updateCollateralizationRatio` function.

1. A user with an under-collateralized position in that token will not be liquidated, as the `validateLiquidation` function checks if the current collateral is less than or equal to zero, which is always false.
2. This allows users to maintain positions without sufficient collateral, leading to potential insolvency and an unfair advantage over other users.
3. In a worst-case scenario, this could result in a complete collapse of the collateralization mechanism, causing substantial financial damage to the protocol and its users.

Recommendation:

1. Implement a minimum allowable collateralization ratio to prevent the owner from setting it to zero.
2. Add a validation check within the `updateCollateralizationRatio` function to ensure the new value is greater than zero and within an acceptable range (e.g., less than or equal to 100%).

MEDIUM-16 | RESOLVED

Possible Denial of Service Condition in the LpToken Contract When Attempting To Withdraw From The Protocol.

The `LpToken.sol` contract is a pausable contract which implements a `whenNotPaused` modifier. This modifier is used to disallow deposits if any security risk is found. However, the modifier is also used for the `withdraw` and `redeem` functions, making it impossible for the users to withdraw their funds from the protocol.

Recommendation:

Only use pausable mechanisms for deposit functions and avoid using them for withdrawals.

MEDIUM-17 | RESOLVED

`allowUSDCtoVault` increases allowance by X instead of setting it to X.

The `allowUSDCtoVault` function withing the `MMTrade.sol` smart contract is used for setting MMTrade to Trade's allowance to an specific amount:

```
/// @notice Allows the vault to spend a specified amount of USDC on behalf of the trade
contract
    /// @param _amount The amount of USDC to allow the trade contract to spend
    function allowUSDCtoVault(uint256 _amount) external onlyOwner {
        usdc.safeIncreaseAllowance(trade, _amount);
    }
```

However, the function is not setting the allowance to X as expected but increasing it by X. Consider the following scenario:

1. Actual allowance is 100.
2. `alloUSDCtoVault` is called with `_amount` equal to 50.
3. Allowance is expected to be set to 50 but instead it is set to 150 (100 + 50).

The function is not following the expected behavior indicated by the NatSpec comments.

Recommendation:

There are 2 possible options:

1. Call `forceApprove` to set the approved value directly.
2. First call `allowance` and calculate the value that should be passed to `safeIncreaseAllowance` in order to set the new value to `amount`.

Client comment: Valid Issue. MMTrade Contract is not used currently.

MEDIUM-18 | RESOLVED

`reservedSizeShort` can not reach its maximum allowed value

The `_increaseGlobalShortSize` function within the `TradeFacet.sol` smart contract is used to increase the `reservedSizeShort` of a specific token. It should fulfill that the new value for `reservedSizeShort` if within the allowed `MaxShortSize`. However, the `require` statement is wrongly implemented as it does not allow `_increaseGlobalShortSize` to reach `MaxShortSize`.

```
function _increaseGlobalShortSize(address _token, uint256 _amount) private {
        s.reservedSizeShort[_token] = s.reservedSizeShort[_token] + _amount;
        uint256 MaxSize = s.MaxShortSize[_token];

        if (MaxSize != 0) {
            require(MaxSize > s.reservedSizeShort[_token], "Vault: max shorts exceeded");
}
        emit GlobalShortSizeIncreased(_token, _amount);
```

```
    }
```

Consider the following scenario:

1. `MaxShortSize` is set to X.
2. The new value for `reservedSizeShort` is X.
3. The `require` statement is not met as it only allows X - 1 at maximum but it should allow X as it is the maximum value.

Recommendation:

Consider changing the `require` condition from `>` to `>=`.

MEDIUM-19 | RESOLVED

Loss of decimal precision for `currentFundingTime`

The `settleFundingPayments` function in the `Formula.sol` smart contract contains a `currentFundingTime` variable that is calculated as:

```
uint256 currentFundingTime = (block.timestamp / fundingInterval) * fundingInterval;
```

As it can be seen, `currentFundingTime` is equal to itself but divided and then multiplied by `fundingInternal`. This division can lead to loss of decimal precision.

Recommendation:

Do not divide and multiply by `fundingInternal`. Set `currentFundingTime = block.timestamp`

`updateBalancesForAll()` and `updateBalances()` functions are callable when the contract is paused.

The `updateBalancesForAll()` function within the `TradeFacet.sol` smart contract should only be callable when the contract is not paused as stated in the NatSpec comments: 'This function is only callable when the contract is not paused.'. However, the `whenNotPaused` modifier is missing. The `TradeFacet.sol` contract is neither pausable.

```
/// @notice Updates the balances for all index tokens in the contract.
   /// @dev Iterates through all index tokens stored in `s.allIndexTokens` and calls
`updateBalances` for each.
```

```
   ///      This function is only callable when the contract is not paused.
   function updateBalancesForAll() public {
       uint256 len = s.allIndexTokens.length;
       for (uint256 i = 0; i < len; ++i) {
           updateBalances(s.allIndexTokens[i]);
       }
   }
```

The same scenario is present within the `updateBalances()` function.

Recommendation:

1. Set `TradeFacet` as a pausable contract.
2. Implement the `pause()` and `unpause() functions.
3. Add a `whenNotPaused` modifier to the function.

Transaction displacement lead to force users to stake more time than expected

The functions `addLiquidity` and `reduceLiquidity` within the `MMTrade.sol` smart contract serve for adding and reducing liquidity to the vault, respectively. When a user executes `addLiquidity`, the `lastAdded` mapping is updated:

```
function addLiquidity(uint256 _amount) public nonReentrant whenNotPaused {
        require(_amount > minUSDC, "MMTrade: Amount less than minimum USDC");

        lastAdded[msg.sender] = block.timestamp;
        usdc.safeTransferFrom(msg.sender, address(this), _amount);

        _mint(msg.sender, _amount);

        emit BotLiquidityAdded(msg.sender, _amount);
    }
```

This value is later used when the user decides to reduce liquidity by calling `reduceLiquidity`. A minimum time needs to be elapsed since adding liquidity in order to allow a user to reduce it.

```
require(lastAdded[msg.sender] + mimTime <= block.timestamp, "MMTrade: Liquidity cannot be
removed yet");
```

If the transaction for adding liquidity is displaced then the user will need to wait more time than expected in order to reduce liquidity.

Consider the following scenario:

1. `minTime` is set to 1 day, for example.
2. Alice executes `addLiquidity`.
3. Alice's transaction remains in the mempool waiting for being added to a block.
4. Attacker sees the transaction and front runs it in order to displace the transaction and not being included in the block.
5. Attacker continues repeating the same attack, displacing the transaction more.
6. Finally, the transaction is added to a block, after X amount of time.
7. The user will have to wait `minTime + displaced time` (1 day + X) in order to reduce liquidity.

Recommendation:

Add a deadline parameter to the `addLiquidity` function, if the transaction is executed after the deadline revert. In this case the user could decide to submit the transaction again or not.

LOW-2 | RESOLVED

Optimal utilization percentage can be set to more than 100%

The `addOptimalUtilization()` function within the `VaultFacet.sol` smart contract is used to set an optimal utilization percentage for a specific token but there are not checks to ensure that the new value is within a threshold, less or equal to 100% percentage value.

```
/**
    * @notice Adds optimal utilization percentage for a specific index token.
    * @dev Can only be called by the contract owner (governance).
    * @param _percentage The optimal utilization percentage.
    * @param _indexToken The address of the index token.
    */
   function addOptimalUtilization(uint256 _percentage, address _indexToken) external
onlyOwner {
       s.optmialUtilization[_indexToken] = _percentage;
   }
```

Recommendation:

Add a require to ensure that the value is less or equal to 100%:

```
require(_percentage < BASIS_POINTS_DIVISOR, "invalid value");
```

LOW-3 | RESOLVED

`init` function can be called with malicious values by attacker.

The `init` function within the `DiamondInit.sol` smart contract implements an `initializer` modifier which only allows calling this function once but it does not implement an access restriction modifier to only allow a trusted address to call this function. If the function is not executed in the same tx that the deploy it could be front-run by an attacker to set unexpected values.

Recommendation:

There are two possible solutions:

1. Add an access restriction modifier to only allow the owner to execute the function.
2. Ensure that the `init` call is executed in the same transaction as the deploy.

LOW-4 | RESOLVED

Missing Referrer Existence Check in `generateMyCode`

Description: In the referral.sol There's no check to ensure that the `referrer` address exists and is valid before generating a referral code.

Scenario:
1. A user could inadvertently or maliciously generate a referral code for an invalid address.
2. This could result in referral codes being assigned to unintended or incorrect addresses.

Recommendation:
Add a check to ensure that the `referrer` address is valid and not zero.

LOW-5 | RESOLVED

Non-zero Amount Validation in `addEarnings`

Description: In the referral.sol The `addEarnings` function does not check if the `amount` is greater than zero. Adding zero earnings does not make sense and could lead to unnecessary event emissions.

Scenario:
1. A user could call `addEarnings` with an amount of zero.
2. This would result in unnecessary state changes and event emissions.

Recommendation:
Add a check to ensure that the `amount` is greater than zero.

Position leverage is hardcoded when transferring position as a result of a liquidation.

When a position becomes liquidable it is transferred to the `Escrow.sol` contract to get it later liquidated by the protocol liquidator bot or an external liquidator by calling `liquidatePosition()`. When the position is transferred, it is transferred with a x20 leverage.

```
Position memory newPosition = Position({
            size: _getliqPositionSize,
            collateral: _getliqPositionSize / 20,
            averagePrice: collateralAtliquidationPrice / leverage,
            entryFundingRate: position.entryFundingRate,
            entryCumulativeInterestRate: position.entryCumulativeInterestRate,
            reserveAmount: position.reserveAmount,
            viaOrder: position.viaOrder,
            realisedPnl: 0,
            isLong: position.isLong,
            lastIncreasedTime: position.lastIncreasedTime,
            _indexToken: indexToken
        });
```

The used leverage is hardcoded to x20, this is a business decision. However, the conditions may change in the future and the leverage would not be possible to get changed directly.

The hardcoded 20 is also used in `_checkPositionForLiquidation`:

```
function _checkPositionForLiquidation(
        address _account,
        address _indexToken,
        bool _islong,
        uint256 _currentCollateral
    ) internal view returns (uint256, uint256, uint256, uint256, uint256, uint256) {
        bytes32 key = getPositionKey(_account, address(0), _indexToken, _islong);
        Position memory position = getPosition(key);
        uint256 decreasedCollateralValue = position.collateral - _currentCollateral;
        uint256 _getliqPositionSize = getLiquidationPositionSize(_account, _indexToken,
_islong);
        uint256 leverage = (position.size / position.collateral);
        uint256 collateralAtliquidationPrice = _getliqPositionSize / leverage;
        uint256 liquidatorCollateralNeeded = (_getliqPositionSize / 20);
        uint256 collateralAfterFeesDeduction = _currentCollateral;

        return (
            _getliqPositionSize,
            collateralAtliquidationPrice,
            leverage,
            decreasedCollateralValue,
            collateralAfterFeesDeduction,
            liquidatorCollateralNeeded
        );
    }
```

Recommendation:

Add a `set` function which implements an access restriction modifier to change the leverage factor used when liquidating a position. It is recommended using a 2 step function with a grace period so that an important change is not directly made effective.

LOW-7 | RESOLVED

Unlimited Registrations with Same Referral Code

Description: In the referral.sol The current implementation allows unlimited registrations using the same referral code, which can be exploited by creating multiple accounts to earn referral rewards illegitimately.

Scenario:

1. A user generates a referral code using `generateMyCode`.
2. The user creates multiple Ethereum addresses.
3. The user registers each address using the same referral code by calling `registerReferral` multiple times.
4. The user then triggers referral earnings for each registered address, accumulating illegitimate rewards.

Recommendation:

Introduce a limit on the number of registrations allowed per referral code.

LOW-8 | RESOLVED

Lack of Self-referral Prevention

Description: In the referral.sol The contract does not prevent users from registering themselves using their own referral code, leading to potential abuse where users earn rewards illegitimately.

Scenario:

1. A user generates a referral code.
2. The user registers themselves using their own referral code.
3. The user triggers referral earnings, benefiting from self-referrals.

Recommendation:

Add a check in the `registerReferral` function to prevent self-referrals.

LOW-9 | RESOLVED

Duplicated function

The function `claimable` in the `ViewFacet.sol` smart contract is a duplicate of the `getClaimableAmount` function in the `VaultFacet.sol` smart contract. There are some small variations between them and the `getClaimableAmount` is the one that has been tested within the contracts repository.

Recommendation:
Remove the `claimable` function from the `ViewFacet.sol` smart contract and use the `getClaimableAmount` function instead.

INFORMATIONAL-1 | ACKNOWLEDGED

minUSDC Can Be Bypassed While Adding Liquidity

In the MMTrade contract liquidity can be added using the addLiquidity function which verifies if the amount of liquidity added respects the minimum amount allowed .

```
require(_amount > minUSDC, "MMTrade: Amount less than minimum USDC");
```

Since minUSD is set in the initializer an attacker can frontrun the initializer and call the addLiquidity function with an amount less than the minUSDC (since it is `0` before initialization).

Recommendation:
Ensure initialization is a part of the deploy script
Client comment: We deploy MMTrade with initialization.

INFORMATIONAL-2 | RESOLVED

Incorrect NatSpec comment about decimals

The `InterestRateParameters` struct within the `Formula.sol` smart contract contains this comment: `All parameters are with 6 decimals.` that is later contradicted by other comments. It has been confirmed with the client that this comment is incorrect.

Recommendation:
Delete the incorrect comment.

INFORMATIONAL-3 | RESOLVED

updateBalances Can Return Early

In the updateBalances function we can return early in the first if branch i.e. if the if branch is executed then there is no need to execute the rest of the code because the interval would be calculated as `0` meaning `0` interest accrued over time.

Recommendation:
Change the code to →

```
if (s.lastBalanceUpdateTime[_indexToken] == 0) {
            s.lastBalanceUpdateTime[_indexToken] = block.timestamp;
            return;
        }
```

INFORMATIONAL-4 | RESOLVED

_transferIn Function Can Be Removed

The _transferIn function in the trade facet contract is not being used anywhere , this function can be removed.

Recommendation:
Remove the _transferIn function from the trade facet.

INFORMATIONAL-5 | RESOLVED

_validateOrder Should Have A Sanity Check For Collateral Too

_validateOrder in the Router contract is used to validate an order and has a sanity check for the index size ,

```
if (position.size != order.amount) {
                order.amount = Math.min(order.amount, position.size);
            }
```

But there is no sanity check for collateral i.e. if we are reducing more collateral than the position actually has.

Recommendation:
Have a sanity check for collateral too

Contract not pausable

The `DiamondInit.sol` smart contract is `PausableUpgradeable` but it does not implement any pause/unpause functions. The `_pause()` and `_unpause()` functions from `PausableUpgradeable` are internal functions so that if the contract that inherits does not call these functions then it will not be pausable.

Recommendation:
Add pause/upause functions:

```
/**
    * @notice Pauses the contract, preventing certain actions from being executed.
    * @dev Can only be called by the contract owner (governance).
    */
   function pause() external onlyOwner {
       _pause();
   }

   /**
    * @notice Unpauses the contract, allowing paused actions to resume.
    * @dev Can only be called by the contract owner (governance).
    */
   function unpause() external onlyOwner {
       _unpause();
   }
```

INFORMATIONAL-7 | DUPLICATE

Static Referral Rewards

Description: In the referral.sol Referral rewards are static and do not take into account the activity level of the referred users, potentially leading to abuse where users create inactive accounts to earn rewards.

Scenario:

1. A user generates a referral code.
2. The user creates multiple accounts and registers them.
3. The user triggers referral earnings regardless of the activity level of the referred accounts.

Recommendation:

Adjust referral rewards based on the activity of the referred user. For example, require certain actions (e.g., making a deposit, completing a trade) before referral earnings are granted.

INFORMATIONAL-8 | RESOLVED

Missing events

The following functions are missing events when key values are updated.

- setDiamondContract() function in the Deposit contract
- setProtocolLiquidator() and setDiamondContract() functions in the Escrow contract
- addSequencer(), addDiamond(), and setFundingInterval() functions in the Formula contract.

Recommendation:

Add relative events based on the variables to be updated.

INFORMATIONAL-9 | RESOLVED

Unnecessary if keyword

In the substractFundingRates() function of the Formula contract, currentFundingRate._direction is checked with the _entryFundingRate._direction. However, in the else case, no need to check if currentFundingRate._direction is not equal to the _entryFundingRate._direction.

Recommendation:
Replace “else if” with “else”.

INFORMATIONAL-10 | RESOLVED

Reading the array length from storage costs more gas

In the liquidatePosition() function of the Escrow contract, liquidablePositions.length is read at every literal.

Recommendation:
Cache the liquidablePositions.length with a memory variable.

INFORMATIONAL-11 | RESOLVED

Unused variables

gov and BASE_RATE variables in the Formula contract are not used at all.

Recommendation:
Remove the unused variables.

INFORMATIONAL-12 | RESOLVED

Unnecessary assignment

In the assignCompartment() function of the VaultFacet contract, s.compartments[_asset] is updated with compartment. However, compartment is already set to storage, so no need to assign again.

Recommendation:
Remove the assignment line.

INFORMATIONAL-13 | RESOLVED

Not only the keeper can call `addEarnings` but anybody

The `addEarnings()` function's NatSpec comment within the `Referrals.sol` smart contract states that: ` Called by the keeper contract to distribute the referralEarnings to the user`. However the `addEarnings()` function does not implement any access restrictions so anybody can call the function.

```
function addEarnings(address referred, uint256 amount) external returns (bool) {      (bool
isValidReferrer, address referrer) = getReferrer(referred);
```

```
        if (!isValidReferrer) return false;

        referralEarnings[referrer] += amount;
        USDC.safeTransferFrom(msg.sender, address(this), amount);
        emit EarningsAdded(referred, referrer, amount);
        return true;
    }
```

Recommendation:
Implement an onlyKeeper modifier that checks if the caller is the owner, if not, revert the transaction.

New findings after fixes:

HIGH-1 | ACKNOWLEDGED

Users Might Be Unable To Make Their Position Healthy

When new index tokens are added (refer finding 14) the isIndexToken[] mapping is toggled to false for all the older index tokens (VaultFacet.sol) . Users who had a position open previously with tokenA(say) and that position is coming close to liquidation , they would want to use "_increasePosition()" to make their position healthy , but if in between this the position's index token (tokenA) has been toggled to false they would not be able to increase the position since there is a requirement in the function _increasePosition() →

```
require(s.isIndexToken[_incPosParams.indexToken], "index token is not valid");
```

and the key of the position is dependent on the position's index token →

```
bytes32 key = getPositionKey(_incPosParams.account, address(0),
_incPosParams.indexToken, _incPosParams.isLong);
```

Therefore , this would force the position to be liquidated

Recommendation:

Make sure there are no positions open corresponding to the older index tokens

	./Deposit.sol ./facets/DiamondCutFacet.sol ./facets/OwnershipFacet.sol ./facets/TradeFacet.sol ./facets/ViewFacet.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./facets/DiamondLoupeFacet.sol **./facets/VaultFacet.sol** **./upgradeInitializers/DiamondInit.sol** **./Governor.sol** **./libraries/LibDiamond.sol**
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./Formula.sol ./Escrow.sol ./LpToken.sol ./MMTrade.sol ./Diamond.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./Router.sol ./Referrals.sol ./Keeper.sol ./AppStorage.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Filament team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Filament team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo