

Aori Security Review

May 9th, 2023

Report Prepared By:

Georgi Georgiev (Gogo), Independent Security Researcher

Audited through the **Hyacinth** platform.

Table of contents

Disclaimer

Risk classification

- Impact
- Likelihood
- Actions required by severity level

Executive summary

- Summary
- Scope

Findings

- Critical severity
 - Borrowers can wait to get back their full collateral when an option expired ITM.
 - Positions will become non-liquidatable when the collateral value is too low.
 - Borrowers can open and settle their position without paying any interest.
 - An attacker can remove the collateral of all borrowers.
- High severity
 - Router keepers can drain manager vaults entirely.
 - Assumption on collateral token precision leads to the wrong liquidation flow.
 - Wrong interest is paid to lenders for call options.
 - DoS after the first position for zero-to-non-zero allowance tokens is opened.
 - All collateral from rejected position requests will be stuck in the router.
 - Wrong price will be returned when the token's USD price feed's decimals != 8.
 - Privileged liquidators can liquidate any borrowers at any time.
 - First depositor inflation attack on the vaults.
- Medium severity
 - Malicious users can grief borrowers to prevent them from adding collateral.
 - Initial margin value is incorrectly computed for call options.
 - Token vaults and price oracles can be overwritten in the margin manager.
 - User positions will be overwritten when executed in the same block.
 - Hardcoded USDC decimals assume the same precision across different blockchains.
 - The position router can be re-initialized by the contract owner.
 - Rebasing and fee-on-transfer underlying tokens are not handled properly.
 - Chainlink oracle tolerance will exceed 2 hours for most price feeds.
 - Arbitrum sequencer should be checked if down.
 - Inherited methods from ERC4626 can be used instead of the custom ones.
 - Missing input validation on privileged functions.
 - Non-standard ERC20 tokens vulnerabilities.

- Low severity
 - Missing access control for `settlePosition` and `addCollateral`.
 - Wrong function return values.
 - `accruePositionInterest` should return instead of `revert`.
 - Missing event.
 - Unreachable code.
 - Not all ERC20 tokens implement `decreaseAllowance`.
 - Wrong `secondPerYear` value.
 - Misleading comment regarding configuration variable precision.
 - Unnecessary approvals.
 - Round staleness check on price oracle.
- Informational
 - Unused variables.
 - Redundant imports.
 - Typographical mistakes.
 - Wrong string revert message.
 - Public functions can be marked `external`.

Disclaimer

Audits are a time, resource and expertise bound effort, where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities but not their absence

Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost, or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behaviour, but there are no funds at risk.

Likelihood

- High - there is a direct attack vector, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only conditionally possible incentivized attack vector, but still relatively likely.
- Low - depends on too many or too unlikely assumptions or requires a huge stake by an attacker with little or no incentive.

Actions required by severity level

- Critical - client must fix the issue.
- High - client must fix the issue.
- Medium - client should fix the issue.
- Low - client could fix the issue.

Executive summary

Summary

Protocol name	Aori
Repository	https://github.com/elstongun/Aori
Commit hash	b9bdd443a71a77214858349fd7466993c23921ce
Methods	Manual review

Scope

- src/Margin/MarginManager.sol
- src/Margin/PositionRouter.sol
- src/Margin/Structs.sol
- src/Margin/Vault.sol

Findings

Critical severity

C.01. Borrowers can wait to get back their full collateral when an option expired ITM.

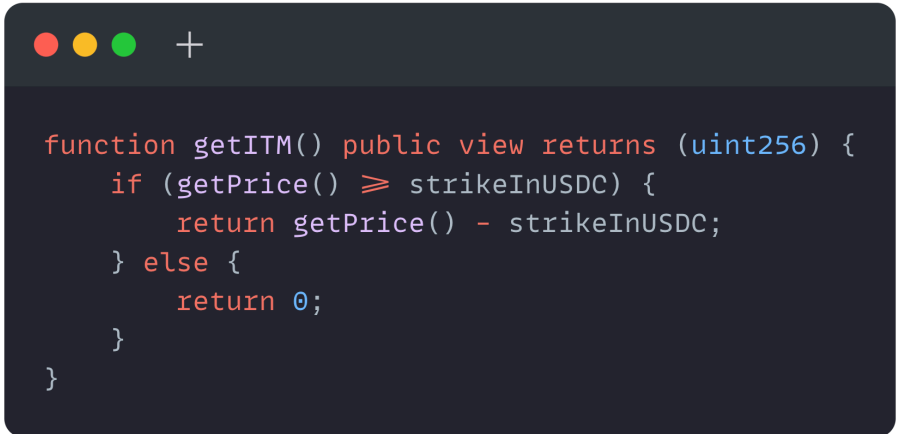
The current flow of settling a position is as follows:

1. Call `MarginManager.settlePosition`.
2. Check whether the option is ITM or OTM.
3. Depending on the status of the option, call `Vault.settleOption` with the appropriate parameters.
4. If this is the first time this function is executed, the option will be settled, and the vault will receive back the collateral. Otherwise, `vault.settleOption` will simply return.
5. Borrower's collateral will be returned either in full amount (OTM) or with deducted loss (ITM).

The problem occurs in step 2. when the `MarginManager` calls `position.option.getITM()` to determine whether the option expired ITM or OTM.

The first time the above flow is executed, everything will go as intended. However, for all sequential borrowers that settle their positions, the position status might be wrong since `position.option.getITM()` will return the current ITM/OTM status of the option and not the one on settlement.

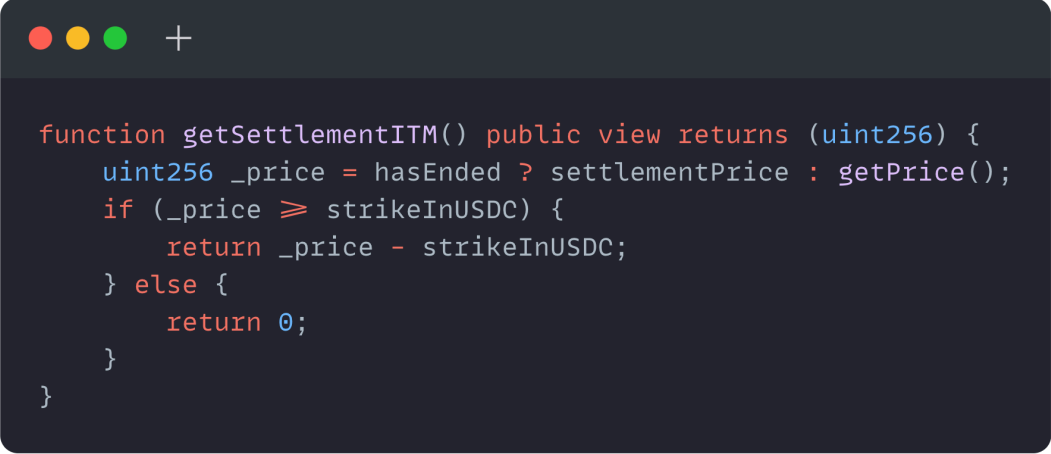
When we look up the implementation of `getITM()` in `OptionCall`, it can be seen that it compares the current price of the underlying asset retrieved from `getPrice()` to the strike price:



```
function getITM() public view returns (uint256) {
    if (getPrice() >= strikeInUSDC) {
        return getPrice() - strikeInUSDC;
    } else {
        return 0;
    }
}
```

So the vulnerability is that all borrowers after the first one can abuse this to always receive their full collateral, even if the options expired ITM, by simply waiting for the price of the underlying asset to go in the opposite direction so the current option status becomes OTM again.

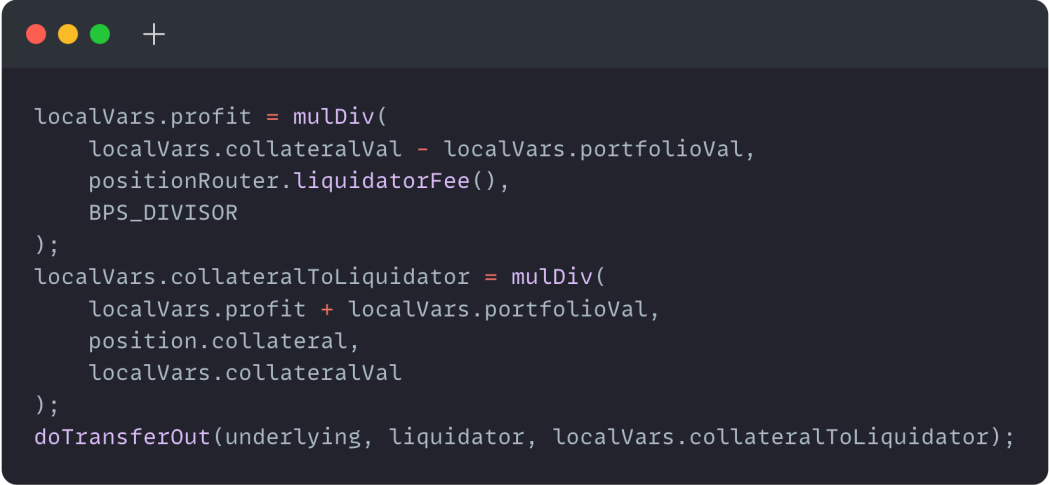
To fix this, the correct implementation should use the `settlementPrice` instead of the current one retrieved from `getPrice()` after the option has expired:



```
function getSettlementITM() public view returns (uint256) {
    uint256 _price = hasEnded ? settlementPrice : getPrice();
    if (_price ≥ strikeInUSDC) {
        return _price - strikeInUSDC;
    } else {
        return 0;
    }
}
```

C.02. Positions will become non-liquidatable when the collateral value is too low.

The following formula is used to determine the collateral to be given to the liquidator in `MarginManager.liquidatePosition`:



```
localVars.profit = mulDiv(
    localVars.collateralVal - localVars.portfolioVal,
    positionRouter.liquidatorFee(),
    BPS_DIVISOR
);
localVars.collateralToLiquidator = mulDiv(
    localVars.profit + localVars.portfolioVal,
    position.collateral,
    localVars.collateralVal
);
doTransferOut(underlying, liquidator, localVars.collateralToLiquidator);
```

The `collateralVal` is based on the price of the underlying asset retrieved from the Chainlink price feed, while the `portfolioVal` is based on the `fairValueOfOption` passed by the permissioned liquidator and the `optionSize` of the position to liquidate.

A position becomes liquidatable once the `collateralVal` falls under a certain threshold set in the `PositionRouter`. Upon deployment, this threshold is set to 200% (of the `portfolioVal`).

The vulnerability is that if the collateral value falls significantly below the threshold to less than 100% of the position value and the position is not liquidated yet, it will become non-liquidatable since the subtraction on the second line of the above snippet will revert due to underflow.

Consider refactoring the liquidator profit formula.

C.03. Borrowers can open and settle their position without paying any interest.

The `accruePositionInterest` function in `MarginManager` is used to take a certain percentage of the borrower's collateral and pay it to the lenders in the corresponding vault when the position is updated, either when collateral is added or on settlement.

However, this function is not called in `MarginManager.settlePosition`, which means borrowers are not required to pay any interest for their position before settlement, resulting in a significant loss of yield for all lenders.

To fix this issue, `accruePositionInterest` should be called at the beginning of `MarginManager.settlePosition`.

C.04. An attacker can remove the collateral of all borrowers.

When borrowers come close to liquidation, they can call `MarginManager.addCollateral` to increase their own or any other position's collateral.

The previous borrower's collateral seems to be meant to be cached in the `currentCollat` local variable and then used to update the `position.collateral` in storage. However, it is left uninitialized, which means the `position.collateral` value will be overwritten by the new `collateralToAdd`:

```
function addCollateral(bytes32 key, uint256 collateralToAdd) public returns (uint256) {  
    // ...  
    uint256 currentCollat;  
    // ...  
    underlying.transferFrom(msg.sender, address(this), collateralToAdd);  
    position.collateral = currentCollat + collateralToAdd;  
}
```

Since the function has no access control, an attacker can abuse this by calling `addCollateral` for each open position, passing 0 as `collateralToAdd`, effectively breaking the entire margin system.

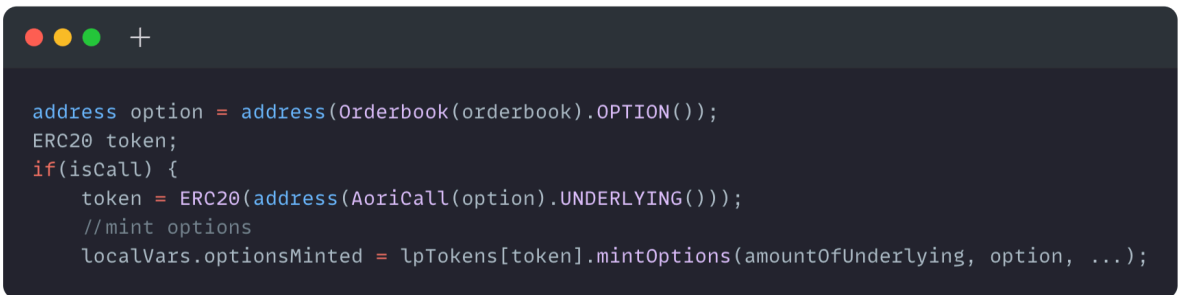
The `position.collateral` should be updated in the following way:

```
position.collateral = position.collateral + collateralToAdd.
```

High severity

H.01. Router keepers can drain manager vaults entirely.

When users want to open a position in the margin manager, they first have to make a request in the position router. Then privileged accounts with a keeper role have to either accept or reject the position request based on its parameters. If the position request is accepted, the provided collateral will be transferred to the margin manager, and the corresponding options will be minted with the amount of underlying funds borrowed from the vault.



```
address option = address(Orderbook(orderbook).OPTION());
ERC20 token;
if(isCall) {
    token = ERC20(address(AoriCall(option).UNDERLYING()));
    //mint options
    localVars.optionsMinted = lpTokens[token].mintOptions(amountOfUnderlying, option, ...);
}
```

The above code snippet is from `MarginManager.openShortPosition`. The option address and the underlying token address are the main parameters of the option request struct that determine which option should be minted and which vault should be used. The problem is that the orderbook parameter is never validated, neither in the margin manager nor in the router.

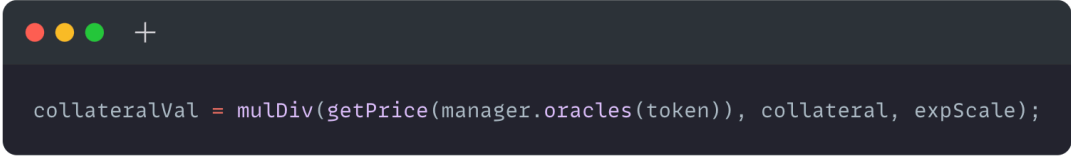
Therefore, a potential rug pull vector is as follows:

1. A router keeper creates a position request with a malicious mocked orderbook contract's address.
2. The collateral for this position is 0, and `type(uint256).max` is passed as the `underlyingAmount`.
3. The router keeper accepts and therefore executes the position request.
4. A mock option contract is returned, and a real underlying token is used, which has a corresponding vault with, let's say, 1M USDT lenders' funds.
5. The mock option contract address is passed to `vault.mintOptions()`.
6. `underlyingAmount` is then approved to the malicious option contract, and therefore the whole vault balance can be drained.

Even though the router keeper is considered a trusted privileged account, it is recommended to prevent the risk described above by verifying the returned options contract address. This can be done by checking if it is deployed (`isListed`) by the options factory contracts.

H.02. Assumption on collateral token precision leads to the wrong liquidation flow.

`PositionRouter.isLiquidatable` returns the collateral value calculated in the following way:



```
collateralVal = mulDiv(getPrice(manager.oracles(token)), collateral, expScale);
```

The `expScale` variable is a constant set to $1e18$, which means that the result of the above equation, intended to be in 6 decimals precision (USDC), will only be correct when the underlying/collateral token is also in 18 decimals precision.

Since the protocol is intended to support any kind of ERC20 tokens as collateral for options, in many cases, this would result in returning a significantly lower value for the collateral of a given position, leading to unexpected liquidations.

Consider using `10**token.decimals()` instead of `expScale`.

H.03. Wrong interest is paid to lenders for call options.

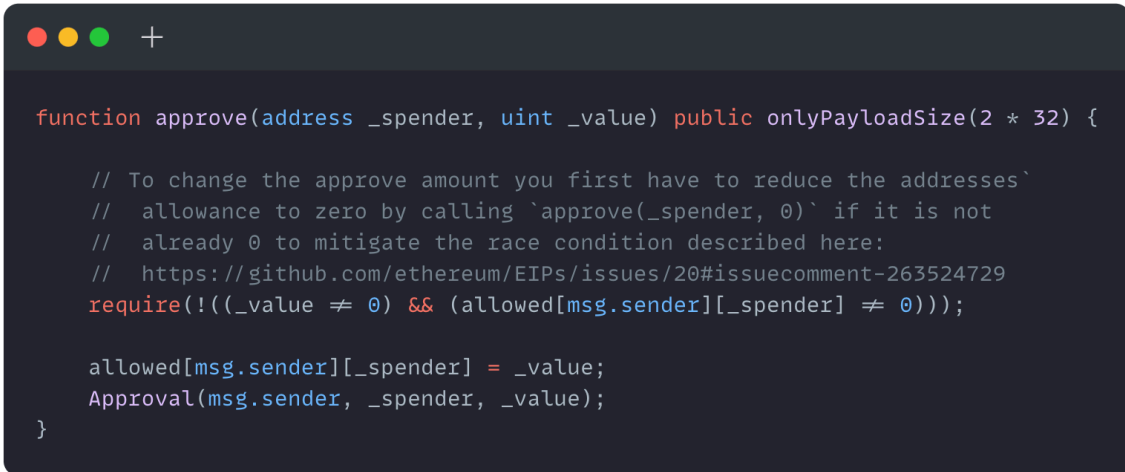
The interest owed in `accruePositionInterest` is calculated using the `positionRouter.getInterestOwed` function, which takes the type of the option (call or put) as its first parameter.

The problem is that `accruePositionInterest` always passes `false` for `isCall`, resulting in incorrect interest calculation. This can cause either excessively high or insufficient interest payments to the lenders due to the different formula and decimals used.

To resolve this issue, pass `isCall` instead of `false`.

H.04. DoS after the first position for non-zero to non-zero allowance tokens is opened.

A well-known vulnerability regarding tokens like USDT, MANA, etc. that contains the following safety check in their `.approve` function:



```
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {  
  
    // To change the approve amount you first have to reduce the addresses`  
    // allowance to zero by calling `approve(_spender, 0)` if it is not  
    // already 0 to mitigate the race condition described here:  
    // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729  
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));  
  
    allowed[msg.sender][_spender] = _value;  
    Approval(msg.sender, _spender, _value);  
}
```

The `PositionRouter` approves the collateral that a user has deposited to the margin manager contract when making a position request. However, this allowance is not spent directly in the following call, which means that any sequential approval will revert, resulting in a denial-of-service (DoS) issue for opening positions.

Consider setting the allowance to 0 first and then increasing it to the desired amount.

H.05. All collateral from rejected position requests will be stuck in the router.


Currently, when users want to open a position in the margin manager, they first have to make a request in the position router. Then privileged accounts with a keeper role have to either accept or reject the position request based on its parameters.

The problem is that if the keeper decides not to execute a given position due to reasons such as a too low collateral-to-underlyingAmount ratio or malicious request parameters, the collateral that the user has already deposited will be stuck in the router contract forever. This occurs because `rejectIncreasePosition` only deletes the user's position without returning their collateral.

Consider modifying the functionality to return the user's collateral if their position request is rejected.

H.06. Wrong price will be returned when the token's USD price feed's decimals != 8.

The `PositionRouter.getPrice` function makes an assumption regarding the number of decimals in the Chainlink price feed.




```
//8 is the decimals() of chainlink oracles, return USDC scale
return (uint256(price) / (10**2));
```

However, there are cases where token/USD price feed oracles may not return a price with 8 decimals precision, but with 18 or a different number of decimals.

For example, the AMPL/USD price feed has 18 decimals (source: <https://etherscan.io/address/0xe20CA8D7546932360e37E9D72c1a47334af57706#readContract>).

Consider implementing the following change to accommodate different decimal precisions:



```
return (uint256(price) / (10**(oracle.decimals() - USDC.decimals()))
```

H.07. Privileged liquidators can liquidate any borrowers at any time.

The Aori margin protocol relies on permissioned liquidators to liquidate undercollateralized positions in the `MarginManager`.

To determine if a position is undercollateralized, the collateral amount is multiplied by the underlying token price, while the value of the option is calculated by multiplying the option size with an external input parameter called `fairValueOfOption`.

The centralisation risk arises from the fact that permissioned liquidators can input any value for `fairValueOfOption`, enabling them to liquidate borrowers' positions at their discretion.

To mitigate this risk, consider retrieving the fair value of an option from an oracle using a widely accepted pricing model like the Black-Scholes pricing model.

H.08. First depositor inflation attack on the vaults.

A well-known attack vector that exists for almost all share-based liquidity pool contracts, wherein an early user can manipulate the price per share and profit from late users' deposits due to precision loss caused by the rather large value of the price per share.

A malicious early user can call `deposit()` with 1 wei of underlying tokens as the first depositor of the vault and obtain 1 wei of shares.

Then, the attacker can send $10000e18 - 1$ asset tokens directly to the vault and inflate the price per share from 1 to $10000e18$ (calculated as $(1 + 10000e18 - 1) / 1$).

Consequently, a future user who deposits $19999e18$ will only receive 1 wei (calculated as $19999e18 * 1 / 10000e18$) of shares tokens.

They will immediately lose $9999e18$ or half of their deposits if they `redeem()` right after the `deposit()`.

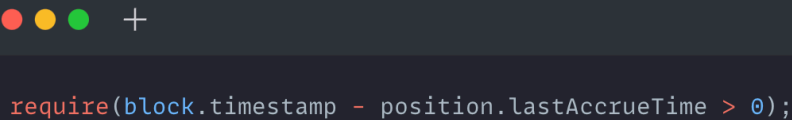
Consider requiring a minimal amount of share tokens to be deposited by the first depositor.

Medium severity

M.01. Malicious users can grief borrowers to prevent them from adding collateral.

Borrowers pay interest to the vaults by either calling `MarginManager accruePositionInterest` directly or when adding more collateral to an existing position.

When `accruePositionInterest` is called more than once in the same timestamp or block, it reverts because of the following check:



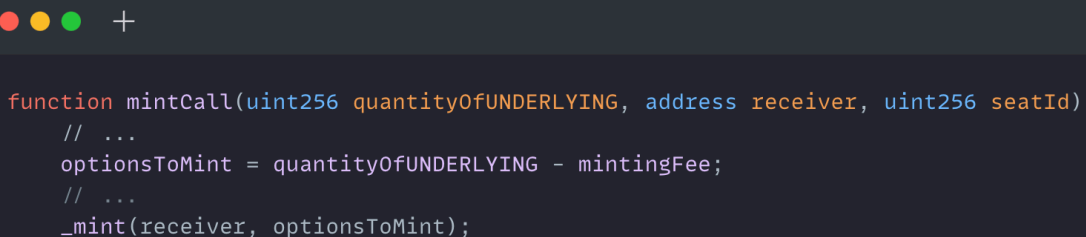
```
require(block.timestamp - position.lastAccrueTime > 0);
```

Since `accruePositionInterest` also has no access control, this can be abused by a malicious user to call `accruePositionInterest` in the same block in which a borrower tries to call `addCollateral`. Such a griefing attack can prevent borrowers from adding more collateral to their close-to-undercollateralized position and therefore allow the attacker to take their collateral.

Consider returning 0 instead of reverting if no time has passed since the last accrual.

M.02. Initial margin value is incorrectly computed for call options.

`PositionRouter.getInitialMargin` expects the options ERC20 contracts (which are out of scope for this audit) to have 18 decimal precision. Although this is indeed the case since all `AoriCall` and `AoriPut` options have 18 decimals, the units of call options minted are equal to the deposited collateral amount in `AoriCall.mintCall`:



```
function mintCall(uint256 quantityOfUNDERLYING, address receiver, uint256 seatId)
    // ...
    optionsToMint = quantityOfUNDERLYING - mintingFee;
    // ...
    _mint(receiver, optionsToMint);
```

Therefore, the `positionVal` for options where the collateral token has a low number of decimals will be significantly lower and may even round down to 0 in certain cases.

Consider fixing the issue in `AoriCall` (out of scope) by rescaling the collateral amount when converting it to options, as is done in `AoriPut`.

M.03. Token vaults and price oracles can be overwritten in the margin manager.

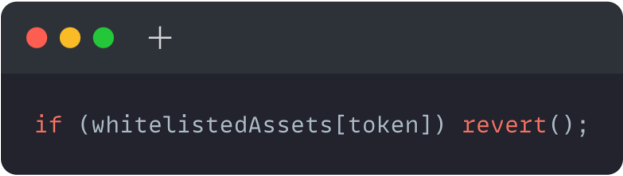
The `MarginManager` implements the following function to whitelist underlying tokens to be used as collateral for options which also attaches a vault as well as a Chainlink oracle to the added asset:



```
function whitelistAsset(  
    ERC20 token,  
    AggregatorV3Interface oracle  
) public onlyOwner returns (ERC20) {  
    whitelistedAssets[token] = true;  
    Vault lpToken = new Vault(...);  
    lpTokens[token] = lpToken;  
    oracles[token] = oracle;  
    return token;  
}
```

The problem is that the owner of the contract can abuse the above functionality to steal any collateral by manipulating the vault or price oracle address when `whitelistAsset` is called a second time.

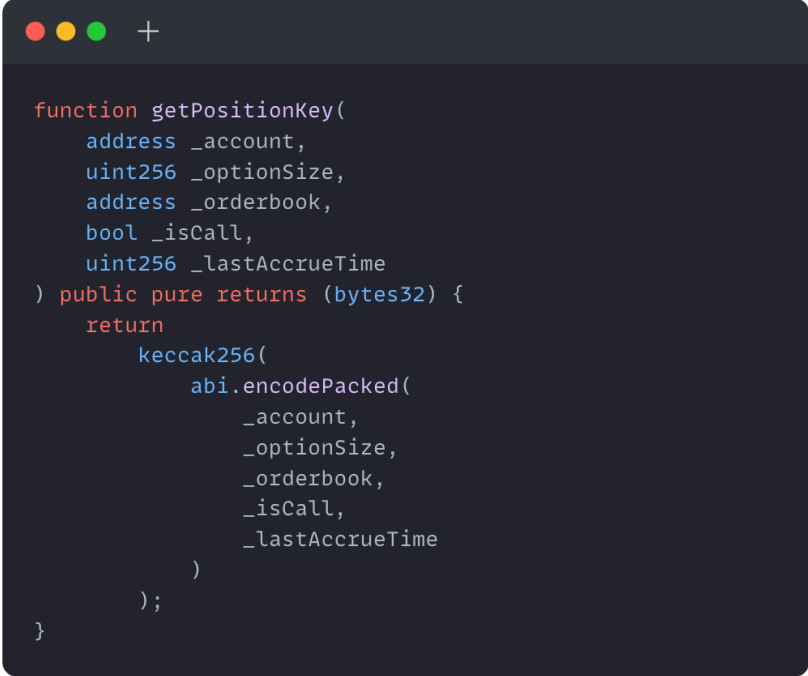
Consider allowing the owner to call `whitelistAsset` only once per asset:



```
if (whitelistedAssets[token]) revert();
```


M.04. User positions will be overwritten when executed in the same block.

When a borrower opens a position in the MarginManager, the position is stored in the positions mapping with a map key computed in the following way:



```
function getPositionKey(
    address _account,
    uint256 _optionSize,
    address _orderbook,
    bool _isCall,
    uint256 _lastAccrueTime
) public pure returns (bytes32) {
    return
        keccak256(
            abi.encodePacked(
                _account,
                _optionSize,
                _orderbook,
                _isCall,
                _lastAccrueTime
            )
        );
}
```

Since none of the above options are completely unique, if more than 1 user position requests are executed in the same block/timestamp and they contain the same parameters, all positions after the first one will be overwritten. As a result, the collateral for the previous ones will be lost or stuck.

Consider using a nonce or updating previous positions that are stored under the same mapping key in MarginManager.

M.05. Hardcoded USDC decimals assume the same precision across different blockchains.

In all contracts in scope (and some out of scope), the USDC precision/scale is a hardcoded constant with a value of 1e6. Although this value itself does not represent a problem since USDC is known to have 6 decimals on most blockchains, but since Aori aims to support multiple blockchains, this may result in denial of service if the protocol is someday deployed on BSC, where USDC has 18 decimals instead of 6.

Consider making the USDCScale variable immutable and setting it to USDC.decimals() in the contract constructors upon deployment.

M.06. The position router can be re-initialized by the contract owner.

Most of the state variables set in the `initialize` function can be changed after initialisation by the contract owner by calling the corresponding setter functions. However, there are 3 parameters that should not be able to change at any time during the contract lifecycle, and those are: `callFactory`, `putFactory`, and `manager`.

Giving the owner of the router contract the permission to change these parameters by calling `initialize` more than once allows them to steal any collateral from pending requests, empty any vault balance, cause denial of service to the whole margin system, etc.

Consider inheriting the `Initializable` contract from OpenZeppelin and applying the `initializer` modifier to prevent the owner from calling the above function more than once.

M.07. Rebasing and fee-on-transfer underlying tokens are not handled properly.

Some tokens take a transfer fee (e.g., STA, PAXG), while others do not currently charge a fee but may do so in the future (e.g., USDT).

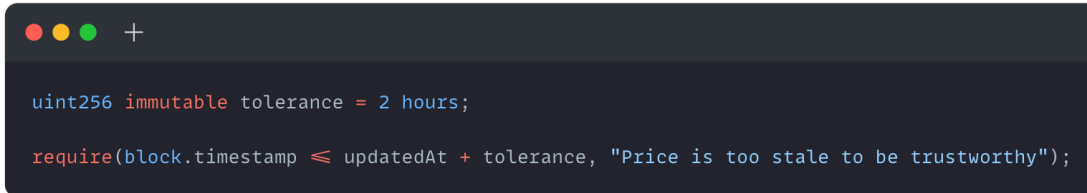
Additionally, certain tokens may make arbitrary balance modifications outside of transfers, such as Ampleforth-style rebasing tokens or Compound-style airdrops of governance tokens, as well as mintable/burnable tokens.

The contracts contain multiple areas that are not designed to handle rebasing tokens or tokens with a fee-on-transfer.

Consider avoiding the use of rebasing or fee-on-transfer tokens. If their usage is still desired, review and fix every instance where tokens are being transferred and adjust the accounting logic accordingly.

M.08. Chainlink oracle tolerance will exceed 2 hours for most price feeds.

Most Chainlink price oracles on Arbitrum (which is one of the supported networks for the Aori protocol) have a heartbeat of 24 hours. This can cause temporary denial-of-service if the price of the asset has not moved beyond the specified Chainlink deviation threshold, as `PositionRouter.getPrice` will revert when the price has not been updated for more than 2 hours.



```
uint256 immutable tolerance = 2 hours;  
require(block.timestamp ≤ updatedAt + tolerance, "Price is too stale to be trustworthy");
```

At the time of writing this report, the price of MATIC/USD has not been updated for more than 3 hours - <https://data.chain.link/arbitrum/mainnet/crypto-usd/matic-usd>.

Consider increasing the staleness tolerance or refactoring the staleness checks to address this issue.

M.09. Arbitrum sequencer should be checked if down.

Chainlink recommends that users utilising price oracles check whether the Arbitrum sequencer is active: <https://docs.chain.link/data-feeds#l2-sequencer-uptime-feeds>.

If the sequencer goes down, the indexed oracles may have stale prices, as L2-submitted transactions (i.e., those by the aggregating oracles) will not be processed.

Use the sequencer oracle to determine whether the sequencer is offline or not.

M.10. Inherited methods from ERC4626 can be used instead of the custom ones.

The Vault contract implements the `depositAssets` and `withdrawAssets` functions, which internally call the inherited `deposit` and `withdraw`, but also apply the `nonReentrant` modifier. In subsequent commits, additional logic has been added to `depositAssets` and `withdrawAssets`, such as a lock period.

Therefore, although it is expected for users to interact only with the new methods implemented in the Vault contract, since the inherited `deposit`, `withdraw`, `mint`, and `redeem` functions are public, users will be able to bypass any validations made in the custom functions.

Consider overriding the inherited methods from the ERC4626 contract and adding the custom logic there, or revert when any of them are called.

M.11. Missing input validation on privileged functions.

The Aori margin contracts rely heavily on centralized components due to the use of multiple privileged roles and insufficient input validation. Some of the responsibilities of privileged accounts include providing prices for options, setting asset price oracles, liquidating positions, and configuring various state variables used throughout the project flow.

Therefore, although implementing a more decentralized margin system would require further development rather than just adding validation checks for input parameters, it is recommended to ensure that setter functions, such as those in `MarginManager`, only accept values within a certain range. For example, introducing an upper bound for liquidation fees, specifying certain ranges for interest configuration, etc.

Consider improving input validation in setter functions and reducing the overall reliance on privileged accounts in the margin system.

M.12. Non-standard ERC20 tokens vulnerabilities.

A well-known issue regarding non-standard ERC20 tokens such as USDT and BNB that do not correctly implement the EIP20 interface, due to missing return boolean variables in methods like `transfer`, `transferFrom`, and `approve`.

Other tokens like ZRX return `false` instead of reverting when a transfer or approval fails. This behaviour would break many functionalities in the codebase.

Consider using the `SafeERC20` library from OpenZeppelin for all token transfers and approvals. Also, make sure to check for all known types of non-standard ERC20 token implementations in the list provided in the following repository - <https://github.com/d-xo/weird-erc20>.

Low severity

L.01. Missing access control for `settlePosition` and `addCollateral`

The `MarginManager` functions `settlePosition` and `addCollateral` currently lack proper access control. This permissionless behaviour has already been used in the previous findings to leverage other vulnerabilities, leading to more critical issues.

Although it might be intended to allow any address to settle and add collateral to any position, consider restricting these functions to only be accessible by the position owners.

L.02. Wrong function return values

The following functions return unexpected values, which do not have any impact on the contracts in scope since those values are not checked/used:

- `MarginManager.liquidatePosition` returns an empty position.
- `MarginManager accruePositionInterest` returns an incorrect value compared to the specification.
- `PositionRouter.getPrice` returns 0 when the price is 0, while reverting might be a more appropriate action.
- `Vault.settlePosition` is missing a return value so it always returns `false`.

L.03. `accruePositionInterest` should return instead of revert

`MarginManager.accruePositionInterest` reverts when the following condition is met:

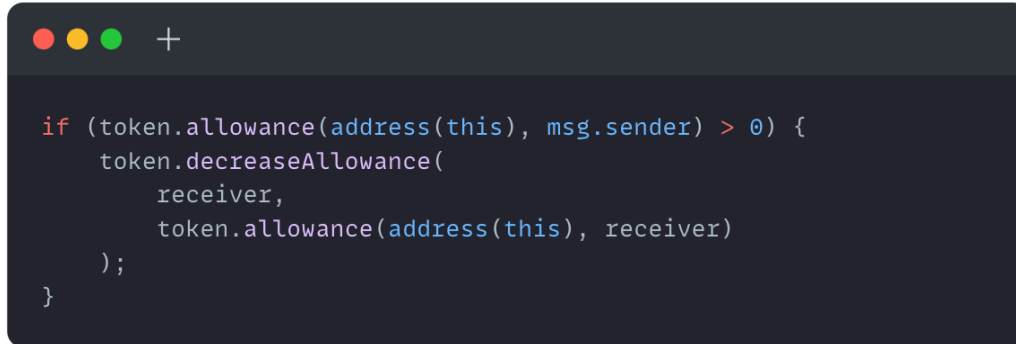
This can lead to unexpected behaviour, as shown in previous findings. A more appropriate way to handle a second call to this function in the same block for the same position would be to simply return 0.

L.04. Missing event

The `PositionUpdated` event is missing in `MarginManager.accruePositionInterest`, which could be needed since the function modifies the position's collateral and `lastAccrueTime` parameters.

L.05. Unreachable code

MarginManager.doTransferOut contains the following lines of code which do not appear to be used in any scenario:



```
if (token.allowance(address(this), msg.sender) > 0) {  
    token.decreaseAllowance(  
        receiver,  
        token.allowance(address(this), receiver)  
    );  
}
```

L.06. Not all ERC20 tokens implement decreaseAllowance

The decreaseAllowance function is a non-standard EIP20 function and therefore is not implemented by many ERC20 tokens. Consider using approve(0) instead of decreaseAllowance in MarginManager.doTransferOut.

L.07. Wrong secondPerYear value

In PositionRouter, secondPerYear is set to 31540000, while 365 days have 31536000 seconds, or a standard year has 31556926 seconds.

L.08. Misleading comment regarding configuration variable precision

A comment in PositionRouter states that the interest rate-related parameters should be in 4 decimals precision, while in the code, they are assumed to be in 18 decimals, and the tests also set them in 18 decimals precision. This may lead to incorrect configuration values being set at some point in time.

L.09. Unnecessary approvals

Unnecessary approval calls are made in PositionRouter.executeOpenPosition for the collateral amount to the MarginManager.

Consider either removing the approvals or using transferFrom in the MarginManager instead of transfer in PositionRouter.executeOpenPosition.

L.10. Round staleness check on price oracle

A stale Chainlink oracle price might be carried over from previous rounds. Consider checking whether answeredInRound is greater than or equal to the roundId.

Informational

I.01. Unused variables

The following global and local variables, arguments or properties are not used:

- `struct Position`, the `token` property is not used.
- `PositionRouter`, `liquidatorSeatId` is not used.
- `openShortPositionRequest`, `orderbook` should not be used; `option` is sufficient.
- `openShortPositionRequest`, `optionsToMint` is calculated but not used.
- `struct Vars`, `profitInUnderlying` is not used.
- `Vault`, `USDScale` is not used.

I.02. Redundant imports

The following imports are redundant:

- In `Struct.sol`, the `ERC20.sol` import is unnecessary.
- In `Vault.sol`, `Ownable.sol` is imported and inherited, but no `onlyOwner` functions are implemented.

I.03. Typographical mistakes

The `Vault` name and symbol in `MarginManager.whitelistAsset` seem to include a typo since there is no space between the custom string and the appended vault name.

The `settleVars` struct does not follow the `PascalCase` naming convention. The same issue is commented out in `Vault.sol`.

I.04. Wrong string revert message

The revert message in `accruePositionInterest` and `getInterestOwed` is not completely correct.

I.05. Public functions can be marked external

Multiple functions are marked `public` even though they are not called internally. Therefore, they could be marked as `external`.