Ithil v2 Security Review

Version 1.0.5

January 18, 2024

Conducted by:
Alin Mihai Barbatei (ABA), Independent Security Researcher

Table of Contents

1 About ABA

Alin Mihai BARBATEI, known as ABA, is an independent security researcher experienced in Solidity smart contract auditing. Having several solo and team smart contract security reviews, he consistently strives to provide top-quality security auditing services. He also serves as a smart contract auditor at Guardian Audits.

2 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 identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant 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 functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 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.

3.4 Informational findings

In addition to the above 4 severity types, **informational** findings will also be mentioned. These include recommendations, observations, improvements, and security best practices aimed at enhancing the codebase.

4 Executive summary

Overview

Project Name	Ithil v2
Codebase	https://github.com/Ithil-protocol/v2-core
Blockchain	Arbitrum
Language	Solidity
Initial commit	0ca9eb434a3dad3506de773bab6e81092f1dee0b
Remediation commit	3217f84015488782843bcf707bb1466b33360f64
Audit Methodology	Static analysis and manual review

Scope

Vault.sol
Manager.sol
Constants.sol
irmodels/AuctionRateModel.sol
services/Service.sol
services/CreditService.sol
services/DebitService.sol
services/Whitelisted.sol
services/BaseRiskModel.sol
services/credit/CallOption.sol
services/debit/GmxService.sol
services/debit/AaveService.sol
services/debit/AngleService.sol
services/debit/FraxlendService.sol

Issues Found

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	1	1	0	0
High risk	0	0	0	0
Medium risk	4	4	0	0
Low risk	12	12	0	0
Informational	13	9	0	4

5 Findings

5.1 Critical risk

5.1.1 Incorrect GMX service reward calculations leads to blocking position closing even on liquidations

Severity: *Critical risk*

Proof of concept: *PoC URL*

Context: *GmxService.sol:107*

Description

When a debit position from the GMX service, `GmxService` is closed, either for liquidation or closed by the owner, if there are not enough rewards accumulated in the system, the position can not be closed due to an underflow. The service reaches this point due to incorrect reward calculation.

Consider the action of opening and closing a position one after the other. This operation has no rewards or benefit for the protocol but the user would still receive rewards in that situation as it is implemented now.

`Opening`

```
        totalCollateral += agreement.collaterals[0].amount;
        virtualDeposit[id] = (agreement.collaterals[0].amount * (totalRewards +
            totalVirtualDeposits)) / totalCollateral;
        totalVirtualDeposits += virtualDeposit[id];
```

`Closing`

```
        uint256 finalBalance = weth.balanceOf(address(this));
        uint256 newRewards = totalRewards + (finalBalance - initialBalance);
        uint256 totalWithdraw = ((newRewards + totalVirtualDeposits) * agreement.
            collaterals[0].amount) / totalCollateral;
        uint256 toTransfer = totalWithdraw >= virtualDeposit[tokenID]
            ? totalWithdraw - virtualDeposit[tokenID] <= finalBalance
                ? totalWithdraw - virtualDeposit[tokenID]
                : finalBalance
            : 0;
        totalRewards = newRewards - toTransfer;
```

When closing, the `newRewards` for a position that was just opened is the `totalRewards` at open, since `finalBalance - initialBalance` is 0 in this scenario. The 2 calculations side by side, with a few replacements, are:

```
open:  virtualDeposit[id] = (agreement.collaterals[0].amount * (totalRewards +
    oldTotalVirtualDeposits)) / totalCollateral;
close:      totalWithdraw = (agreement.collaterals[0].amount * (totalRewards +
    oldTotalVirtualDeposits + virtualDeposit[id])) / totalCollateral;
```

Because `totalWithdraw` takes into consideration the update `totalVirtualDeposits`, which includes the virtual deposit, it will always be higher then the virtual deposit itself. This results in the `toTransfer` variable, the one to be subtracted from total rewards, to always be greater then zero.

When `toTransfer` is larger then `newRewards`, the subtraction on line L110 underflows as there are not enough rewards in the system to allow a position close. This is a severe issue because liquidations also cannot be closed.

```
        totalRewards = newRewards - toTransfer;
```

A malicious actor can simply created and close a position whenever there are any rewards accumulated, sometimes at a minor loss to himself, and continuously empty the contract of rewards, blocking positions from being closed.

Recommendation

Fix the reward calculation by subtracting the `virtualDeposit` from `totalVirtualDeposits` before calculating the withdrawable rewards when closing a position.

As a failsafe, another thing to take into consideration is adding an *allow partial rewards* logic for gained rewards (defaulted to false). Example if there are not enough rewards to be transferred for whatever unpredictable reason, `(toTransfer > newRewards)`, if `allowedPartialRewards` was provided (in the `data` payload for example), then transfer what it can transfer, otherwise revert.

Resolution: Resolved. The recommended fix was implemented in **8aebf14**, **583d187** and **2b715a6**.

5.2 Medium risk

5.2.1 Profits and losses calculations are applied retroactively on feeUnlockTime changes

Severity: *Medium risk*

Proof of concept: *PoC URL*

Context: *Vault.sol:60-67*

Description

When the `feeUnlockTime` variable from the `Vault` contract is updated by calling the `setFeeUnlockTime` function, the current locked profits and losses are not calculate beforehand with the old value before the change is applied.

Since the unlock time directly impacts the value of profits and losses, the next operation that calculates and stores the locked profits/losses (repaying or borrowing) will incorrectly calculate the values as if the entire time between the last borrow/repay up until now there was the current fee unlock time.

Depending if the new value for `feeUnlockTime` is greater or lower then the initial one, the protocol can incur losses.

The attached POC demonstrates how lowering the `feeUnlockTime` will result in less profits and less debt compared to when the time is profits/losses are correctly updated after the call to `setFeeUnlockTime`

Recommendation

In the `setFeeUnlockTime` function from the `Vault` contract, before updating `feeUnlockTime` with the new value, update current profits, losses and latest repay time.

Resolution: Resolved. The recommended fix was implemented in **d153944** and **4365678**.

5.2.2 Vault depositors can avoid having their liquidity blocked and cause call option owners loss when closing

Severity: *Medium risk*

Context: *CallOption.sol:191-198*

Description

When a call option is closed, if the owner does not receive his full due amount, the protocol borrows the difference from the vault and provides it to him. The borrowed amount is capped to the maximum free liquidity available in the vault at that time.

```
        if (toTransfer > transfered) {
            // Since this service is senior, we need to pay the user even if
               withdraw amount is too low
            // To do this, we take liquidity from the vault and register the loss
            // If we incur a loss and the freeLiquidity is not enough, we cannot
               make the exit fail
            // Otherwise we would have positions impossible to close: thus we
               withdraw what we can
            freeLiquidity = vault.freeLiquidity() - 1;
            toBorrow = toTransfer - transfered > freeLiquidity ? freeLiquidity :
               toTransfer - transfered;
        }
```

If closing a call option would block depositors from withdrawing their tokens due to lack of liquidity, a depositor can withdraw his balance before the option matures and add re-add it after the option was closed to avoid having his funds blocked.

This action would result in call option owners receiving less tokens then in a normal protocol flow.

Recommendation

Completely resolving this issue would require severe changes such as introducing a global withdraw window for vault depositors on top of liquidity availability.

It can be considered that options owners will do their due diligence and calculated when is the optimal time for them to close their position to minimize losses, considering they have a one month time window to close the position.

The above view is somewhat optimistic and would be better completed with the following mitigations:

1. Clearly document this issue. This is different from normal liquidity dependent risks that appear with borrowing and lending since users can be actively targeted by other users.

2. Provide a slippage value for how much the owner of a call option accepts to lose when closing his position due to liquidity issues. At one point the call option owner will have to accept to close his position with a loss regardless, since the window will close. The slippage value can be encoded in the `data` argument.

Adding the slippage variable would also save the owner when, by coincidence, within the same block a large vault withdrawal is done and it is prioritized by the sequencer before the his option close transaction.

Resolution: Resolved. The recommended fix was implemented in **996c73a**.

5.2.3 Attacker can gas grief position closing

Severity: *Medium risk*

Proof of concept: *PoC URL*

Context: *CallOption.sol AaveService.sol AngleService.sol FraxlendService.sol GmxService.sol*

Description

When opening either a debit or credit position, there is no validation that a user has not passed more then the required loan or collateral tokens. This can be abused to effectively add a tax for anyone closing the positions. For credit positions it may lead to a collateral "ransom" type of situation and for debit positions the added tax can be larger then the liquidation amount received by liquidators, effectively making liquidations non-profitable and as such leaving the protocol at risk for bad debt to accumulate.

When a call option is created, the agreement must contain 1 loan token and 2 collateral tokens. These tokens are the ones validated and used in the overall protocol logic.

```
    // This is a credit service with one extra token, Ithil
    // therefore, the collateral length is 2

    if (agreement.loans[0].token != address(underlying)) revert
        InvalidUnderlyingToken();
    if (agreement.collaterals[1].token != address(ithil)) revert InvalidIthilToken()
        ;
    if (agreement.loans[0].amount == 0) revert ZeroAmount();
```

However there is no validation that, in open agreement, a user has not passed more then 1 loan token or more then 2 collateral tokens. Since the entire call option logic works only with addresses that they validated, the extra token addresses are only saved to storage and emitted both on opening a call option (when the `PositionOpened` event is emitted) and on closing one (when the `PositionClosed` event is emitted). Both events emit the entire agreement:

```
    event PositionOpened(uint256 indexed id, address indexed user, Agreement
        agreement);
    event PositionClosed(uint256 indexed id, address indexed user, Agreement
        agreement);
```

Using the above lack of validation, an attacker can:

- open a call option with a large collateral token array.
- calculate so that the collateral he would be blocking is equal in value to the gas used to close the position
- when the option expires up to a point that anybody can close it, the closer would need to pay a very high gas in order to get roughly the same token value added back to the `CallOption` contract. Meaning having the ITHIL collateral tokens returned to the `totalAllocation` accounting so it can be used by other market participants.

Effectively an attacker, at a greater loss to himself, ca cause protocol/user losses by overpaying for closing a position.

We can calculated the approximate losses needed to inflict damage. Running the provided PoC with 4000 addresses as collateral we have the following gas consumption estimation

src/services/credit/CallOption.sol

Function Name	min	avg	median	max
close	18058825	18058825	18058825	18058825
open	45146785	45146785	45146785	45146785

We can roughly consider that for every 45 million gas spent by an opener, a closer would need 18 million to spend. This is a 2.5/1 ration, meaning for every 2.5 units of gas lost by the attacker, the closer must also lose 1 unit of gas. Considering a gas price of 0.1 GWEI on Arbitrum (nansen.ai), that would mean that an attacker has to pay 0.045 ETH to cause a 0.018 ETH gas cost of closing.

An attack would also need to have enough lending tokens as to be able to equivalent the collateral value to those levels, have those tokens vested for a minimum period, and when the position will be eventually closed, accept a risk that he may not receive his full loan token amount back. Realistically the attacker loss to protocol/user loss would probably be no lower then 3/1 with a vesting period for the attacker.

For an attacker that wishes to directly harm the protocol at whatever the cost, these conditions would still be acceptable.

A more easy and direct way of abusing this exploit is when creating debit positions. By applying the same rational (again with 4000 empty collateral address), we see that for opening and closing a GMX service position there is 1.7/1 ratio of how much an attacker needs to lose in order to inflict losses to the protocol.

src/services/debit/GmxService.sol

Function Name	min	avg	median	max	# calls
close	27301572	27301572	27301572	27301572	1
open	46502419	46502419	46502419	46502419	1

Example, an attacker needs to lose 0.0465 ETH on opening a debit position to make the closer lose 0.027 ETH on each close. If the position margin that is received by the liquidator is less then 0.027 ETH in value, then the position in effect becomes unprofitable to close and will either:

- not be closed by any 3rd party
- be closed at a loss by the protocol team (liquidator)

Recommendation

Each service, regardless if credit or debit, must specifically verify that the number of collateral and loan tokens in the agreement is exact the number they expect to work with.

Resolution: Resolved. The recommended fix was implemented in **942095c**.

5.2.4 Debit position loss calculation may lead to less fees

Severity: *Medium risk*

Context: *DebitService.sol:139*

Description

Closing a debit position at a loss is reserved only for the trusted liquidator role. With the current implementation, there is an issue in how the loss is calculated:

```
    if (
        obtained[index] < agreement.loans[index].amount && msg.sender != liquidator
            && liquidator != address(0)
    ) {
        revert LossByArbitraryAddress();
    }
```

The total rewards obtained from closing the position is compared to the amount that was borrowed from the vault and if not enough is considered a loss. This is incomplete because it does not take into consideration the due fees for having kept the position opened.

Consider the following situation:

- a position becomes close to be considered at a loss (the obtained amounts would be less then the loaned amount)
- for all the time the position has been opened, a due fee has been accumulating
- user waits until the final moment and closes his position, when obtained rewards are only slightly above the borrowed amount (if he is not liquidated before-hand)
- in this case, the entire obtained rewards are returned to the vault, since the difference would be less then the due fee plus loan amount (user gains nothing)
- but because the user had the opportunity to wait until his position rewards came close to the loaned amount (hoping for it not to be liquidated) the protocol receives less fees that entitled.

If the loss would of been calculated while taking into consideration the fees, user would have needed to close his position earlier, otherwise risk not being able to close his position at all since it would of been at loss. In that scenario, the protocol would gain the full fees. Also, if the position does become a loss, the liquidation would accrue faster due to the liquidator being a team controlled entity that activates rapidly, resulting in more fees being paid to vault holders.

From the moment the rewards to-be-gained are less then the borrowed amount plus fees, users have no incentive to close their position since all tokens would go to the vault. In fact, users have an incentive to not close it hoping that it will become profitable. This behavior results in less fees being paid overall.

Recommendation

Modify the loss verification to also take into consideration the due fees, along side the agreement loan amount.

Resolution: Resolved. The recommended fix was implemented in **bb19833**.

5.3 Low risk

5.3.1 Inadequate input validations in CreditService.open allows passing user controlled adresses into the system

Severity: *Low risk*

Context: *CreditService.sol:19-23*

Description

When opening an agreement for a crediting services, the function `open` from the `CreditService` contract is used as entry point. This functions attempts to validates the provided agreement input has valid vault and collateral tokens, but does so incorrectly.

```
    address vaultAddress = manager.vaults(agreement.loans[index].token);
    if (
        agreement.collaterals[index].itemType != ItemType.ERC20 ||
        agreement.collaterals[index].token != vaultAddress
    ) revert InvalidInput();
```

The issue is that for any address which is not tracked by the manager, the `vaultAddress` will be defaulted to `address(0)` which then is checked against the `agreement.collaterals[index].token` which again can be set as `address(0)` by the caller. Also the `agreement.collaterals[index].itemType` is a user controlled input which is compared to a `ItemType.ERC20` element, the first in its enumeration, which is 0 as default.

Because of the way the above checks are done, a malicious actor can:

- set `agreement.loans[index].token` a contract token he created
- create an empty array and set it as `agreement.collaterals`
- when the code is executed, `vaultAddress` becomes `address(0)`, and since collaterals are empty arrays, they default to `0`/`address(0)` and the above check passes

With the check passed, an attacker can then have an external call within the `CreditService.open` as the next code to be executed is a `IERC20(agreement.loans[index].token).safeTransferFrom` command.

As the codebase is currently, the above issue is of low criticality because collateral, vault and loan token addresses are further validated in all services that extend the CreditService contract (both FixedYieldService and CallOption contracts) and result in execution termination if non-compliant. Also the external call foothold that the malicious party has from the safeTransferFrom function call in CreditService.open does not open the contract to any attack.

Recommendation

Validate that vaultAddress is different from address(0) in the open function from the CreditService contract when validating it.

Consider adding a NOT_SET initial first entry in the ItemType enumeration from IService in order to block all possible future default value abuse scenarios.

Resolution: Resolved. The recommended fix was implemented in **68d4b60**.

5.3.2 Users WETH rewards estimates can be incorrect

Severity: *Low risk*

Context: *GmxService.sol:116-123*

Description

The GmxService contract allows users to check the amount of rewards they would receive if closing their position by calling the function wethReward. This function does not take into consideration specific corner cases regarding available liquidity WETH liquidity that in some cases would result in receiving less tokens than indicated.

```
    function wethReward(uint256 tokenID) public view returns (uint256) {
        uint256 collateral = agreements[tokenID].collaterals[0].amount;
        uint256 newRewards = totalRewards + rewardTracker.claimableReward(address(
            this));
        // calculate share of rewards to give to the user
        uint256 totalWithdraw = ((newRewards + totalVirtualDeposits) * collateral) /
             totalCollateral;
        // Subtracting the virtual deposit we get the weth part: this is the weth
            the user is entitled to
        return totalWithdraw - virtualDeposit[tokenID];
    }
```

Specifically:

- function reverts on cases where rewards would be 0 (when virtualDeposit > totalWithdraw)
- does not take into consideration WETH and reward availability, similar to how the rewards are actually calculated in the _close function

```
        uint256 finalBalance = weth.balanceOf(address(this));
        uint256 newRewards = totalRewards + (finalBalance - initialBalance);
        // calculate share of rewards to give to the user
        uint256 totalWithdraw = ((newRewards + totalVirtualDeposits) * agreement.
            collaterals[0].amount) /
            totalCollateral;
        // Subtracting the virtual deposit we get the weth part: this is the weth
            the user is entitled to
        // Due to integer arithmetic, we may get underflow if we do not make checks
        uint256 toTransfer = totalWithdraw >= virtualDeposit[tokenID]
            ? totalWithdraw - virtualDeposit[tokenID] <= finalBalance
                ? totalWithdraw - virtualDeposit[tokenID]
                : finalBalance
            : 0;
```

Recommendation

Modify the `wethReward` function for it to perfectly imitate the reward calculation done when the position is closed.

Resolution: Resolved. The recommended fix was implemented in **f9ca7f8**.

5.3.3 Missing onlyWhitelisted on GMX Service _open function

Severity: *Low risk*

Context: *GmxService.sol:59*

Description

The pattern for every debit service implemented is to guard opening a position only by whitelisted addresses if this is desired. The `_open` function from the `GmxService` contract is missing the `onlyWhitelisted` modifier.

Recommendation

Add the `onlyWhitelisted` modifier to the `_open` function.

Resolution: Resolved. The recommended fix was implemented in **cdd3555**.

5.3.4 PositionOpened event emitted with incorrect id

Severity: *Low risk*

Context: *Service.sol:98*

Description

After a position is opened using the Service contract, at the end, an `PositionOpened` event is emitted with the currently opened position. This event is incorrectly emitted with the next token ID value, not the current one, since the `id++` is executed after the `_safeMint` function call in the `[Service.open](http://Service.open)` function.

This causes 3rd party event-consumers to mistakenly consider different positions as being opened.

Recommendation

Emit the `PositionOpened` with id value before incrementing it.

Resolution: Resolved. The recommended fix was implemented in **bd42410**.

5.3.5 Important contract changes do not emit events

Severity: *Low risk*

Context: *Global*

Description

There are several instances where important contract changes are done but events are not emitted:

- `DebitService`
  - `setMinMargin` when the minimum margin for a token is set
  - `setLiquidator` when a liquidator is set (or removed)
- `CallOption`
  - `allocateIthil` when ITHIL tokens are allocated for call options
  - `sweepIthil` when ITHIL tokens are removed from the `CallOption` contract
- `AuctionRateModel`
  - `setRiskParams` when setting risk spread, halving time and others

Recommendation

Emit events when the indicated operations are executed.

Resolution: Resolved. The recommended fix was implemented in **6628e47**.

5.3.6 Add input validation on every possible argument

Severity: *Low risk*

Context: *Global*

Description

Not checking the parameters provided in a call can lead to severe issues. Some have already been discussed separately in this report, where they lead to a more critical severity impact. For the others indicated here, of lower severity, add proper validations:

- `Vault.sweep`:
  - validate that `to` is not `address(0)`
- `Manager.sweep`
  - validate that `to` is not `address(0)`
- `Service`
  - `constructor`
    * validate that `_manager` is not `address(0)`
  - `_saveAgreement`

    * validate that the length of `agreement.loans` and `agreement.collaterals` have a max cap. There is no logic in allowing an infinite number of loans/collaterals.
    * consider a limit on how many agreements one user can have, and limit to it

- `Whitelisted`
  - for both `removeFromWhitelist` and `addToWhitelist` a check for duplication in the provided `users` array can be done. This however may not be worth it from a gas usage point of view. It is up to the discretion of the protocol team if it is needed.
- `DebitService.setMinMargin`
  - validate that `token` is not `address(0)`
- `GmxService.constructor`
  - validate that `_manager`, `_router` and `_routerV2` are not `address(0)`
- `FraxlendService.constructor`
  - validate that `_manager` and `_fraxLend` are not `address(0)`
- `AngleService.constructor`
  - validate that `_manager` and `_steur` are not `address(0)`
- `AaveService.constructor`
  - validate that `_manager` and `_aave` are not `address(0)`
- for all debit services, in the constructor, validate that `_deadline` is greater then 0 if this has sense to the project. A minimum and a maximum value can also be taken into consideration.
- `CallOption.constructor`
  - validate that `_manager`, `_ithil` and `_underlying` are not `address(0)`
  - validated that `_minLoan` is greater then 0. Setting it to 0 allows for the Dutch auction spam attack
  - change the `assert` into a `require`, asserts are usually used for checking invariants or inner states, not input validation
  - for `_halvingTime`, `_tenorDuration` and `_initialVesting`, consider if there is any use cases where these would be 0, and if not guard against it. Also consider maximum/minimum interval checks.
  - on line 81, check that the `_vaultAddress` variable from the operation `_vaultAddress = manager.vaults(_underlying);` is not `address(0)`. If the underlying token was not added to the manager/vaults, the resulting vault address is `address(0)`

Recommendation

Perform the indicated changes.

Resolution: Resolved. The recommended fix was implemented in **10afc88** and **31e5436**.

5.3.7 Incorrect validation for max-base surpassed

Severity: *Low risk*

Context: *AuctionRateModel.sol:75*

Description

When updating the new base and latest borrow variables in the `_updateBase` function of the `AuctionRateModel` contract, a validation is done to ensure that the new base does not surpass the maximum allowed value:

```
if (newBase + spread >= 1e18) revert InterestRateOverflow();
```

This check is however incorrect because it does not allow for equality. The new base plus spread can touch the maximum value (1e18).

Recommendation

Change the >= comparator from the if statement to >.

Resolution: Resolved. The recommended fix was implemented in **60a72bf**.

5.3.8 Incorrect quote price affects liquidation score

Severity: *Low risk*

Context: *GmxService.sol:125-144*

Description

The `GmxService` contract provides a `quote` function that will be used to determine liquidation score. The quote value is incorrectly calculated when determining the liquidation score for opened debit orders.

- when getting the assets under management in `USDG` equivalent the call to the `getAumInUsdg` function is with the `false` argument. This is incorrect and will use a lower price to estimate AUM value then when minting which sets it to `true`. (onchain GlpManager.sol:213).

```
    uint256 aumInUsdg = glpManager.getAumInUsdg(false);
```

- the current implementation labels the token redeemable amount as the `usdgDelta` and uses it to calculate the fee basis points. This is incorrect as the second argument for the `getFeeBasisPoints` function must be the same `usdgAmount` value also used in the `getRedemptionAmount` function. (see onchain USDG Vault.sol:1216) since one represents the amount of tokens received and the other the USDG equivalent.

```
    uint256 usdgDelta = usdgVault.getRedemptionAmount(agreement.loans[0].token,
        usdgAmount);

    uint256 feeBasisPoints = usdgVault.getFeeBasisPoints(
        agreement.loans[0].token,
        usdgDelta,
        usdgVault.swapFeeBasisPoints(),
        usdgVault.taxBasisPoints(),
        false
    );
```

The above errors result in a different fee basis point to be calculate then expected. The overall result is a slightly different quote value which is used for liquidity score calculation.

Recommendation

Get the AUM using the maximum price and use the correct token amount for calculating the fee.

Resolution: Resolved. The recommended fix was implemented in **5e9eb34** and **7e7aa20**.

5.3.9 Attacker can continuously negate Dutch auction price discount for opening positions

Severity: *Low risk*

Proof of concept: *PoC URL*

Context: *CallOption.sol:205-210*

Description

Call options open price is composed of an initial price plus the latest spread value which is decreased by a Dutch auction model type logic. It is calculated by the `_currentPrice` function from the `CallOption` contract:

```
    function _currentPrice() internal view returns (uint256) {
        return
            block.timestamp < 2 * halvingTime + latestOpen
                ? initialPrice + (latestSpread * (2 * halvingTime + latestOpen -
                  block.timestamp)) / (2 * halvingTime)
                : initialPrice;
    }
```

- `halvingTime` and `initialPrice` are constants and chosen when the `CallOption` contract is deployed
- `latestSpread` is calculated as a function of price and the remaining allocation in inverse proportionality
- `latestOpen` is the last time a call option has been opened

By observing the formula, we see that when `latestOpen` is as close to `block.timestamp` as possible, the effective total price becomes `initialPrice + latestSpread` meaning that the Dutch auction type incremental price lowering is not applied. This mechanism was created to favor options buys the longer the time between 2 opened options is.

An attacker can completely negate the price reduction by continuously opening call options with 1 WEI. The cost for the attacker is insignificant and he would need only one transaction per block, or every few blocks, to keep price capped to the latest spread. This attack results in users paying more then normal for call creating options.

The issue is not present with debit positions because of the minimum margin requirement, making the attack severely costly for an attacker.

Recommendation

Add a minimum loan amount when opening any position, either credit or debit. Currently only that the amount is greater then 0 is validated, but this does not suffice.

Resolution: Resolved. The recommended fix was implemented in **8ae503e** and **63b76bd**.

5.3.10 Vault is not ERC4626 compliant

Severity: *Low risk*

Context: *Vault.sol:191,167*

Description

There are several issues with the current vault implementation that result in it being non-compliant with the ERC4626 standard.

The `deposit` and `mint` functions are guarded by an `unlocked` modifier that reverts when vault is locked. Because of this, the following functions are non-compliant:

- `maxDeposit`

  > MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.

  - it is not overwritten, a locked vault should return 0 but `maxDeposit` returns the normally calculated value

- `maxMint`

  > MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0.

  - it is not overwritten, a locked vault should return 0 but `maxMint` returns the normally calculated value

The `withdraw` and `redeem` functions both validate incorrectly that the assets amount to receive does not exceed or is exactly equal to the available liquidity

```
        if (assets >= freeLiq) revert InsufficientLiquidity();
```

- the if clause: `assets >= freeLiq` should not include equality since:
  - if the amount is equal to free liquidity then it should be allowed in both cases
  - `maxWithdraw` and `maxRedeem` consider the total free liquidity as valid to be received, thus the current logic also breaks EIP compliance for the `maxWithdraw` and `maxRedeem` functions

  > MUST return the maximum amount of assets that could be transferred from owner through withdraw and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

> MUST return the maximum amount of shares that could be transferred from owner through redeem and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

The `previewWithdraw` and `previewRedeem` functions need to be overloaded and modified to take into consideration the free liquidity, net loans and locked losses as part of the calculation. This is to respect:

- `previewWithdraw`

  > MUST return as close to and no fewer than the exact amount of Vault shares that would be burned in a withdraw call in the same transaction

- `previewRedeem`

  > MUST return as close to and no more than the exact amount of assets that would be withdrawn in a redeem call in the same transaction

Recommendation

Overwrite the `maxDeposit` and `maxMint` functions and change them to return 0 when the vault is locked.

Modify `withdraw` and `redeem` so that the available free liquidity check the does not take into consideration equality.

Overwrite `previewWithdraw` and `previewRedeem` with the correct behavior.

Resolution: Resolved. The recommended fix was implemented in **c61ff68**, **602e6f8**, **689bd72**, **c917594**, **eeabbb9** and **d995e87**.

5.3.11 MAX_RATE can be surpassed

Severity: *Low risk*

Context: *AuctionRateModel.sol:31*

Description

In the `AuctionRateModel` contract, the base rate and risk spread can be set to more the the `MAX_RATE` by calling the `setRiskParams` function which checks that individually the values are not larger that `MAX_RATE` but not their sum.

```
    if (token == address(0) || baseRate > MAX_RATE || riskSpread > MAX_RATE ||
        halfTime == 0) revert InvalidInitParams();
```

Recommendation

Add a check that the new base rate and spread do not surpass `MAX_RATE` when setting it by the owner.

Resolution: Resolved. The recommended fix was implemented in **41b10c8**.

5.3.12 First depositor attack protection may be insufficient

Severity: *Low risk*

Context: *Manager.sol:45-46*

Description

The current first depositor attack mitigation relies on the `Manager` contract deploying the vault and instantly depositing 1 WEI of underlying assets. The manager contract, which acts as a factory contract, does not have the means to withdraw the 1 WEI share equivalent that was received upon the first deposit.

An important observation to add, is that for the `Mangaer` contract, once a vault has been deployed, it cannot be changed to a different vault.

Current mitigation leaves the contract vulnerable to a second depositor attack, which if successful, even at a **severe loss to the attacker**, would cause loss of funds to the users of the protocol or the protocol being needed to redeploy.

Consider the following scenario:

- manager deploys vault and does the initial 1 WEI deposit:

```
Vault:
 totalAssets = 1
 totalShares = 1

manager:
 shares = 1
```

- malicious attacker, `bob`, deposits 1 WEI also

```
Vault:
 totalAssets = 2
 totalShares = 2

manager:
 shares = 1

bob
 shares = 1
```

- `bob` sends 1 ether directly to vault

```
Vault:
 totalAssets = 2 + 1 ether
 totalShares = 2

manager:
 shares = 1

bob
 shares = 1
```

- at this point, if the manager could withdraw his 1 share the attacker is at a direct loss of 50%
- withdrawals and deposits can be blocked by the team at this point and the loss would be 100% for the attacker but the project would need to redeploy, which it can only if there are no positions opened
- if the attack would continue, say `alice` deposits 0.5 ether into the vault
    - gets 0 shares because:

```
assets.mulDiv(supply, totalAssets(), roundingDown);
= 0.5 ether * 2 / (2 + 1 ether)
= 1 ether / (2 + 1 ether)
= 0
```

at this point the vault has:

```
Vault:
 totalAssets = 2 + 1.5 ether
 totalShares = 2

manager:
 shares = 1

bob
 shares = 1

alince
 shares = 0
```

- if `bob` withdraws 1 share he gets `0.75 ether`, a 25% loss
- by this logic, if someone deposits another 0.5 ether then `bob` will not be at a loss, will be break-even
- but if someone deposits enough to not round down, say 2 ether, `bob` and the new depositor, would be at a loss if withdrawn, making the attack even more unprofitable but still adding the user loss

The project will be deployed on Arbitrum making front-running not feasible at this point in time. Thus it is not possible for an attacker to see pending transactions, he would blindly need to risk losing a large amount of liquidity, at least double what the next deposit would be. The attacker also does not know that amount so he would require risking, forcing him to use an amount based on presumptions.

The preconditions for this attack are heavily against the attacker but if he is willing to take **a severe lose of** tokens he can harm the depositors after himself.

Recommendation

Modify the initial deposit value into the vault from 1 WEI to at least 1000 WEI, virtual shares usually start at 1e6 but 1000 WEI is an effective value as well.

Resolution: Resolved. The recommended fix was implemented in **36f7dc6**.

5.4 Informational findings

5.4.1 Not all supported ERC20 tokens implement the EIP-2612 standard

Severity: *Informational*

Context: *Vault.sol:144-155*

Description

Ithil intends to initially support a specific subset of tokens and subsequently add others as they are needed. One of the tokens that is intended to be supported is `AURA` which does not support approvals to be made via signatures (Permit).

Vaults constructed with the `AURA` token as the asset will revert when calling the function `depositWithPermit` is called.

It is also important to mention that some tokens from other chains do not completely follow the EIP-2612 standard. A classical example of this is the DAI token on Ethereum. Since the project will be deployed only an Arbitrum (where DAI, a supported token, respects the standard) this is not an issue.

Recommendation

There should be clear documentation that indicates that not all protocol supported tokens, `AURA` for example, will also support the `depositWithPermit` functionality.

Resolution: Acknowledged by the team.

5.4.2 Use built-in time units when possible

Severity: *Informational*

Context: *Global*

Description

Throughout the codebase there are uses of time periods being written directly as seconds. For a better code visibility these can be changed with in-built units:

- `Constants.sol#L5`
  - `uint256 constant ONE_YEAR = 31536000;` -> `uint256 constant ONE_YEAR = 365 days;`
- `Vault.sol#L37`
  - `feeUnlockTime = 21600; //six hours` -> `feeUnlockTime = 6 hours;`
- `CallOption.sol#L69`
  - `13 * 30 * 86400` -> `13 * 30 days`

Recommendation

Use the available time units and replace where it is appropriate.

Resolution: Resolved. The recommended fix was implemented in **c071ef7**.

5.4.3 Remove unused imports

Severity: *Informational*

Context: *AuctionRateModel.sol:6 Service.sol:8*

Description

There are unused imports in the codebase:

- `import { DebitService } from "../services/DebitService.sol";` from `AuctionRateModel` contract
- `import { Vault } from "../Vault.sol";` from `Service` contract

Recommendation

Remove them unused imports.

Resolution: Resolved. The recommended fix was implemented in **490a029**.

5.4.4 Misleading function name

Severity: *Informational*

Context: *Service.sol:63-67*

Description

The function `toggleLock` from the `Service` contract does not toggle the lock variable, it sets it:

```
function toggleLock(bool _locked) external onlyGuardian {
    locked = _locked;

    emit LockWasToggled(locked);
}
```

Recommendation

Either change the implementation of the `toggleLock` to actually toggle the `locked` storage variable or rename it to `setLock`.

Resolution: Resolved. The recommended fix was implemented in **63465ac**.

5.4.5 Add natspec to codebase

Severity: *Informational*

Context: *Global*

Description

Natspec and proper documentation helps any extending developer as well as future auditors to better understand the codebase. Project includes inline comments for a part of the more critical operations but does not include natspec for the majority of functions.

Recommendation

At a minimum, add natspec for every function that is callable from the exterior. Although it would be better to add to every function there is. Also, there are several instances when comments were written to document functions but with only 2 backslashes `//`. Convert these into natspec comments.

Resolution: Resolved. The recommended fix was implemented in **7489bdb** and **db4d7e1**.

5.4.6 Use a 2 step ownership transfer routine

Severity: *Informational*

Context: *Global*

Description When transferring ownership of a contract, mistakes that transfer the ownership to an unwarned address can be avoided by using a 2 step transfer routine. In the first step a new owner is proposed and in the second step the new owner must accept the ownership.

Recommendation

For the following contracts:

- `AuctionRateModel`
- `Manager`
- `Service`
- `Whitelisted`

use OpenZeppelin's 2 step ownership transfer contract

Resolution: Acknowledged by the team.

5.4.7 Disable ownership renouncing

Severity: *Informational*

Context: *Global*

Description There are several key contracts that have an owner:

- `AuctionRateModel`
- `Manager`
- `Service`
- `Whitelisted`

It is possibly, by mistake, to renounce ownership of the deployed underlying contract, leaving key critical operations blocked.

Recommendation

Consider overloading the `Ownable2Step/Ownable.renounceOwnership` method and making it revert on call.

Resolution: Acknowledged by the team.

5.4.8 Use constants where appropriate

Severity: *Informational*

Context: *Global*

Description

For better code visibility, use meaningful constants where applicable. Instances where these can be used in the current codebase with suggestions:

- `Vault.sol:L63`
  - `30 seconds` → `MINIMUM_FEE_UNLOCK_TIME`
  - `7 days` → `MAXIMUM_FEE_UNLOCK_TIME`
- `GmxService.sol:142`
  - `10000` → `BASIS_POINTS_DIVISOR`
- `CallOption.sol:166,171,221`
  - `1e18` → `FULL_PORTION`
- `AuctionRateModel.sol:28,75`
  - `1e18` → `MAX_RATE`

Recommendation

Apply the indicated changes.

Resolution: Resolved. The recommended fix was implemented in **0557a51**, **2efa09e** and **2ed6cdb**.

5.4.9 Do not transfer 0 amount tokens

Severity: *Informational*

Context: *Vault.sol:218,253 GmxService.sol:113 CallOption.sol:200 DebitService.sol:153*

Description

For the `borrow` and `repay` functions from the `Vault` contract, an asset amount of 0 is allowed. When passed, it results in a 0 transfer being done.

When closing a GMX service position in the `_close` function of the `GmxService` contract, the `toTransfer` variable can be 0, at which point the token transfer at the end of the function will transfer 0 tokens.

When closing a credit Call option position, in the `_close` function of the `CallOption` contract, the `toCall` variable can be 0, resulting in a 0 token transfer on L200.

When a debit position is closed with no rewards left for the user, after repaying the vault, a token transfer with a 0 amount is initiated. This is redundant and wastes gas. Since liquidations have a high chance or reaching this point, the gas overhead of adding an amount check before doing the transfer is outweigh by the gains of not doing 0 transfers.

Recommendation

For both functions from the `Vault` contract, check if the `assets` argument is not 0 before initiating the transfer. In the `_close` function from the `GmxService` contract, check that `toTransfer` is not 0 before initiating the WETH transfer. In the `_close` function from the `CallOption` contract, check that `toCall` is not 0 before initiating the transfer.

Add a check that `obtained[index]` is greater then `repaidAmount` before initiating the `safeTransfer` to the `agreementOwner` in the `DebitService`.

Resolution: Resolved. The recommended fix was implemented in **0557a51**, **2efa09e**, **2ed6cdb** and **ff4fa23**.

5.4.10 Remove outdated comments

Severity: *Informational*

Context: *DebitService.sol:77-80, GmxService.sol:72*

Description There are several instances of outdated comments that at time are misleading:

- from the `GmxService` contract the following: `//This check is here to protect the msg.sender from slippage, therefore reentrancy is not an issue`
- in the `DebitService` contract:

```
// No need to launch borrow if amount is zero
uint256 freeLiquidity;
// this call does not constitute reentrancy, since transferring additional
    margin
// has the same effect as opening a single position with the sum of the two
    margins
```

Recommendation

Remove the indicated comments or replace them with relevant comments.

Resolution: Resolved. The recommended fix was implemented in **026fef0**.

5.4.11 Missing USD value slippage option for GLP purchases

Severity: *Informational*

Context: *GmxService.sol:67*

Description

When a position is opened using the `GmxService` contract, GMX's GLP token is bought. The purchase is missing one of the slippage value check, specifically the the minimum acceptable USD value of the GLP purchased, `_minUsdg`.

Users can still guard themselves using the minimum acceptable GLP slippage amount variable, `_minGlp`, which is used by the protocol.

Recommendation

Add the possibility to add minUSDG slippage when opening GMX debit positions.

Resolution: Resolved. The recommended fix was implemented in **a4cbbc3**.

5.4.12 Use an updated version of OpenZeppelin library

Severity: *Informational*

Context: *Global*

Description

The project uses the OpenZeppelin libraries version 4.8.0 which is outdated. Although that specific version does have know bugs, none directly affect the project.

Recommendation

Updated to a more recent version while considering any breaking changes that are introduced.

Resolution: Acknowledged by the team.

5.4.13 Add security contact to contracts

Severity: *Informational*

Context: *Global*

Description

It is recommended to add a `custom:security-contact` natspec tag to the contracts, with the email address for the team security contact in case a whitehat identifies an issue with the on-chain contracts of the protocol to more easily be able to contact them.

Recommendation

Add the indicated information.

Resolution: Resolved. The recommended fix was implemented in **7e35cb6**.