Rage Trade

Audit Report

Prepared by Christoph Michel
April 11, 2022.

Contents

1 Introduction

Rage Trade is a perpetual swaps protocol enabling traders to take leverage on crypto assets. The documentation is available here.

> The documentation has been revised following the audit recommendations.

1.1 Scope of Work

The auditors were provided with a GitHub repository at commit hash 21324c8 (Jan 17th 2021). Auditing the fixes was done on the commits mentioned here (Feb 28th 2021). There has *not* been a *code freeze* for this commit, the contracts are still in development and might not represent any future contracts deployed on-chain.

The task was to audit the contracts, consisting of the following files with their `sha1` hashes:

File	SHA1
`oracles/BaseOracle.sol`	a74e55dc0ee6a7643391948c4d0a69b5575a91c9
`oracles/ChainlinkOracle.sol`	4ca894b94a5aace7d1d2328fb7de22836dded1a9
`libraries/SignedFullMath.sol`	6aaba992dec84bffcf83fc503a2dfd2d519d22f7
`libraries/VTokenPosition.sol`	a34185b91bf64a4e564f8a52b9ecb3bb4314c648
`libraries/Uint32L8Set.sol`	8442cf553ac7518c35ed62b4add1ec03a93f96fa
`libraries/LiquidityPositionSet.sol`	ec5ac3621b0a3b37c19c508767276fc042373f88
`libraries/Tick.sol`	d54cdb2f0d5645b788330a21b1da815cb45e761a
`libraries/VTokenLib.sol`	6832246d40caa49403562e9d707043f2a85d3a65
`libraries/DepositTokenSet.sol`	2011bd4f2b80abf1f62d727dcbeec983936947c2
`libraries/Account.sol`	dcba22cb5c74b209199e6530bb5ade06643cccc4
`libraries/FundingPayment.sol`	90f2ec509063462ebcd273cd02e6b0174db40ab8
`libraries/SignedMath.sol`	229d5d9577fe57aea74b1929d2703abb889603e7

File	SHA1
libraries/Uint32L8Array.sol	f093453f50f0b512f00e4d23fe9ec7ae228a79e9
libraries/Uint48.sol	fa77b7756cd223de36517fb55bc5576aff797466
libraries/UniswapV3PoolHelper.sol	6133638550b193de014ef0a1219e352cb3fbe85e
libraries/PriceMath.sol	cd26df739497a4202587172ec3dfb518ded17778
libraries/TickBitmapExtended.sol	57239a0d1c88dbc78cf484fec99576737152f79c
libraries/LiquidityPosition.sol	331f28c6b9ea7042911aa88a683666f0822a39f3
libraries/Calldata.sol	96a762487408ee0184c53dff27a72e5c052d2f5a
libraries/Arbitrum.sol	88385d93a6f2fade174de5f3c4f334158a252dc5
libraries/Uint48L5Array.sol	14de0ae5d1daef7ad5769afbe22a69240407816a
libraries/RTokenLib.sol	04feadf633a945f37c8f9b0fc2d6e63e627f48de
libraries/SimulateSwap.sol	a095376e2e2663d6aeaf895ecc500693b395b6b8
libraries/VTokenPositionSet.sol	efcbb1d00716eac955cb6e47ecef6c2f5992ca4f
libraries/GoodAddressDeployer.sol	a11473170d416565885f73601ccde9468a3261c3
utils/ProxyAdminDeployer.sol	442c94a5103890c5d3cab3eb184d0529a7237584
utils/TxGasPriceLimit.sol	5a5a38b4bbe2fbaf894fe41a3b513e3b6011681e
utils/ProxyAdmin.sol	2b9e9274f1690bfae8f898958db822bd343d01a0
utils/TransparentUpgradeableProxy.sol	bea965f99b289107aa3c33b66d8e10b5bef4cae7
utils/Governable.sol	c0d2707fbbb655d2f82da2a5aa1ed08115295618
utils/OptimisticGasUsedClaim.sol	0f70c47905657f85138a9ad289f3ba9b4e1b9397
utils/Extsload.sol	9df2f66e604ed7415f32de925ff42eceda4e3877
protocol/insurancefund/InsuranceFund.sol	c57ecf458b23ca9503fb297c12534a63e637fedb
protocol/insurancefund/InsuranceFundDeployer.sol	9645ffc31dba621d2d72dfd930abf15960f233cb
protocol/wrapper/VPoolWrapperDeployer.sol	c0897c6b3b1455319f8280c19548d43f6e4512a7
protocol/wrapper/VPoolWrapper.sol	8e12335d577045ad9878c931b4d0976ef02a4b82
protocol/clearinghouse/ClearingHouseArbitrum.sol	7a21b1af74ae1666f9f4cdee0988c043e0764864
protocol/clearinghouse/ClearingHouseStorage.sol	abe48ef88ac6f5748269bcd24e003a374d2b239a

File	SHA1
`protocol/clearinghouse/ClearingHouseDeployer.sol`	0cd24cce2f5ab23c9ef12d8be347ef4350edf9d6
`protocol/clearinghouse/ClearingHouse.sol`	afceebba054ef63b1e6a66a4990a5566c5a776c1
`protocol/clearinghouse/ClearingHouseEthereum.sol`	cbc3355abd7e6f86c3b013e264a580d1c9379663
`protocol/clearinghouse/ClearingHouseView.sol`	afeebde3dd47ec71ff7ce0999b046744fcd1dcee
`protocol/tokens/VBase.sol`	732b71a4367ed6ed2a9904878f7fde00bc35319b
`protocol/tokens/VToken.sol`	e3f724e6be2a30a7cf861d5bc933ec3b5c0c3cc0
`protocol/tokens/VBaseDeployer.sol`	2d9f74874a382e8fe42b3fd418572066201196a2
`protocol/tokens/VTokenDeployer.sol`	84f4b3b422a72ff2e63ab45269338c37456db9f4
`protocol/RageTradeFactory.sol`	1c2f3c86e9cc58bb2d9a0ebca2a235f31a611bdb
`interfaces/IVToken.sol`	ea5fc7d9c9afdc942ed0e6f717a9f45409befa14
`interfaces/IVBase.sol`	b237c4bd52899e8b2ff3c5b08e7a295045cfc6f0
`interfaces/IInsuranceFund.sol`	caa41861fa4cd56dadc7f198e692c8387851f0b3
`interfaces/IOracle.sol`	4ea9e4be08990ae8c919d89d623e638878889718
`interfaces/IGovernable.sol`	d54ad13d3d55c15957abb70ef93c93f5ce2dea42
`interfaces/IClearingHouse.sol`	878e06ee2ec7853f1aa9096c55f4816a365b5b96
`interfaces/IVPoolWrapper.sol`	f4432291017aa63fd44bcec4582c2fe95a3d4dda

The rest of the repository was out of the scope of the audit.

1.2 Security Assessment Methodology

The smart contract's code is scanned both manually and automatically for known vulnerabilities and logic errors that can lead to potential security threats. The conformity of requirements (e.g., specifications, documentation, White Paper) is reviewed as well on a consistent basis.

1.3 Auditors

Christoph Michel

2 Severity Levels

We assign a risk score to the severity of a vulnerability or security issue. For this purpose, we use 4 *severity levels* namely:

MINOR

Minor issues are generally subjective in nature or potentially associated with topics like "best practices" or "readability". As a rule, minor issues do not indicate an actual problem or bug in the code. The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

LOW

Low-severity issues are generally objective in nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is a clear reason not to.

MEDIUM

Medium-severity issues are bugs or vulnerabilities. These issues may not be directly exploitable or may require certain conditions in order to be exploited. If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations that make the system exploitable.

HIGH

High-severity issues are directly exploitable bugs or security vulnerabilities. If unaddressed, these issues are likely or guaranteed to cause major problems or, ultimately, a full failure in the operations of the contract.

3 Discovered issues

3.1 Unrestricted function access in VPoolWrapper (high)

When creating a new pool, the `vPoolWrapper` proxy is created and initialized in `RageTradeFactory.initializePool`. All user interactions should go through the `ClearingHouse` which calls the `VPoolWrapper` at some point but the contract functions can also be called directly on the `VPoolWrapper` contract as it is missing any access restrictions.

For example, an attacker can call `vPoolWrapper.swapToken` or `vPoolWrapper.liquidityChange` to arbitrarily swap tokens or add liquidity in the Uniswap pool as the `VPoolWrapper` mints the required vTokens on-demand in the `uniswapV3SwapCallback`/`uniswapV3MintCallback`. An attacker can therefore freely modify the liquidity and token price of the UniswapV3 pool and exploit it to their advantage.

Recommendation

Restrict public `VPoolWrapper` functions to only be callable by the `ClearingHouse`.

Response

> Fixed in 1f9d870

The issue has been fixed.

3.2 Wrong trade position increase computation when changing liquidity (high)

The `LiquidityPosition.liquidityChange`'s `balanceAdjustments.traderPositionIncrease` is supposed to indicate the compositional change of vTokens of the existing liquidity position since the last liquidity provision. It subtracts the vToken amount that is redeemable for the position at the current price (`tokenAmountCurrent`) by `position.vTokenAmountIn`. However, `vTokenAmountIn` tracks the vAmount used to mint the **last liquidity change** instead of the entire position:

```
function liquidityChange(
    Info storage position,
    uint256 accountNo,
    IVToken vToken,
    int128 liquidity,
    IVPoolWrapper wrapper,
    IClearingHouse.BalanceAdjustments memory balanceAdjustments
) internal {
    (
        // @audit-info if liquidity > 0 => vTokens were minted to
           provide liquidity => values are positivevTokens => values
           are negative
        int256 basePrincipal,
        int256 vTokenPrincipal,
        IVPoolWrapper.WrapperValuesInside memory wrapperValuesInside
    ) = wrapper.liquidityChange(position.tickLower, position.tickUpper,
        liquidity);

    position.update(accountNo, vToken, wrapperValuesInside,
       balanceAdjustments);
    balanceAdjustments.vBaseIncrease -= basePrincipal;
    balanceAdjustments.vTokenIncrease -= vTokenPrincipal;

    uint160 sqrtPriceCurrent = wrapper.vPool().sqrtPriceCurrent();
    {
        // @audit-info getAmountsForLiquidity at current price at OLD
           liquidity
        (int256 tokenAmountCurrent, ) = position.tokenAmountsInRange(
           sqrtPriceCurrent);
        // @audit-info how much the vToken in the position increase
           since the last liquidity add
        balanceAdjustments.traderPositionIncrease += tokenAmountCurrent
            - position.vTokenAmountIn;
    }

    if (liquidity > 0) {
        position.liquidity += uint128(liquidity);
        // @audit new vTokenAmountIn is just set to whatever was minted
            above
        position.vTokenAmountIn = vTokenPrincipal;
    } else if (liquidity < 0) {
        position.liquidity -= uint128(liquidity * -1);
        position.vTokenAmountIn = 0;
    }
}
```

This leads to issues when minting only a tiny amount of liquidity but the `tokenAmountCurrent` takes into account the entire liquidity position.

Example

- Initially, call `liquidityChange(1000, ...ticks)`. Imagine, 1000 vToken and 1000 vBase were used. The first time, `traderPositionIncrease = 0 - 0 = 0`. Then `position.vTokenAmountIn = 1000` is set.
- Perform a second tiny `liquidityChange(1, ...ticks)`. Imagine it used 1 vToken and 1 vBase. Then `tokenAmountCurrent = 1000` (assume price hasn't changed) and therefore `traderPositionIncrease = 1000 - 1000 = 0`. Then `position.vTokenAmountIn = vTokenPrincipal = 1` is set.
- Finally, when doing another tiny `liquidityChange(1, ...ticks)` using 1 vToken and 1 vBase the `traderPositionIncrease` is overestimated. Then `tokenAmountCurrent = 1001` (initial + second, assume price hasn't changed). But now `traderPositionIncrease = 1001 - 1 = 1000`. Then `position.vTokenAmountIn = 1` is set.
- Every time a tiny liquidity amount is added, the trader's `traderPositionIncrease` increases by `1000`.

Recommendation

Fix the function. Should `vTokenAmountIn` track `tokenAmountCurrent + vTokenPrincipal` instead?

Response

> Fixed in 5c1d419 and 0e188a5

The issue has been fixed.

3.3 `Uint32L8ArrayLib/Uint48L5ArrayLib.exclude` out-of-bounds access (high)

The `Uint32L8ArrayLib.exclude` function iterates over the array, tries to find the `elementIndex` and the last non-free element index (reuses `i`). (The array is filled from left to right, no non-zero elements come after a zero element.) It then replaces the found element at `elementIndex` with the last non-free element, and clears the last non-free element spot.

```
function exclude(uint32[8] storage array, uint32 element) internal {
    if (element == 0) {
        revert IllegalElement(0);
    }

    uint256 elementIndex = 8;
    uint256 i;

    for (; i < 8; i++) {
        if (array[i] == element) {
            elementIndex = i;
        }
        if (array[i] == 0) {
            i = i > 0 ? i - 1 : 0; // last non-zero element
            break;
        }
    }

    // @audit-info if element was found
    if (elementIndex != 8) {
        // @audit-info the last non-empty index was the index of
            element => can just clear it, no need to fill in holes
        if (i == elementIndex) {
            array[elementIndex] = 0;
        } else {
            // @audit-info the last non-empty index was NOT the index
                of element => need to move last element to element to be
                 removed
            // move last to element's place and empty lastIndex slot
            (array[elementIndex], array[i]) = (array[i], 0);
        }
    }
}
```

However, there is a bug in the code. Imagine `array = [1,2,3,4,5,6,7,8]` and element `8` should be removed by calling `exclude(array, 8)`. Then `elementIndex = 7` but the last non-free element index is `i = 8`. The function tries to run this code: `(array[elementIndex], array[i])= (array[i], 0)` but `array[i]` is out-of-bounds and reverts.

If the array is full, the last element cannot be removed.

> Note that this library is used for the `VtokenPositionSet` and `DepositTokenSet`. Trying to close the last token position in a full array will fail and break core functionality. The same issues can be found in the `Uint48L5ArrayLib` library which is used for the LP positions.

Recommendation

Fix the function. Consider the following pseudo-code (not tested):

```
function exclude(uint32[8] storage array, uint32 element) internal {
    if (element == 0) {
        revert IllegalElement(0);
    }

    uint256 elementIndex = 8;
    uint256 emptyIndex = 8;

    for (; i < 8; i++) {
        if (array[i] == element) {
            elementIndex = i;
        } else if (array[i] == 0) {
            emptyIndex = i;
            break;
        }
    }

    if (elementIndex != 8) {
        // swap with last non-empty index; emptyIndex > 0 because array is non-empty (element was found) and elements are filled left-to-right
        array[elementIndex] = array[emptyIndex - 1];
        array[emptyIndex - 1] = 0;
    }
}
```

Response

> Fixed in 4587ac2

The issue has been fixed in a different way.

3.4 Wrong check in `Calldata.limit` (high)

The `Calldata.limit` reverts if `msg.data.length <= limit_` but should *limit* the `msg.data`, i.e., revert is the size is greater than the limit: `msg.data.length > _limit`. Otherwise, keepers can arbitrarily expand the calldata and receive inflated gas reimbursements at the protocol's loss.

Recommendation

Change the inequality to `if (msg.data.length > limit_)`.

Response

> Fixed in 188be91

The issue has been fixed.

3.5 Unsafe casts (high)

Solidity does not check if values fit in the value range of the new type when doing type casts. Similar to overflow bugs, unchecked type casts can lead to severe bugs as the values are not the expected ones.

The code uses Uniswap's `SafeCast` library (`@uniswap/v3-core-0.8-support/contracts/libraries/SafeCast.sol`) occasionally but it should be used every time it's necessary:

- `SignedFullMath.mulDiv1/2`: For the `int256` type cast on the `FullMath.mulDiv` results.
- `VPoolWrapper._onSwapStep`: For the `int256` type cast on the `vTokenAmount` result.
- `Account.removeProfit`: For the `int256` type cast on the `amount` parameter. Users could steal funds as the `uint256 amount` is transferred but the converted type is subtracted.
- `DepositTokenSet.getAllDepositAccountMarketValue`: For the `int256` type cast on the computed market value.
- `VTokenPositionSet.getAccountMarketValue`: For the `int256` type cast on the computed market value.
- `VTokenPositionSet.getLongShortSideRisk`: For the `int256` type cast on the max net position.
- `ClearingHouse._liquidateLiquidityPositions`: For the `uint256` type cast on the `keeperFee`.
- `VPoolWrapper._onSwapStep`: For the `int256` type cast on all `vTokenAmount`s and `vBaseAmount`s.

Recommendation

In addition to the above-mentioned locations, perform safe type casts *everywhere throughout the codebase*.

Response

> Fixed in e793cbe and ff96f10

3.6 `Uint32L8Set.reduce` does not work (`medium`)

The `Uint32L8Set` library represents a set of 8 32-bit integers in a single `uint256` integer. Its `reduce` function is supposed to call a `fn` on each of the elements of the set. However, there is a bug that skips elements because the set-representing 256-bit integer is shifted twice:

```
function reduce(
    Uint32L8Set set,
    function(uint256, uint32, uint8) view returns (uint256, bool) fn,
    uint256 initialAccumulatedValue
) internal view returns (uint256 accumulatedValue) {
    unchecked {
        accumulatedValue = initialAccumulatedValue;
        uint256 unwrapped = Uint32L8Set.unwrap(set);
        uint32 val;
        console.log('reduce-unwrapped', unwrapped);
        for (uint8 i; i < 8; i++) {
            val = uint32(unwrapped >> (32 * i));
            bool stop;
            console.log('reduce-for-inp', accumulatedValue, val, i);
            (accumulatedValue, stop) = fn(accumulatedValue, val, i);
            console.log('reduce-for-res', accumulatedValue, stop);
            if (stop) break;
            // @audit bug: already gets shifted by 32*i above, no need to cut it off here
            unwrapped >>= 8;
        }
    }
}
```

Imagine `set` is the 256-bit representation of the 8 32-bit integers `[1,2,3,4,5,6,7,8]` and `reduce(set, fn, 0)` is called on it. In the loop iterations, the current loop element is stored in `val` as a bit shift `unwrapped >> (32 * i)` followed by truncation to 32-bits. At the end of each loop, the `unwrapped` value itself is shifted again by 8 bits for no reason.

The reducer function is called with wrong values:

```
set: 0x0000000800000007000000060000000500000004000000030000000200000001
(set:
    215679573381144830513811895868694400695694534256768036697775454289921)

// should be
// reducer fn, 1
// reducer fn, 2
// reducer fn, 3
// reducer fn, 4
// reducer fn, 5
// reducer fn, 6
// reducer fn, 7
// reducer fn, 8
reducer fn, 1
reducer fn, 50331648
reducer fn, 262144
reducer fn, 1280
reducer fn, 6
reducer fn, 134217728
reducer fn, 0
reducer fn, 0
```

Recommendation

> This function is currently not used and therefore not exploitable and its severity level has been reduced. It's recommended to remove it from the codebase so the version with the bug is not used in future development.

Fix the function. Remove the `unwrapped >>= 8;` at the end of the loop.

Response

> Fixed in 1a08c63

The file has been removed.

3.7 Uint32L8Set.exclude can leave holes (medium)

The Uint32L8Set library represents a set of 8 32-bit integers in a single uint256 integer. The exclude function removes an element in-place but the include function assumes that elements are added left to right and that there are no "holes" in the set, i.e., if a zero element is encountered, there will only be zero elements afterwards. This is not the case.

The _includeReducer returns as soon as it finds an empty index and elements could be duplicated.

Recommendation

> This function is currently not used and therefore not exploitable and its severity level has been reduced. It's recommended to remove it from the codebase so the version with the bug is not used in future development.

Fix the exclude function to not leave holes.

Response

> Fixed in 1a08c63

The file has been removed.

3.8 Extsload.extsload(bytes32[]) corrupts memory (medium)

The Extsload.extsload(bytes32[] memory slots) function iterates past the last element and writes to the memory past the last element, corrupting memory. If important data is stored there, it'll be overwritten.

```
function extsload(bytes32[] memory slots) external view returns (
    bytes32[] memory) {
     assembly {
         // @audit-info end = 32 + slotsAddr + slot.length * 32 =
             slotsFirstElementAddr + slot.length * 32
         let end := add(0x20, add(slots, mul(mload(slots), 0x20)))
         for {
             // @audit-info points to slots.length
             let pointer := slots
         } lt(pointer, end) {

         } {
             // @audit-info skips the slots.length at initial `slots`
                address
             // @audit-issue but reads @ `end` in last iteration
             pointer := add(pointer, 0x20)
             let value := sload(mload(pointer))
             mstore(pointer, value)
         }
     }

     return slots;
 }
```

- The end variable points *past* the last element.
- The pointer is initialized to the slots.length field, which comes before the first element of slots.
- The pointer is then advanced by 32 bytes in the loop body.
- Therefore the loop iterates slots.length + 1 times, once too many. In the last iteration, where pointer = end - 32 < end, the pointer is first increased to end in the loop body, to the variable which points *past* the last element. This memory field is overwritten.

Recommendation

> This function is currently not used and therefore not exploitable and its severity level has been reduced. It's recommended to remove it from the codebase so the version with the bug is not used in future development.

Fix the function by decreasing the loop iterations by 1. Consider this function:

```
function extsload(bytes32[] memory slots) external view returns (
   bytes32[] memory) {
    assembly {
        // @audit-info end = 32 + slotsAddr + slot.length * 32 =
           slotsFirstElementAddr + slot.length * 32 = past last element
        let end := add(0x20, add(slots, mul(mload(slots), 0x20)))
        for {
            // @audit-info points to first element
            let pointer := add(slots, 32)
        } lt(pointer, end) {
        } {
            let value := sload(mload(pointer))
            mstore(pointer, value)
            pointer := add(pointer, 0x20)
        }
    }

    return slots;
}
```

Response

> Fixed in c5d7e41

The issue has been fixed.

3.9 `VTokenPosition.deactivate` performs wrong check (`medium`)

The `VTokenPosition.deactivate` function reverts only if *both* the token balance and token LP position are non-zero, instead of if *any of them* is non-zero. A token position can be deactivated if there are no LP positions but even if there's still a negative balance. Luckily, all current callers of this function (`update`) check that both values are zero before calling it and it is currently not exploitable.

Recommendation

Change `if (set.positions[truncated].balance != 0 && !set.positions[truncated].liquidityPositions.isEmpty()){ revert }` to `if (set.positions[truncated].balance != 0 || !set.positions[truncated].liquidityPositions.isEmpty()){ revert }`.

Response

> Fixed in bd8999f

The issue has been fixed.

3.10 UniswapV3 oracle cardinality never increased (`medium`)

The `RageTradeFactory` code deploys a UniswapV3 oracle but does not increase the cardinality (observation slots) of the oracle. Therefore, only a single block of historic price data can be stored. This is especially important as `UniswapV3PoolHelper.twapTick` returns the *current tick* (which can be manipulated) if there is no observation that is old enough.

Recommendation

We recommend computing how many observation slots are needed (at least `twapPeriod / averageBlockTime`) and calling `uniswapV3Pool.increaseObservationCardinalityNext(slots)` when deploying a pool.

Response

> Fixed in b5a4906

The issue has been fixed by initialzing the pool with a cardinality of 100.

3.11 Users cannot withdraw when RTokens are unsupported (`medium`)

The `ClearingHouse.updateSupportedDeposits` function allows disabling a token for deposits. The `addMargin`/`removeMargin` functions revert in `_getRTokenWithChecks` if these tokens are suddenly unsupported. Users cannot add or remove margin anymore.

Recommendation

Communicate this risk to users. Consider allowing users to close their positions and retrieve their margin deposits in case the token becomes unsupported.

Response

> Fixed in c36a2fb

The issue has been fixed by allowing users to withdraw tokens that became unsupported.

3.12 Might not be able to take profit (`medium`)

There might not be enough `rBase` tokens to take profit with `ClearingHouse.removeProfit`. Imagine a user opens a leveraged short on ETH, then liquidity is pulled from the vPool and margin withdrawn from the platform, then, over time, the ETH market value (`VTokenPosition.marketValue`) using TWAP goes to zero. The user should be able to withdraw their leveraged short profit but there might not be enough `rBase` tokens in the contract.

Example

1. someone adds margin, provides super-concentrated liquidity at some price range [1.0, 1.1] and the current price is in the middle of it

Imagine all of this happens in the same block so it doesn't change the TWAP:

1. someone adds margin, shorts all tokens on leverage by selling into this liquidity, price goes out of liquidity range to zero. (but TWAP still measures it at 1.05)
2. the LP withdraws all their liquidity from that range, receives only vTokens. Should be able to withdraw most of their margin again because the redeemed vTokens are still priced at original TWAP?

In the following blocks the TWAP goes to zero and the profit gets too large for what's currently in the platform.

Response

> This scenario can come in if there is a sudden fall in the market which stays like that. Using TWAP protects against the flash attacks but introduces a lag from the current price. In this specific scenario a liquidator will liquidate the account which would be negative when TWAP becomes 0 and the insurance fund will cover the difference. The probability of this event (price going to 0 / sudden large fall) should be very low, especially for large cap tokens. No fix needed.

In addition, the team has considered adding protocol-owned liquidty across a large range to make TWAP manipulations more costly.

3.13 Permissionless removal of limit orders can be dangerous (`medium`)

Anyone can remove limit orders of any account that are filled by calling the `ClearingHouse.removeLimitOrder` function.

While this is the intended behavior it can still lead to issues:

- Trades in the opposite direction can be frontrun and liquidity can be pulled by anyone, leading to unexpected, worse trades if no tight slippage parameters are set.
- The check if removing the limit order is valid is determined by the **TWAP** (see `VTokenPositionSet.removeLimitOrder`'s `currentTick = vToken.getVirtualTwapTick(protocol)`) and liquidity position's `baseValue` is also simulated to be redeemed at the TWAP. But the actual liquidity is removed **at the current price**. It could be that the current price is different from the TWAP and the removal ends up redeeming both vToken and vBase. The redeemed tokens are then priced at the

TWAP again (see `VTokenPositionSet.getAccountMarketValue`), i.e., **the account market value can change after a limit order removal** and there is no margin check. Attackers can remove limit orders which might put accounts at liquidation risk and then liquidate them.

Response

> Issue fixed - Limit order removal should use currentTick rather than twapTick to check if order is filled. Removal of limit order away from TWAP price (if current price is far away due to flash attack) would not be an issue since token value on removal would be higher (Spreadsheet). Fixed in d3d8299 (and df442dd, 667f30e, 7ed75a4).

The issue has been fixed. As the liquidity is only removed if the order is fully filled now (judged by the current price used for redemption), the market value (`TWAP * vToken + vBase`) does not decrease if the current price moves further in the direction of the TWAP. The redeemed amounts would not change further and therefore removals of limit orders cannot decrease the market value anymore.

3.14 `SignedFullMath` always rounds down (`low`)

The `SignedFullMath.mulDivRoundingDown` functions always round down, even if no rounding is necessary. For example, `mulDivRoundingDown(-2, 1, 2)= -2 * 1 / 2` should be `-1` but is rounded to `-2`.

Recommendation

Only round down if `a * b % denominator != 0`.

Response

> Fixed in 8e11dff

The issue has been fixed.

3.15 Real token never initialized (`minor`)

The `ClearingHouse.initRealToken` function is never called by the `RageTradeFactory` and the map `realTokenInitilized[realToken]` remains false.

Recommendation

As the map `realTokenInitilized` does not seem to be used for anything in the contract, consider removing these functions.

Response

> Real token initialized not needed anymore - have removed it in 5d40761

The issue has been fixed.

3.16 `LiquidityPositionSet.getLiquidityPosition` can be more efficient (`minor`)

The `LiquidityPositionSet.getLiquidityPosition` first includes the position in the set (`_include(...)`) and then checks that the position at that included `positionId` is initialized.

Recommendation

The position does not need to be included first, it's enough to immediately check if the position is initialized:

```
function getLiquidityPosition(
    Info storage set,
    int24 tickLower,
    int24 tickUpper
) internal returns (LiquidityPosition.Info storage position) {
    if (tickLower > tickUpper) {
        revert IllegalTicks(tickLower, tickUpper);
    }

    // @audit does not need to include and iterate over the set
    uint48 positionId = Uint48Lib.concat(tickLower, tickUpper);
    position = set.positions[positionId];

    if (!position.isInitialized()) revert InactiveRange();
    return position;
}
```

Response

> Fixed in b2c51e0

The issue has been fixed.

3.17 `Uint32L8ArrayLib.include` can be more efficient (`minor`)

The `Uint32L8ArrayLib.include` function keeps iterating even if the element was not found and the empty index was already set. (The array is filled from left to right, no non-zero elements come after a zero element.)

> The same issues can be found in the `Uint48L5ArrayLib` library.

Recommendation

Some iterations can be saved by **`break`**ing once the `emptyIndex` has been set.

```
function include(uint32[8] storage array, uint32 element) internal {
    if (element == 0) {
        revert IllegalElement(0);
    }

    uint256 emptyIndex = 8; // max index is 7
    for (uint256 i; i < 8; i++) {
        if (array[i] == element) {
            return;
        }
        // @audit should use break here instead of == 8
        // if (emptyIndex == 8 && array[i] == uint32(0)) {
        //     emptyIndex = i;
        // }
        if (array[i] == uint32(0)) {
            emptyIndex = i;
            break;
        }
    }

    if (emptyIndex == 8) {
        revert NoSpaceLeftToInsert(element);
    }

```

```
        array[emptyIndex] = element;
    }
```

Response

> Fixed in 1a08c63

The file has been removed.

3.18 TWAP includes weighted zero prices of invalid Chainlink rounds (high)

The `ChainlinkOracle` contract assumes that `roundId`s are incremental and decrements these (starting from the latest round) to fetch historical price data. The Chainlink docs say that they are not incremental:

> "roundId is NOT incremental. Not all roundIds are valid." "The `roundId` can jump significantly when the phaseId is updated"

The code currently does also not filter out invalid roundIds (`latestPrice == 0` or `latestTS == 0`). The TWAP can currently be drastically reduced if an invalid round is encountered (`latestPrice == 0` but `latestTs > 0`) as it will include the 0 price with a non-zero duration weight.

This can lead to sudden liquidations.

Recommendation

Be aware that this way of enumerating historic rounds is against what Chainlink suggests. Filter out prices of invalid round.

Response

> Fixed in 758f794

If a price of `0` is encountered, the oracle now stops and returns the TWAP for the non-zero price period.

3.19 Miscellaneous (minor)

- `Account.LiquidationParams`: The `fixFee` and `minRequiredMargin` variables do not exist on the **`struct`** anymore but are still listed in the docs.

> Fixed in 308e9ab

- `Account.liquidateLiquidityPositions`: The `protocol.liquidationParams.liquidationFeeFraction` is divided by `1e5` whereas all other `Account.LiquidationParams` are in basis points (`1e4`). Document the different base for all of these parameters and consider changing all percentages to a common base.

  > Acknowledged

- `Account.getLiquidationPriceX128AndFee`: The parameter descriptions in the docs do not match the parameters

  > Fixed in 308e9ab

- `Account.updateLiquidationAccounts`: Function uses a named return variable `liquidatorBalanceAdjustments` but never sets it. The last `balanceAdjustments` assignment should be to `liquidatorBalanceAdjustments` instead (and the **`return`** statement should be skipped).

  > Fixed in 308e9ab

- `Uint32L8Set._existsReducer`: Function not used.

  > Fixed in 1a08c63

- `Uint48Lib.concat`: No need to mask the 24 lower bits on the 24-bit `val2` value with `and(val2, 0x000000ffffff)`. One can directly use the 24-bits of `val2`.

  > False positive, required as `val2` is a *signed* integer and EVM performs sign extension.

- `ClearingHouseStorage.realTokenInitilized`/`ClearingHouseView.isRealTokenAlreadyInitilized`: TYPO: `Initilized -> Initialized`

  > Fixed in 5d40761

- `ClearingHouse`: The `*WithGasClaim` functions are not part of the `IClearingHouse` interface.

  > Fixed in 800623f

- `ClearingHouseEthereum`: The `TODO put a upper limit to tx.gasprice` seems to already be resolved by use of the `checkTxGasPrice(tx.gasprice)` modifier.

  > Acknowledged

- `ClearingHouseEthereum`: EVM gas costs can change over time with new forks.

  > Acknowledged

- `VBaseDeployer._isVBaseAddressGood`: comment should say "most significant hex char of address is "f"" instead of "d".

> Fixed in e2710a1

- `VPoolWrapper.swap`: Regarding the TODO questioning if the `assert(vTokenIn_simulated == vTokenIn && vBaseIn_simulated == vBaseIn)` should be removed: This invariant should always be true, so we recommend keeping it to be able to easier detect any issues.

  > Acknowledged

- `Tick.getUniswapFeeGrowthInside`: This function is not used and should be removed.

  > Fixed in 801deed

4 Conclusion

A critical issue has been found that leads to arbitrary liquidity and price manipulations of the pools. Two issues have been found that break the user accounting under certain circumstances. Unsafe type conversions to signed integer values are done throughout the codebase which can be exploited.

Overall, the codebase is of high quality. The documentation could be improved, the protocol is currently described at a very high level in the gitbook docs or with computations at a very low level in several spreadsheets. Consider finding a middle ground and adding these docs to the gitbook. The inline code comments are helpful but there could be more of them, especially when writing assembly where several bugs were found. The codebase is structured in a modular way which makes each module short and easy to read by itself but a single user interaction touches many libraries making it hard to keep the surrounding context in mind. All in all, we still think this modular approach is fitting for the protocol. The test environment is outstanding, covering all files with several different test scenarios.

Update: The issues have been fixed and the documentation has been revised and improved.

Disclaimer

This audit is based on the scope and snapshot of the code mentioned in the introduction. The contracts used in a production environment may differ drastically. Neither did this audit verify any deployment steps or multi-signature wallet setups. Audits cannot provide a guarantee that all vulnerabilities have been found, nor might all found vulnerabilities be completely mitigated by the project team. An audit is not an endorsement of the project or the team, nor guarantee its security. No third party should rely on the audits in any way, including for the purpose of making any decisions about investing in the project.