sigma prime

Tracer DAO

Tracer Protocol

Smart Contract Security Assessment

Version: 1.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Tracer smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Tracer smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Tracer smart contracts.

Overview

Tracer is a protocol built to democratise derivatives (financial contracts that derive their value from one or multiple underlying assets) by vastly reducing the barriers to participate in derivative markets.

The protocol will be governed by a Decentralised Autonomous Organisation (DAO) named the "Tracer Protocol DAO", responsible for developing, deploying and managing the `TracerFactory`, which can itself be used by users to deploy `Tracer` markets. The first type of derivative that will be supported by the Tracer protocol will be perpetual swaps (similar to futures and options contracts but without an expiration date).

Tracer is being developed by Tracer DAO, a collective composed of crypto-economists, open-source ideologues and other like-minded individuals working towards building technology to support a free and decentralised world.

This report focuses on the smart contracts that constitute the Tracer perpetual swap system as outlined in the tracer whitepaper [1].

Security Assessment Summary

This review was conducted on the files hosted on the tracer-protocol repository and targeted commit cb003be.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

Beyond the findings mentioned in this report, we would like to add a few remarks around the trust model and design of the protocol. As it stands, there a few main contracts which hold the value of tokens deposited into the system, specifically `Account` and `Insurance`. The `Account` contract holds all tokens for all deployed tracers that use this account contract (likely to be most tracer contracts in order for them to use the standard insurance pool). A tracer market can be deployed by any user and thus the global `Account` and by extension `Insurance` contracts are able to run arbitrary code via external calls to these arbitrary tracer contracts. As these contracts can hold potentially large values of tokens, we strongly recommend separating the tracer market accounts so that there is a not a single contract holding all the tokens for all markets and trusted markets are not required to run arbitrary code via external calls to malicious tracers. The same recommendation is for the `Insurance` contract. Having a tracer market instead inherit the logic of these global contracts will also decrease the gas usage of the contracts by minimizing the required external calls.

As for the trust model, any users are able to deploy their own Tracer markets with administrative privileges. These markets can perform various re-entrancies, price manipulations and fundamentally control the tokens that are deposited into the related market. This is by design, and users should only use markets for which they trust the related owners. Therefore, the testing team did not focus on issues relating to malicious tracer owners. The testing team did however focus on attacks involving malicious markets in the presence of honest markets where malicious markets could affect honest markets.

Given the time frame, the testing team reviewed various aspects and mechanics of the swap protocol mathematics, and the related issues have been raised in this report. Given the severity of the issues found and the complexity of the protocol mechanics, we recommend further testing at scale (preferably via a public testnet) to further ensure that the protocol design operates as expected.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [2, 3].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 36 issues during this assessment. Categorized by their severity:

- Critical: 5 issues.
- High: 6 issues.
- Medium: 6 issues.
- Low: 8 issues.
- Informational: 11 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Tracer smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
TCR-01	Unsafe Casting of `int256` to `uint256`	Critical	Open
TCR-02	Funding Rate Manipulation #1	Critical	Open
TCR-03	Funding Rate Manipulation #2	Critical	Open
TCR-04	`claimReceipts()` `market` Parameter is Unchecked	Critical	Open
TCR-05	Malicious Users Can Take All Rewards and Prevent Transfers	Critical	Open
TCR-06	Restriction on User Withdrawals	High	Open
TCR-07	Hourly Price Consists of a Single Trade	High	Open
TCR-08	Risk Free Negative Price Trades	High	Open
TCR-09	Signature Use in Alternate Markets	High	Open
TCR-10	Insurance Funding Rate Increases Indefinitely	High	Open
TCR-11	Double Voting by Delegaters	High	Open
TCR-12	Possible Double Voting for Proposers	Medium	Open
TCR-13	`get24HourPrices()` Forces Prices to be Positive	Medium	Open
TCR-14	Insurance Pool Tokens Can Be Transferred Without Claiming Rewards	Medium	Open
TCR-15	Reenterable Withdrawal Pattern	Medium	Open
TCR-16	Insufficient Timing Parameter Validation	Medium	Open
TCR-17	Insufficient Input Validation on `proposalId` in Voting Functions	Medium	Open
TCR-18	Simple Majority Not Required to Pass Proposals	Low	Open
TCR-19	Incorrect use of Absolute Value in Calculations	Low	Open
TCR-20	TWAPS Exclude Hours Where Price is Zero	Low	Open
TCR-21	Overwrite `orderIdByHash` for Duplicate Orders	Low	Open
TCR-22	Insertion of Expired Orders	Low	Open
TCR-23	Matching Filled Orders	Low	Open
TCR-24	Potential Contract Configuration Mismatch	Low	Open
TCR-25	Partially Implemented `feeRate`	Low	Open
TCR-26	Tracer constructor parameters	Informational	Open

TCR-27	Checks-Effects-Interaction Pattern	Informational	Open
TCR-28	Unnecessary `calcEscrowLiquidationAmount()` function	Informational	Open
TCR-29	Lack of Limitations/Validation To Set Variables in `Tracer`	Informational	Open
TCR-30	User Created Markets Can Be Exploited By Owner	Informational	Open
TCR-31	Zero Address Passes `verifySignature()`	Informational	Open
TCR-32	Front Running	Informational	Open
TCR-33	Trades can be Executed with Different Amounts	Informational	Open
TCR-34	Orders Can Be Created That Can Never Be Filled	Informational	Open
TCR-35	Unused Variables & Functions	Informational	Open
TCR-36	Miscellaneous General Statements	Informational	Open

TCR-01	Unsafe Casting of `int256` to `uint256`		
Asset	`Tracer.sol`		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In the functions `permissionedTakeOrder()` and `matchOrders()`, the order price is cast to a `uint256` to calculate the base change, as seen in the following line.

```
int256 baseChange = (fillAmount.mul(uint256(order.price))).div(priceMultiplier).toInt256();
```

It is a valid use case to have price as a negative number. Solidity uses 2's compliment hence when casting a negative `int256` to a `uint256` the result will be greater than 2^{255}, since the first bit of a negative `int256` will always be set to one (1).

The impact in `Tracer` is that the `baseChange` will be significantly overstated and will contain the wrong sign. A malicious user could positively move their `base`, for a trade that should move their `base` in the opposite direction.

The inverse would occur to the user who took the other side, negatively impacting their base.

This would signficantly increase the margin for the user who's base moved in the positive direction allowing them to withdraw large sums of the base token.

Additionally there is the requirement, `require(baseChange > 0, "TCR: Margin change <= 0")` which should not hold true for negatively priced orders.

Another case of incorrect casting is in `Tracer.sol:L156`, namely, in the price and gas oracle contracts. `Tracer.sol:permissionedMakeOrder()` attempts to place an on-chain order and queries an instance of `GasOracle.sol` to receive the off-chain gas cost. `GasOracle.sol:latestAnswer()` calculates the gas cost in terms of the underlying market asset price. However, it is not equipped to handle negative prices and will therefore return a negative gas cost which is later cast from `int256` to `uint256`. As a result, orders will likely not be made as accounts will fail the `marginIsValid()` check in `Tracer.sol:L198`.

Recommendations

We recommend that the calculations for `baseChange` are all executed as `int256` rather than `uint256`. Additionally, the statement `require(baseChange > 0, "TCR: Margin change <= 0")` should be removed.

We also recommend that `GasOracle.sol` queries the absolute value of the price oracle instead of an `int256` value.

TCR-02	Funding Rate Manipulation #1		
Asset	`Pricing.sol` & `Tracer.sol`		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `Pricing.updatePrice()` is called whenever a trade is made. This is used to store hourly price values which are used in the calculations of the funding rate.

Users are able to manipulate this value by making trades with themselves (i.e. taking both the long and short for a trade), which will have a net zero impact on their position.

The malicious user may make numerous trades with prices exponentially higher or lower than the underlying price. This will essentially allow them to set the derivative price and thus the funding rate to one of their choice.

A similar funding rate manipulation can occur by using two different accounts and making pairs of trades each with opposite positions. e.g.

Order1 - UserA short, UserB long: 0.000001@10,000,000,000

Order2 - UserA long, UserB short: 0.000001@10,000,000,000

A similar technique can be used to manipulate the average price when liquidating an order. This can be used to drain the insurance pool.

Recommendations

On chain Time-Weighted-Average-Price (TWAP) systems can easily be manipulated if users can make trades at arbitrary prices. To use a TWAP orders must exist in a priority queue and orders be settled for the best price first. Therefore if a user wishes to execute an order at a very high or low price they would have to execute all orders which have a better price first.

This will still have limitations if there is low liquidity as users would be able to trade through the order book to then execute transactions at the desired price. Furthermore, the order book is empty when the contract is first deployed and so a malicious user may apply this attack.

TCR-03	Funding Rate Manipulation #2		
Asset	`Account.sol` & `Tracer.sol`		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The funding rate is a time-weighted average difference between the oracle price and the market price. Users are either rewarded or taxed based on whether the oracle price is above or below the market price and whether they are long or short.

Users are able to manipulate their funding rate index to potentially earn themselves rewards and/or avoid paying fees.

The funding rate will be applied to a trade *after* the base and quote have been adjusted for the trade as:

$$base = base - (currentGlobalRate - currentUserRate) * quote$$

Consider the following example where:

- $currentGlobalRate > 0$
- $user_1$ and $user_2$ have $currentUserRate_i = 0$

The following steps can be used in increase a users base.

1. $user_1$ and $user_2$ both deposit amount d_i respectively,
2. $user_1$ makes a trade of negligible size (or trades with them self) such that $currentUserRate_1 = currentGlobalRate$
3. $trade_1 : \{price_1, \ amount_1, \ longUser : user_1, \ shortUser : user_2\}$
   Since $currentUserRate_1 = currentGlobalRate$:

   $$\begin{aligned} user_1 : base &= d_1 - amount_1 * price_1 - (currentGlobalRate - currentUserRate_1) * amount_1 \\ &= d_1 - amount_1 * price_1 \end{aligned}$$

   Since $user_2$ was short the quote will be negative the amount of the trade and $currentUserRate_2 = 0$ hence:

   $$\begin{aligned} user_2 : base &= d_2 + amount_1 * price_1 - (currentGlobalRate - currentUserRate_2) * (-amount_1) \\ &= d_2 + amount_1 * price_1 + currentGlobalRate * amount_1 \end{aligned}$$

   Both user will now have $currentUserRate = currentGlobalRate$.

4. We now close the position through $trade_2 : \{price_1,\ amount_1,\ longUser : user_2,\ shortUser : user_1\}$ which gives us

   $user_1 : base = d_1$

   $user_2 : base = d_2 + currentGlobalRate * amount_1$

This attack can be applied if $currentGlobalRate < 0$ by switching the long and short users in steps three (3) and four (4).

The impact of this attack is that users are able to artificially increase their base. By creating numerous accounts and applying this attack multiple times an attacker would be able to withdraw all of the funds in the protocol.

Recommendations

The funding rate needs to ensure that the net impact on the sum of user's base is zero. One method for doing this is applying the global rate directly to the amount in the trade without using user specific rates or user specific quotes, this will ensure that the change in base of both users will be the same absolute value but opposite in sign.

TCR-04	`claimReceipts()` `market` Parameter is Unchecked		
Asset	`Account.sol` & `Receipt.sol`		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In `Account.claimReceipts()` the parameter `market` isn't verified against the `receipt.tracer`. This allows a user to claim receipts based on orders that originate from other (malicious) tracer markets.

This vulnerability is classified as high impact as it is an example of how a malicious tracer market can effect legitimate tracer markets.

This vulnerability allows malicious users to withdraw maximum funds from the insurance pool by manipulating fake orders in malicious tracers. Malicious users could liquidate themselves on legitimate markets and claim orders on fake markets.

Recommendations

We recommend retrieving the `market` from the receipt object, specifically `receipt.tracer`. Then removing the parameter `market` from both `Account.claimReceipts()` and `Receipt.claimReceipts()`.

TCR-05	Malicious Users Can Take All Rewards and Prevent Transfers		
Asset	InsuranceTokenPool.sol		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `InsurancePoolToken` contract is designed to hold a number of stakers and reward them with tracer tokens based on the staker's token share.

The contract overrides the `transfer()` function, enforcing the current user to withdraw their rewards before transferring any amount of tokens to a new account. However, the new account has no such modification and can withdraw an equal share as the original transferrer. This means a single staker can withdraw their rewards then transfer their tokens to a new address that they own and repeat the process until the rewards are entirely drained from the contract.

A follow on effect is that current users in the system can no longer withdraw rewards, in fact the `_withdrawFunds()` function will revert when the rewards are depleted but users are still entitled to rewards (due to the safe math subtraction). This means that in this scenario, a malicious user also prevents honest users from using the `transfer()` function as it will revert due to lack of rewards.

NOTE: See the accompanying tests for the insurance pool token for an example of this in practice.

Recommendations

Transfers of tokens to other accounts need to update the other accounts `lastRewardsUpdate`. Care needs to be taken to not simply reset the variable to prevent malicious users from sending tokens to negatively effect other users.

One potential solution, is to also call `_withdrawFunds()` on the `to` address before the tokens get transferred. This will however mean that users can force other users to withdraw their share of tokens at any time.

The correction to this issue, should also go hand in hand with the correction to TCR-14.

TCR-06	Restriction on User Withdrawals		
Asset	`Account.sol`		
Status	**Open**		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The contract `Account` is used to store the balance ERC20 Tokens that users have deposited in the markets. These user balances are updated as users make trades. A user who makes a profit from these trades would then withdraw their profits and/or deposited balance via the function `withdraw()`.

On line [138] of the function `withdraw()` the users deposited balance is updated as such `userBalance.deposited = userBalance.deposited.sub(amount);`.

The variable `userBalance.deposited` is only increased when a user makes a deposit and decreased when a user withdraws. Since this function uses `SafeMath` and `deposited` is of type `uint256` the function will revert if a user attempts to withdraw more than the total balance they have deposited.

The implications are that a user will be unable to withdraw profits, their withdrawals are limited to the amount they have deposited.

Recommendations

We recommend removing the field `deposited` from the type `AccountBalance` and remove the instances where it is used.

TCR-07	Hourly Price Consists of a Single Trade		
Asset	`Tracer.sol & Pricing.sol`		
Status	Open		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

When an order is matched or taken, the funding rate is updated through two calls to `Pricing`. These calls are `updatePrices()` and `updateFundingRate()`.

In `Tracer.updateInternalRecords()` if more than an hour has passed, `currentHour` is incremented by one (1) modulus twenty-four (24). We then call `Pricing.updatePrice()` followed by `Pricing.updateFundingRate()` as seen in the excerpt below.

```
if (currentHour == 23) {
    currentHour = 0;
} else {
    currentHour = currentHour + 1;
}
// Update pricing and funding rate states
pricingContract.updatePrice(price, ioracle.latestAnswer(), true, address(this));
int256 poolFundingRate = insuranceContract.getPoolFundingRate(address(this)).toInt256();

pricingContract.updateFundingRate(address(this), ioracle.latestAnswer(), poolFundingRate);
```

An issue arises here as the call to `Pricing.updatePrice()` has the parameter `newRecord` flag set to `true`. This means that `pricing.hourlyTracerPrices[currentHour]` will consist of a single answer (the one from the current order).

During the call to `Pricing.updateFundingRate()` when calculating the TWAPs `getHourlyAvgTracerPrice(uint256(j), market)` and `getHourlyAvgOraclePrice(uint256(j), market)` both will only have one entry for `currentHour`, which is the current trade.

Since `currentHour` will have the highest weighting (8) it will therefore have

$$\frac{8}{8+7+6+5+4+3+2+1} = 22\%$$

Note this issue is exaggerated when less than twenty-four (24) hours have past. Example after completion of the first hour it would be

$$\frac{8}{8+7} = 53\%$$

A user who is able to manipulate the `fundingRate` will be able to receive significant compensation in future trades through the calculation of their base in `Account.settle()`, allowing them to inflate their profits at the expense of users holding the opposite position.

Recommendations

This issue may be mitigated by calculating the TWAPs in `Pricing.updateFundingRate()` for `currentHour - 1` thereby ignoring the most recent trade.

Alternatively the issue may also be mitigated in `Trace.updateInternalRecords()` by calling `Pricing.updatePrice()` and `Pricing.updateInternalRecords()` before incrementing `currentHour`.

TCR-08	Risk Free Negative Price Trades		
Asset	`Account.sol`		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The tracer markets are designed to allow for negative price trades. However users can make trades without making deposits (or can make a trade and then withdraw their initial deposit (see Maximum Withdrawal Observations for a simple analysis)).

The issue occurs due to the calculations of notional value, $|quote| * price_{fair}$ Thus if $price_{fair} < 0$ notional value will be less than zero. Since `minMargin` is a portion of the notional value it will also be negative (excluding potential liquidation gas costs).

Now if we take an order at the $price_{fair}$ then our margin will be zero.

Thus, the check `margin > minMargin` in `marginIsValid()` will always pass even though as user has made no deposits.

Recommendations

Update the calculations for `minMargin()`, leveraged notional value and notional value to handle the cases where pricing is negative.

TCR-09	Signature Use in Alternate Markets		
Asset	`Trader.sol`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `Trader` contract makes trades on behalf of users. The purpose is that it is an off-chain order book system which can execute orders on-chain on a user's behalf.

It proves a user requested an order by having them sign the `hashOrder()` of a `LimitOrder`. This ensures that the details of the order match those desired by the user.

The `LimitOrder` contains the field `targetTracer` which links the order to a specific market. However, the function `executeTrade(makers, takers, market)` does not ensure that the function parameter `market` matches the variable `targetTracer` in the order.

The impact is that a user may call `executeOrder()` using signatures that were intended for a different market. Since different markets will have different prices this will result in users creating highly unprofitable trades. A malicious user may capitalise on this by taking the opposite side of these trades.

This attack would require the user to have given permission to the `Trader` in both markets, for the user's margin to be valid after the trade and finally for the transaction to have a valid nonce.

Recommendations

A check should be added to each order to ensure that the `order.targetTracer == market`, thereby preventing orders to be executed in other markets.

TCR-10	Insurance Funding Rate Increases Indefinitely		
Asset	`Pricing.sol`		
Status	**Open**		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The insurance funding rate is calculated in `Insurance.getPoolFundingRate()` to be proportional to the difference between the leveraged notional value and pool holdings. The value returned from this function is a `uint256` as the insurance funding rate should be non-negative.

When the insurance funding rate is being updated in `Pricing.updateFundingRate()` the following occurs.

```
int256 currentInsuranceFundingRateValue =
                                   getOnlyInsuranceFundingRateValue(market, fundingIndex);
int256 IPoolFundingRateValue = currentInsuranceFundingRateValue.add(IPoolFundingRate);
```

Here `IPoolFundingRate = Insurance.getPoolFundingRate()` and thus is always greater than or equal to zero (0). The value `IPoolFundingRateValue` which adds `IPoolFundingRate` and the `currentInsuranceFundingRateValue` then becomes the new insurance funding rate value.

Since we are taking the current insurance funding rate value and adding a number that is greater than or equal to zero, the new insurance funding rate value may only ever increase.

The impact here is that users will have to pay significant fees to the insurance pool even when the pool has reached the target amount.

Recommendations

Consider removing the step where the insurance funding rate is added to the previous insurance funding rate, such that the value is set to the output of `Insurance.getPoolFundingRate()`.

TCR-11	Double Voting by Delegaters		
Asset	`Gov.sol`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If a user has delegated their voting rights to another member of the governance network, the recipient of these rights is able to propose/vote with the combined value of their stake plus any delegated stake. However, as there are no restrictions on withdrawing delegated stake once a vote has already been made, a user is able to double vote by transferring this stake to another account and voting on the same proposal.

This process can be further outlined in the following scenario:

1. Alice and Bob both decide they want to stake their governance tokens.
2. Bob delegates his stake to Alice, relinquishing him of his voting rights until he decides to remove Alice as his delegate.
3. Alice submits an arbitrary proposal, whereby her entire stake (including any delegated stake) is counted as a **Yes** vote.
4. Alice and Bob both wait some time for the warm up period to pass and then Bob proceeds to withdraw his stake while the proposal is still actively being voted on.
5. Bob then transfers his governance token to an arbitrary account controlled by him and restakes his governance tokens.
6. With no cooldown on voting after staking, Bob is able to vote on the same proposal, effectively double voting.

Refer to the test `test_withdraw_double_vote()` in `test_gov.py` for a proof of concept.

Recommendations

Ensure that no vote is counted twice and potentially add a lockup period when entering and exiting the system.

Alternatively, voting could be restricted to pre-existing users who have staked their funds before the respective proposal has been made.

TCR-12	Possible Double Voting for Proposers		
Asset	`Gov.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

If the `lockupPeriod` is less than the `proposalDuration`, it is possible for a proposer to have their votes (i.e. amount staked) counted twice.

Consider the following scenario:

1. The `lockupPeriod` is reduced to 1 day by a successful proposal. The `proposalDuration` is stil set to its default value of 3 days.
2. Alice submits an arbitrary proposal (the targets/data for this proposal do not matter for the purpose of this example). As the proposer, her entire staked amount (let's assume 1000 tokens) is counted as a **Yes** vote.
3. Alice waits 1 day, then proceeds to delegate her votes to Bob (who has the same amount of staked tokens, 1000).
4. Bob proceeds to vote in favour of Alice's proposal. Because Alice delegated her tokens to him, Bob can vote with an amount of 2000 tokens.
5. As a result, at this stage, the total amount of **Yes** votes equals 3000 tokens, while the amount of actual stake backing these votes is only 2000
6. Alice's votes have effectively been counted twice.

Refer to the test `test_double_vote()` in `test_gov.py` for a proof of concept.

Recommendations

Ensure that no vote is counted twice, even for short lockup periods. Consider restricting the values of the `lockupPeriod` to be greater than or equal to the `proposalDuration`.

TCR-13	`get24HourPrices()` Forces Prices to be Positive		
Asset	`Pricing.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Calculation of `get24HourPrices()` takes the average price for each hour in the last twenty-four (24) hours. Then takes an average of those hours.

During the calculations each hourly price is converted to a `uint256` after taking the absolute value as such `(uint256(hourlyPrice.totalPrice.abs())`.

This will have undesirable effects for prices which fluctuate between positive an negative numbers. This is because price x and $-x$ will be treated the same in the average.

Recommendations

Consider keeping the prices as `int256` to account for cases where prices may fluctuate between positive and negative values.

TCR-14	Insurance Pool Tokens Can Be Transferred Without Claiming Rewards		
Asset	`InsuranceTokenPool.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `InsurancePoolToken` contract overrides the `transfer()` function, enforcing the current user to withdraw their rewards before transferring any amount of tokens to a new account. However, the ERC20 `transferFrom()` does not have a similar override. This means a staker can transfer their tokens to another account without withdrawing their rewards, leaving unclaimed rewards in the pool.

NOTE: See the accompanying tests for the insurance pool token for an example of this in practice.

Recommendations

The `transferFrom()` function should also be overridden to call `_withdrawFunds()` and/or handled with the correction for TCR-05.

TCR-15	Reentrerable Withdrawal Pattern		
Asset	Account.sol		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The withdrawal of funds from the `Account` contract is vulnerable to reentrancy.

```
function withdraw(uint256 amount, address market) external override {
    ITracer _tracer = ITracer(market);
    require(amount > 0, "ACT: Withdraw Amount <= 0");
    Types.AccountBalance storage userBalance = balances[market][msg.sender];
    require(
        marginIsValid(
            userBalance.base.sub(amount.toInt256()),
            userBalance.quote,
            pricing.fairPrices(market),
            userBalance.lastUpdatedGasPrice,
            market
        ),
        "ACT: Withdraw below valid Margin"
    );
    address tracerBaseToken = _tracer.tracerBaseToken();
    IERC20(tracerBaseToken).safeTransfer(msg.sender, amount);
    userBalance.base = userBalance.base.sub(amount.toInt256());
    userBalance.deposited = userBalance.deposited.sub(amount);
    int256 originalLeverage = userBalance.totalLeveragedValue;
    _updateAccountLeverage(userBalance.quote, pricing.fairPrices(market),
        userBalance.base, msg.sender, market, originalLeverage);

    // Safemath will throw if tvl[market] < amount
    tvl[market] = tvl[market].sub(amount);
    emit Withdraw(msg.sender, amount, market);
}
```

The above code in an excerpt of the function `withdraw()`. Here the check to ensure that the margin is valid (`marginIsValid()`) occurs before the funds are transferred to the user in `IERC20(tracerBaseToken).safeTransfer(msg.sender, amount)`, which occurs before the balances are updated.

If a malicious user was able to gain control over the execution during `IERC20(tracerBaseToken).safeTransfer(msg.sender, amount)` then the user would be able to call `withdraw()` a second time. This second call to `withdraw()` would occur before the users balances have been updated during the first iteration and so they will again pass the check `isMarginValid()`, even if the margin after both transfers would not be positive. Thus, the call `IERC20(tracerBaseToken).safeTransfer(msg.sender, amount)` will be made again. The user could then take control of the execution and call `withdraw()` a third time and then a fourth and so on until they have drained their entire `userBalance.deposited` after which there would be a subtraction underflow preventing further recursions.

Note that the likelihood of this vulnerability is low as the majority of ERC20 tokens to do render control of execution to arbitrary addresses during a call to `transfer()`.

Recommendations

To prevent reentrancy it is recommended to first perform all state updates before making any external calls to contracts which may be outside the protocols control. That is the line `IERC20(tracerBaseToken).safeTransfer(msg.sender, amount)` should occur after all state modifications.

See TCR-27 for further details.

TCR-16	Insufficient Timing Parameter Validation		
Asset	`Gov.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The Tracer governance contract uses various timing-related variables to enforce a timeline between the different stages of a proposal (`warmUp` , `coolingOff` , `proposalDuration` and `lockDuration`).

These parameters can all be changed by participants of the protocol's governance. There are no checks enforced on the value of these parameters: they can all be set to 0 for example, rendering the governance unusable (for example, any subsequent proposal would then have `newProposal.startTime` = `newProposal.expiryTime` = block.timestamp)

Furthermore, as outlined in TCR-12, implicit assumptions are made regarding the relationship between these time variables. These relationships are not enforced when changing the value of these parameters.

Recommendations

Consider expressing and explicitly enforcing the constraints between the timing parameters in the governance contract.

TCR-17	Insufficient Input Validation on `proposalId` in Voting Functions		
Asset	`Gov.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When voting on an non-existent proposal, the related transaction on the `Gov` contract does not revert. This results in the user's lock up period being increased by `lockDuration` (see line [258]), even though their related vote is not valid nor counted.

Mistakes could be made by genuine users and result in an unfair increase of their lockup periods.

Recommendations

Consider reverting the `vote` external call if the Id of the proposal submitted as part of a vote is greater than the number of existing proposals, i.e.:

```
require(proposalId <= proposalCounter)
```

TCR-18	Simple Majority Not Required to Pass Proposals		
Asset	`Gov.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The Tracer governance contract, `Gov.sol` considers a proposal successful (i.e. in a `PASSED` state) as soon as exactly 50% of the staked votes are in favour of the proposal.

This means that Tracer's governance does not require a strict, simple majority for proposals to be passed, as can be see on line [265] of the `Gov.sol` contract:

```
if (votes >= totalStaked.div(2)) {
```

As soon as this threshold is reached, no more votes can be cast by any participants. While this would be considered acceptable in the case of "No" votes, a proposal should attract **more** than half of the total votes to be deemed successful.

Recommendations

We recommend replacing line [265] in `Gov.sol` with a strict inequality where `votes > totalStaked.div(2)`.

Note: Because voting calculations are made based on the total stake of the contract, rather than the number of "yes" votes vs "no" votes, passing a proposal effectively requires an "absolute" majority, as the governance contract does not use quorums

TCR-19	Incorrect use of Absolute Value in Calculations		
Asset	LibBalances.sol		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The function `safeCalcTradeMargin()` is a helper function in determining if a user is able to make a order by calculating the users `base` and `quote` after applying the changes of the order.

The function `safeCalcTradeMargin()` incorrectly takes the absolute value of the price when calculating the change to the `base` as seen in the following.

```
int256 baseChange = (amount.mul(uint(price.abs()))).div(priceMultiplier).toInt256();
```

A negative price should change the base in the opposite direction to a positive price.

Recommendations

We recommend doing all calculations for `baseChange` as `int256` types without taking absolute values or converting to `uint256`.

TCR-20	TWAPS Exclude Hours Where Price is Zero		
Asset	`Pricing.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The Time Weighted Average Price (TWAP) is a weighted average of the prices over the previous eight (8) hours, with the most recent hour having a weighting of eight (8) and the oldest hour a weighting of one (1).

An hour is excluded from the calculations if it has price of zero (0), as seen in the following code.

```
int256 derivativePrice = getHourlyAvgTracerPrice(uint256(j), market);
int256 underlyingPrice = getHourlyAvgOraclePrice(uint256(j), market);
if (derivativePrice != 0) {
    derivativeInstances = derivativeInstances.add(uint256(timeWeight));
    derivativeSum = derivativeSum.add((timeWeight).mul(derivativePrice));
}
if (underlyingPrice != 0) {
    underlyingInstances = underlyingInstances.add(uint256(timeWeight));
    underlyingSum = underlyingSum.add((timeWeight).mul(underlyingPrice));
}
```

However, a price of zero (0) may be a valid price for some markets and thus should not be ignored from the calculations.

Recommendations

We recommend modifying the check to be `pricing.hourlyTracerPrices[hour].numTrades == 0` `pricing.hourlyOraclePrices[hour].numTrades == 0` respectively to account for the zero (0) price.

TCR-21	Overwrite `orderIdByHash` for Duplicate Orders		
Asset	`SimpleDEX.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When orders are made they are given an `orderId` and a helper mapping `orderIdByHash` which takes the hash of the order and returns the `orderId`.

If two orders are made with the same `hashOrder()` value, then the value `orderIdByHash` will be overwritten.

Note to have an overlap in the `hashOrder()` output the orders must have the same `maker`.

Recommendations

Consider rejecting orders which already have an entry in `orderIdByHash`. If users wish to make another order of the same `price` and `amount` they are able to modify the expiration.

TCR-22	Insertion of Expired Orders		
Asset	`SimpleDEX.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There are no sanity checks on the value `expiration` of an order.

As a result it is possible for a user to submit expired orders. These orders are never able to be matched since they are expired.

Recommendations

Consider adding some sanity checks to enforce users to submit orders that have an expiration time greater than the current timestamp.

TCR-23	Matching Filled Orders		
Asset	`SimpleDEX.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `_matchOrder()` allows two orders to be matched together and executed. An order is considered filled when `order.filled` is set to `order.amount`.

During the process of matching two orders the amount to be filled is the minimum remaining of these two orders. When one order is filled the remaining amount is zero (0) and thus the orders will be executed for zero amount.

This would allow users to manipulate the TWAP price by executing already filled orders repeatedly.

Recommendations

Consider adding a check to `_matchOrder()` to ensure `fillAmount` is strictly greater than zero.

TCR-24	Potential Contract Configuration Mismatch		
Asset	`Account.sol` & `Tracer.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Each `Tracer` will have a `Pricing` and `Account` contract. The `Pricing` contract is used to determine the funding rate and the `Account` contract maintains the token balances for the markets. The `Account` contact stores the address of the `Pricing` contract, which is required to perform calculations.

When a `Tracer` is deployed the `Account` and `Pricing` contracts may be set in the constructor and stored in the variables `Tracer.pricingContract` and `Tracer.accountContract`. Similarly, when an `Account` contract is deployed it sets the `Account.pricing` variable.

There are no requirements for `Account.pricing` to match `Tracer.pricingContract`. The impact would be that the calculations of the funding rate and fair prices would different significantly between the `Tracer` and `Account`.

This issue would occur if a user wished to re-use the `Account` contract from one market but set its own `Pricing` contract.

Recommendations

The potential mismatch between pricing contracts may be prevent by any of the following:

- Retrieving the pricing contract from the `Tracer` each time it is used in `Account`
- Caching pricing contracts as a map mapping each `Tracer` to a `Pricing` contract in `Account`
- In the construction of the `Tracer` enforcing `accountContract.pricing == _pricingContract`

TCR-25	Partially Implemented `feeRate`		
Asset	`Tracer.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The variable `feeRate` has only been implemented in the function `makeOrder()`. This variable is not used in the functions `takeOrder()` and `matchOrders()`

The impact is that users are not being charged the allocated fee when taking or matching orders.

Recommendations

We recommend either adding `feeRate` to the calculations in `takeOrder()` and `matchOrders()`. Alternatively, remove `feeRate` if it is not needed.

TCR-26	Tracer constructor parameters
Asset	`Tracer.sol`
Status	Open
Rating	Informational

Description

The `Tracer` contract takes a set of variables in the constructor which are required for the market to function. We note a few observations with these parameters here.

Firstly, a `Tracer` contract allows any user (via the `TracerFactory` contract) to choose their own `Account` contract. This means that malicious users can have malicious tracers with arbitrary malicious account contracts. Fortunately, the global `Insurance` contract has its own `Account` contract set and doesn't rely on the account contract set in the tracer. The `Insurance` contract will still deem such a malicious tracer as valid (as it was deployed from the `TracerFactory`). It is dangerous having contracts such as the `Insurance` contract running arbitrary code (by calling tracer functions which call arbitrary external contracts) and this may not be desired by the developers.

Secondly, the `insuranceContract` is required for contract functionality but is not set in the constructor. As a result users will not be able to take or match orders until the owner has called `setInsuranceContract()` to be an address which implements the `IInsurance` interface.

Recommendations

Ensure the risk profile here is as expected. It is more secure to inject the account contract in the deployment of the tracer by sending it to the deployer (so that arbitrary tracer's can still be deployed) and having the deployer inject the account contract. This is however more restrictive for users deploying custom tracers.

It may be beneficial to reduce the number of required transactions and likelihood of user error by modifying the constructor to take the address of the `insuranceContract`.

TCR-27	Checks-Effects-Interaction Pattern
Asset	`contracts/`
Status	Open
Rating	Informational

Description

Re-entrancy [4] is a common solidity attack that can occur when contracts perform external calls to other user-defined arbitrary contracts. The checks-effects-interactions pattern [5] is a recommended solidity programming pattern to use where possible. Developers should perform all necessary checks, followed by state changes and finally perform all external calls at the end of the function.

Tracer has a number of functions that could theoretically (but with low likelihood) be effected by re-entrancy, an example is in TCR-15.

The following functions do not follow the checks effects interaction pattern:

- Account.sol - `deposit()`
- Account.sol - `withdraw()`
- Insurance.sol - `stake()`
- Insurance.sol - `withdraw()`
- Insurance.sol - `drainPool()`
- Insurance.sol - `updatePoolAmount()`

As an example in `Insurance.sol`

```
        // Pool tokens become margin tokens
        poolToken.burn(msg.sender, amount);
        token.safeTransfer(msg.sender, tokensToSend);
        pool.amount = pool.amount.sub(tokensToSend);
```

The storage change `pool.amount` occurs after the external call.

Recommendations

We recommend all functions (where possible) follow the checks-effects-interaction pattern to minimize the re-entrancy attack surface in these functions.

TCR-28	Unnecessary `calcEscrowLiquidationAmount()` function
Asset	`Account.sol`
Status	Open
Rating	Informational

Description

The `calcEscrowLiquidationAmount()` function returns simply - `getUserMinMargin()` or 0 if this value is less than 0. We raise this as its own separate issue to ensure the calculation is as expected.

The function is:

```
function calcEscrowLiquidationAmount(
    address liquidatee,
    int256 currentUserMargin,
    address market
) internal view returns (uint256) {
    int256 minMargin = getUserMinMargin(liquidatee, market);
    int256 amountToEscrow = currentUserMargin.sub(minMargin.sub(currentUserMargin));
    if (amountToEscrow < 0) {
        return 0;
    }
    return uint256(amountToEscrow);
}
```

The `currentUserMargin` gets subtracted from itself and therefore is an unnecessary parameter for the function.

Recommendations

Ensure the calculation in this function is desired. If so, consider either removing the function entirely and using simply `getUserMinMargin` or removing the effectively unused `currentUserMargin` parameter which currently simply serves to cost extra gas in the function call.

TCR-29	Lack of Limitations/Validation To Set Variables in `Tracer`
Asset	`Tracer.sol`
Status	Open
Rating	Informational

Description

The owner of a `Tracer` is able to set a range of variables which effect user rewards and borrowing functionality.

The functions that do not contain any limitations or validations are:

- `setFeeRate()`
- `setMaxLeverage()`
- `setFundingRateSensitivity()`

It is also possible to set the addresses to the zero address (`0x0`) for each of external contracts. That is:

- `setInsuranceContract()`
- `setAccountContract()`
- `setPricingContract()`
- `setOracle()`
- `setGasOracle()`

Recommendations

Consider adding some reasonable upper and lower bounds where feasible to the setter functions, such as `require(feeRate <= 20%)` and `0 <= maxLeverage < 10x` .

Historically, it has been the case that when nothing has been entered into a UI when interacting with a smart contract address field, the `0x0` address has been sent as a place holder. For this reason we typically recommend safe guarding against users inadvertently entering the `0x0` address.

TCR-30	User Created Markets Can Be Exploited By Owner
Asset	`Tracer.sol`
Status	Open
Rating	Informational

Description

The owner of a `Tracer` market is set when it is create by the `TracerFactory`. Arbitrary users have the ability to create a `Tracer` and set the owner to an arbitrary address. These user generated markets are *not* marked as DAO approved.

The owners of markets have the ability to call specific `onlyOwner` functions such as

- `setInsuranceContract()`
- `setAccountContract()`
- `setPricingContract()`
- `setOracle()`
- `setGasOracle()`
- `setFeeRate()`
- `setMaxLeverage()`
- `setFundingRateSensitivity()`

This gives the owner power to make changes to the market that directly increase the owner's margin.

For example an owner may take all the funds from the market by doing the following steps:

1. Take an order with a long position
2. Call `Tracer.setOracle()` to one an address the returns the price 1,000,000,000x the real value
3. Call `Account.withdraw()` taking the entire market balance (`Account.tvl[market]`)

Recommendations

The development team are aware of risks of this design. The variable `TracerFactory.daoApproved` is used to differentiate user generate markets from DAO owned markets where this attack is not feasible.

It should be made clear to users the potential to steal funds by owners in user generated markets.

TCR-31	Zero Address Passes `verifySignature()`
Asset	`Trader.sol`
Status	Open
Rating	Informational

Description

The function `verifySignature()` is used to ensure that a user has signed a specific message.

The function consists of the following statement,

```
return signer == ecrecover(hashOrder(order), sigV, sigR, sigS);
```

If `ecrecover()` receives an invalid signature it will return the zero address.

As a result, if `signer` is set to the zero address and an invalid signature is given to `verifySignature()` it will return true.

The impact is informational as the zero address is not owned and so the `Trader` would not have permission to act on behalf of the zero address and calls to `executeTrade` will fail.

Recommendations

This issue may be mitigated by adding a check to `verifySignature()` to ensure that the signer is not the zero address.

TCR-32	Front Running
Asset	`Trader.sol` & `Tracer.sol`
Status	Open
Rating	Informational

Description

Ethereum transactions are able to be read in the mempool before they are executed. Users are able to see these transaction then create their own transaction with a higher gas price that will likely be executed first.

Orders that are being made or taken in `Tracer` are vulnerable to front-running as bots may first see these orders then execute their own orders potentially arbitraging two trades.

Front-running can be applied to perform a Denial of Service (DoS) on `Trader.executeTrade()`. A user can cause an entire `executeTrade()` transaction to revert by executing their own transactions first. Possible transactions users can do to cause `executeTrade()` to revert include:

- Have one user remove permission for the `Trader` in the `Tracer`
- Call `executeTrade()` from one of the accounts to increase their nonce
- Withdraw funds from the `Tracer` such that the margin is no longer valid.

Recommendations

Ensure this is the most desirable behaviour, otherwise consider skipping a pair of trades rather than reverting if one of them fails.

TCR-33	Trades can be Executed with Different Amounts
Asset	`Trader.sol`
Status	**Open**
Rating	Informational

Description

The function `executeTrade()` can be called on two orders which have a different amount. The result is that the larger trade will be executed but not entirely filled. It will remain open on the `Tracer` market for other users to trade with.

Recommendations

Ensure this is desirable behaviour, otherwise add a check to verify that the amounts of each pairs of orders are the same.

TCR-34	Orders Can Be Created That Can Never Be Filled
Asset	`Tracer.sol`
Status	**Open**
Rating	Informational

Description

Users can create orders, but they do not effect their balances until another party takes the other side of the order. This means users can create orders, then withdraw all their deposit leaving orders which cannot be taken (as the creating user's margin will fall too low).

We raise this to be aware that there can be many unusable active orders listed on a tracer market.

Recommendations

Ensure this is the desired behaviour.

A system of closing open orders when withdrawing an entire balance could be enforced, should this behaviour not be expected.

TCR-35	Unused Variables & Functions
Asset	`contracts/`
Status	Open
Rating	Informational

Description

The variable `DIVIDE_PRECISION` is not used internally in `Account.sol` and `Pricing.sol` nor by any other contracts.

The variable `Account.currentLiquidationId` is not used. Note this is not to be confused with `Receipt.currentLiquidationId` which is used.

The function `Account.updateAccount()` can only be called by a `Tracer` which does not make this call. Hence, the function is unused.

In `LibBalances.sol` the variable `FEED_UNIT_DIVIDER` is unused in addition to the functions `calcLeveragedNotionalValue()` and `calcMarginPositionValue()`.

Recommendations

We recommend removing the unused variables and functions to improve code usability and maintainability.

TCR-36	Miscellaneous General Statements
Asset	`contracts/`
Status	Open
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment that do not have direct security implications:

- `Insurance.sol` line [158] - `difference` can just be set to `10**18`.
- `InsurancePoolToken.sol` - line [54] - This can be called by anyone and stakers have an incentive to set the maximum possible amount. It would make it easier to use and more intuitive to rename the function to `registerNewDeposits()` without a parameter. The function would account for all new unregistered deposits.
- `Account.sol` `_deposit()` does not require a `depositor` parameter, `msg.sender` is always sent to it.
- `Insurance.sol` the value `pools[market].market` is unnecessary.
- `Insurance.sol` line [276] is unnecessarily blank.
- `Pricing.sol` the values `IPoolFundingRateValue` and `IPoolFundingRate` should both begin with a lower case 'i' to conform with solidity style guides.
- `Pricing.sol` the variable `uint256 hour` is cast to the same type `uint256(hour)` twice in each of the functions `getHourlyAvgTracerPrice()` and `getHourlyAvgOraclePrice()`.
- `Tracer.sol` the event `OrderFilled` field `amount` is set the function parameter `amount` rather than `filledAmount`.
- `Tracer.sol` the variable `gasCost` on line [156] is actually the gas price not gas cost (gas cost = gas price * gas used).
- `Tracer.sol` the variable `FUNDING_RATE_SENSITIVITY` is not a constant and should be camel-case.
- `Tracer.sol` `marketIds` for tracer's do not need to be unique and so can emit similar logs.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Maximum Withdrawal Observations

As part of this review, the team performed some simple analysis on the Solidity functions that allow users to withdraw tokens from their balance. Specifically, we calculated the maximum amount of base tokens a user can withdraw (which pass the `account.marginIsValid()` function).

The purpose of this was to ensure that Solidity was internally consistent in that users could not withdraw more than the sum of their deposits as the price fluctuates. This was also used to visualise how the margin lending shifts as a function of price.

The function used to calculate the maximum amount of tokens a user can withdraw, as well as producing the following graphs is contained within `graph/simple_graph.py` that accompanies this report.

The following two examples are useful for visualizing how the current protocol allows withdrawals between trades taken at positive and negative prices change as the fair price fluctuates.

For two users, (a "long" and short "user" who take the long and short sides of a trade respectively) the following shows the maximum base tokens (as a percentage of their original deposit/trade) they can withdraw as the price fluctuates from their trade price.

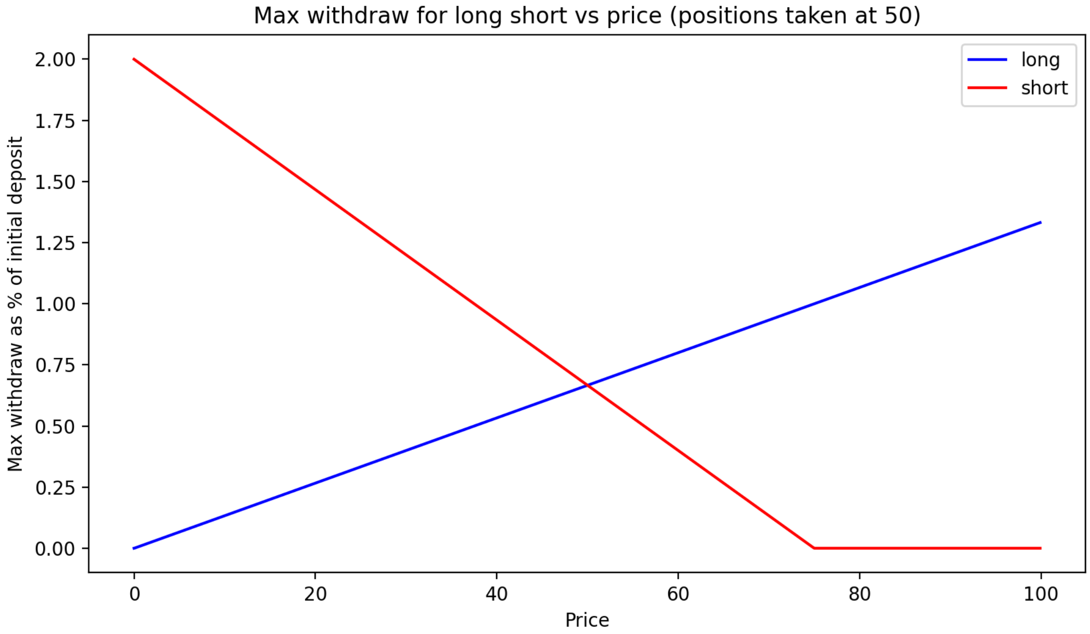

Figure 1: Maximum tokens that can be withdrawn for long and short users who accept a trade at a price of 50. The maximum leverage is set to 3.

In this example, at the trade price, both users cannot withdraw more than their initial deposit. However as the price changes in the short user's favour (tends to 0) the short user can withdraw more than their initial balance. At 0, the short user can withdraw 2 times their initial balance but the long user can withdraw nothing, making this internally consistent.

It could of course be the case that the short withdraws 2x their initial balance and the price shifts back toward the trade price. In this case, the long user could entitled to more tokens than exist in the contract, however is the purpose of the liquidation functionality of the contract, where an external user is expected to liquidate the short users position.

The long user could also liquidate the short users position, in effect negating their trade.

It is interesting to compare the same function against users who trade at a negative price.

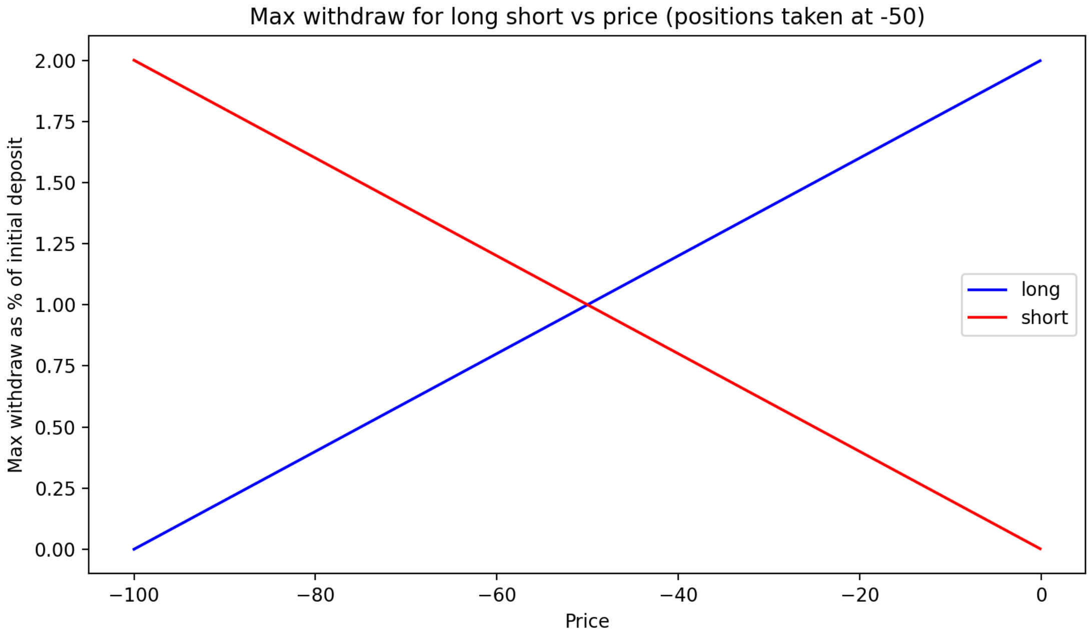

Figure 2: Maximum tokens that can be withdrawn for long and short users who accept a trade at a price of -50. The maximum leverage is set to 3.

Immediately there is an issue here in that at the trade price, both users are able to withdraw their initial deposits, which allows them to do risk-free trades. The limit in withdrawing in a negative price range comes down to keeping the margin positive rather than maintaining the required leverage ratio (as is the case in the positive price case), making the gradients between the long and short symmetric. We expect this is not the desired behaviour of the protocol.

Appendix B Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_deposit                                    PASSED  [1%]
test_deposit_to                                 PASSED  [2%]
test_deposit_malicious                          PASSED  [3%]
test_withdraw                                   PASSED  [5%]
test_withdraw_malicious                         PASSED  [6%]
test_withdraw_reenterancy                       XFAIL   [7%]
test_set_receipt_contract                       PASSED  [8%]
test_set_insurance_contract                     PASSED  [10%]
test_set_gas_price_oracle                       PASSED  [11%]
test_set_factory_contract                       PASSED  [12%]
test_set_pricing_contract                       PASSED  [14%]
test_only_tracer                                PASSED  [15%]
test_is_valid_tracer                            PASSED  [16%]
test_setup_and_deploy                           PASSED  [17%]
test_order_withdraw                             PASSED  [19%]
test_basic_make_take                            PASSED  [20%]
test_basic_make_take_withdraw                   PASSED  [21%]
test_basic_make_take_high_price                 PASSED  [23%]
test_negative_price                             PASSED  [24%]
test_multiple_orders                            PASSED  [25%]
test_price_change_liquidation                   PASSED  [26%]
test_deploy                                     PASSED  [28%]
test_stake                                      PASSED  [29%]
test_delegate                                   PASSED  [30%]
test_propose                                    PASSED  [32%]
test_double_vote                                XFAIL   [33%]
test_withdraw                                   FAILED  [34%]
test_withdraw_double_vote                       XFAIL   [35%]
test_transfer_attack_insurance_pool             PASSED  [37%]
test_transfer_from_insurance_pool               PASSED  [38%]
test_rewards_per_token                          PASSED  [39%]
test_deposit_tokens_to_pool                     PASSED  [41%]
test_negative_prices                            PASSED  [42%]
test_negative_prices                            PASSED  [43%]
test_negative_prices_leveraged_withdraw_deposit PASSED  [44%]
test_make_order                                 PASSED  [46%]
test_permissioned_make_order                    PASSED  [47%]
test_make_order_same_order_twice                PASSED  [48%]
test_make_order_zero_deposits                   XFAIL   [50%]
test_make_order_zero_amount                     XFAIL   [51%]
test_make_order_expired                         XFAIL   [52%]
test_make_order_at_max_leverage                 PASSED  [53%]
test_take_order                                 PASSED  [55%]
test_take_order_expired                         PASSED  [56%]
test_take_order_filled                          PASSED  [57%]
test_take_order_invalid_margin                  PASSED  [58%]
test_take_order_zero_deposit                    PASSED  [60%]
test_match_orders                               PASSED  [61%]
test_match_orders_same_side                     PASSED  [62%]
test_match_orders_different_price               PASSED  [64%]
test_match_orders_expired                       PASSED  [65%]
test_match_orders_filled_orders                 XFAIL   [66%]
test_match_orders_invalid_margin                PASSED  [67%]
test_update_internal_records                    PASSED  [69%]
test_set_user_permissions                       PASSED  [70%]
test_set_insurance_contract                     PASSED  [71%]
test_set_account_contract                       PASSED  [73%]
test_set_pricing_contract                       PASSED  [74%]
test_set_oracle                                 PASSED  [75%]
test_set_gas_oracle                             PASSED  [76%]
test_set_fee_rate                               PASSED  [78%]
```

```
test_set_max_leverage                            PASSED  [79%]
test_set_funding_rate_sensitivity                PASSED  [80%]
test_transfer_ownership                          PASSED  [82%]
test_set_insurance_contract                      PASSED  [83%]
test_set_deployer_contract                       PASSED  [84%]
test_set_approved                                PASSED  [85%]
test_set_approved_bad_owner                      PASSED  [87%]
test_deploy_tracer                               PASSED  [88%]
test_deploy_tracer_and_approve                   PASSED  [89%]
test_execute_trade                               PASSED  [91%]
test_execute_trade_invalid_signature             PASSED  [92%]
test_execute_trade_invalid_nonce                 PASSED  [93%]
test_execute_trade_array_lengths                 PASSED  [94%]
test_execute_trade_empty_arrays                  PASSED  [96%]
test_execute_trade_different_tracer              XFAIL   [97%]
test_verify_signature_zero_address               XFAIL   [98%]
test_execute_trade_zero_address                  PASSED  [100%]
```

Appendix C Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Ryan Garner, Lachlan Webb, Jason Potts, Chris Berg, and Sinclair Davidson. Tracer Perpetual Swaps, Available: https://tracer.finance/media/whitepapers/perp-swaps/Tracer_Perpetual_Swaps.pdf.

[2] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[3] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[4] Sigma Prime. Solidity Security - Reentrancy. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#reentrancy. [Accessed 2018].

[5] Solidity. Solidity - checks effects interactions. Read the docs, 2018, Available: https://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern. [Accessed 2018].