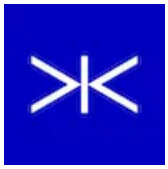

Tracer
Findings & Analysis Report

2021-09-16

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the Tracer smart contract system written in Solidity. The code contest took place between June 23—June 30, 2021.

Wardens

12 Wardens contributed reports to the Tracer code contest:

- 0xsanson
- shw
- 0xRajeev
- pauliax
- gpersoon
- cmichel
- a_delamo
- Jmukesh
- Lucius
- s1m0
- tensors
- hrkrshnn

This contest was judged by cemozerr.

Final report assembled by ninek and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 43 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity, 13 received a risk rating in the category of MEDIUM severity, and 24 received a risk rating in the category of LOW severity.

C4 analysis also identified 32 non-critical recommendations.

Scope

The code under review can be found within the C4 Tracer code contest repository is comprised of 44 smart contracts written in the Solidity programming language and includes 2,453 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings

[H-01] Wrong trading pricing calculations

Submitted by 0xsanson, also found by shw

In the `Pricing` contract, an agent can manipulate the trading prices by spamming a high amount of trades.

Indeed an agent can create a high amount of orders at an arbitrary price and with a near-zero amount (so the agent doesn't even need large funds); next he/she pairs

the orders with another account and calls `Trader.executeTrade` ; now every order calls a `Pricing.recordTrade` using the arbitrary price set by the agent.

Since the trades are all made in the same hour, by the way `hourlyTracerPrices[currentHour]` is calculated, it skews the average price towards the price set by the agent. This arbitrary value is used to calculate the `fundingRates` and the `fairPrice` , allowing a malicious agent the ability to manipulate the market.

Recommend passing the `fillAmount` parameter to `recordTrade(...)` , and calculate `hourlyTracerPrices[currentHour].trades` summing `fillAmount` instead of 1 every trade.

raymogg (Tracer) confirmed:

> Issue is valid, and there appear to be a few other issues that reference similar problems.

> The Trader contract will have a whitelist allowing only select relayers to push orders on chain. As long as off chain order books have sufficient liquidity, this issue is then mitigated as users can't just arbitrarily match orders and send them in, they must be matched on a book with liquidity. To alter the price you would then need to eat through significant liquidity (increasing the cost of this attack).

[H-02] Use of incorrect index leads to incorrect updation of funding rates

Submitted by 0xRajeev

The `updateFundingRate()` function updates the funding rate and insurance funding rate. While the instant/new funding rates are calculated correctly, the cumulative funding rate calculation is incorrect because it is always adding the instant to 0, not the previous value. This is due to the use of `[currentFundingIndex]` which has been updated since the previous call to this function while it should really be using `[currentFundingIndex-1]` to reference the previous funding rate.

The impact of this, is that the cumulative funding rate and insurance funding rates are calculated incorrectly without considering the correct previous values. This affects the settling of accounts across the entire protocol. The protocol logic is significantly impacted, accounts will not be settled as expected, protocol shutdown and contracts will need to be redeployed. Users may lose funds and the protocol takes a reputation hit.

Recommend using `[currentFundingIndex-1]` for non-zero values of `currentFundingIndex` to get the value updated in the previous call on lines L155 and L159 of `Pricing.sol`.

raymogg (Tracer) confirmed:

> Confirmed as an index issue with funding rate 👍

[H-03] Malicious owner can drain the market at any time using `SafetyWithdraw`

Submitted by 0xRajeev, also found by pauliax and gpersoon

The `withdrawERC20Token()` in `SafetyWithdraw` inherited in `TracerPerpetualSwaps` is presumably a guarded launch emergency withdrawal mechanism. However, given the trust model where the market creator/owner is potentially untrusted/malicious, this is a dangerous approach to emergency withdrawal in the context of guarded launch.

Alternatively, if this is meant for the owner to withdraw "external" ERC20 tokens mistakenly deposited to the Tracer market, then the function should exclude `tracerQuoteToken` from being the `tokenAddress` that can be used as a parameter to `withdrawERC20Token()`.

The impact of this is that, if a malicious owner of a market withdraws/rugs all `tracerQuoteToken`s deposited at any time after market launch, all users lose deposits and the protocol takes a reputational hit and has to refund the users from treasury.

Therefor, it is recommended that, for a guarded launch circuit breaker, design a pause/unpause feature where deposits are paused (in emergency situations) but withdrawals are allowed by the depositors themselves instead of the owner. Alternatively, if this is meant to be for removing external ERC20 tokens accidentally deposited to market, exclude the `tracerQuoteToken` from being given as the `tokenAddress`.

raymogg (Tracer) confirmed but suggested a severity of 2 :

> The only reason for the dispute on severity is that as part of the security model, the owner can manipulate the market in other ways (such as changing the oracle being used), so this trust assumption over the owner already exists. For this reason the team thinks this issue is closer to a medium

> This however is a good issue as it is not the greatest circuit breaking mechanism, and as noted in #7 can reflect badly on the project without the exploit being used. The mechanism is being removed and replaced with more structured circuit breaker.

cemozerr (Judge) commented:

> Marking this as high risk, as regardless of the owner manipulating in other ways, the threat persists.

[H-04] Logic error in fee subtraction

Submitted by 0xsanson

In `LibBalances.applyTrade()`, we need to collect a fee from the trade. However, the current code subtracts a fee from the short position and adds it to the long. The correct implementation is to subtract a fee to both (see `TracerPerpetualSwaps.sol` L272). This issue causes withdrawals problems, since Tracer thinks it can withdraw the collect fees, leaving the users with an incorrect amount of quote tokens.

Recommend changing `+fee` to `-fee` in the highlighted line.

raymogg (Tracer) confirmed:

> Valid issue 👍

[H-05] Insurance slippage reimbursement can be used to steal insurance fund

Submitted by cmichel

The `Liquidation` contract allows the liquidator to submit "bad" trade orders and the insurance reimburses them from the insurance fund, see `Liquidation.claimReceipt`. The function can be called with an `orders` array, which does not check for duplicate orders. An attacker can abuse this to make a profit by liquidating themselves, making a small bad trade and repeatedly submitting this bad trade for slippage reimbursement.

Example:

- Attacker uses two accounts, one as the liquidator and one as the liquidatee.
- They run some high-leverage trades such that the liquidatee gets liquidated with the next price update. (If not cash out and make a profit this way through trading, and try again.)
- Liquidator liquidates liquidatee
- They now do two trades:
    - One "good" trade at the market price that fills 99% of the liquidation amount. The slippage protection should not kick in for this trade
    - One "bad" trade at a horrible market price that fills only 1% of the liquidation amount. This way the slippage protection kicks in for this trade
- The liquidator now calls `claimReceipt(orders)` where `orders` is an array that contains many duplicates of the "bad" trade, for example 100 times. The `calcUnitsSold` function will return `unitsSold = receipt.amountLiquidated` and a bad `avgPrice`. They are now reimbursed the price difference on the full liquidation amount (instead of only on 1% of it) making an overall profit

This can be repeated until the insurance fund is drained.

The attacker has an incentive to do this attack as it's profitable and the insurance fund will be completely drained.

Recommend disallowing duplicate orders in the `orders` argument of `claimReceipt`. This should make the attack at least unprofitable, but it could still be a griefing attack. A quick way to ensure that `orders` does not contain duplicates is by having liquidators submit the orders in a sorted way (by order ID) and then checking in the `calcUnitsSold for` loop that the current order ID is strictly greater than the previous one.

BenjaminPatch (Tracer) confirmed:

> Valid issue. The recommended mitigation step would also work. 👍

[H-06] Wrong price scale for `GasOracle`

Submitted by cmichel

The `GasOracle` uses two chainlink oracles (GAS in ETH with some decimals, USD per ETH with some decimals) and multiplies their raw return values to get the gas price in USD.

However, the scaling depends on the underlying decimals of the two oracles and could be anything. But the code assumes it's in 18 decimals.

> "Returned value is USD/Gas * 10^18 for compatibility with rest of calculations"

There is a `toWad` function that seems to involve scaling but it is never used.

The impact is that, If the scale is wrong, the gas price can be heavily inflated or under-reported.

Recommend checking `chainlink.decimals()` to know the decimals of the oracle answers and scale the answers to 18 decimals such that no matter the decimals of the underlying oracles, the `latestAnswer` function always returns the answer in 18 decimals.

raymogg (Tracer) confirmed and disagreed with severity:

> Disagree with severity as while the statement that the underlying decimals of the oracles could be anything, we will be using production Chainlink feeds for which the decimals are known at the time of deploy.

> This is still however an issue as you don't want someone using different oracles (eg non Chainlink) that have different underlying decimals and not realising that this contract will not support that.

cemozerr (Judge) commented:

> Marking this a high-risk issue as it poses a big threat to users deploying their own markets

Medium Risk Findings

[M-01] Use of deprecated Chainlink API

*Submitted by 0xRajeev, also found by a*delamo, cmichel and shw_

The contracts use Chainlink's deprecated API `latestAnswer()`. Such functions might suddenly stop working if Chainlink stopped supporting deprecated APIs.

The impact is that, if the deprecated API stops working, prices cannot be obtained, the protocol stops and contracts have to be redeployed.

Recommend using V3 interface functions.

raymogg (Tracer) confirmed in a separate issue

[M-02] No check `transferFrom()` return value

Submitted by s1m0, also found by pauliax, shw, 0xRajeev, JMukesh, Lucius and cmichel

The smart contract doesn't check the return value of `token.transfer()` and `token.transferFrom()`, some erc20 token might not revert in case of error but

return false. In the TracerPerpetualSwaps:deposit and Insurance:deposit this would allow a user to deposit for free. See issue page for other places.

Recommend wrapping the call into a `require()` or using openzeppelin's SafeERC20 library.

raymogg (Tracer) confirmed

[M-03] Deflationary tokens are not supported

Submitted by cmichel, also found by s1m0 and 0xRajeev

There are ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`.

The `deposit()` functions of `Insurance` and `TracerPerpetualSwaps` assume that the external ERC20 balance of the contract increases by the same amount as the `amount` parameter of the `transferFrom`.

The user is credited the full amount without the taxes (`userBalance.position.quote`).

Recommend as one possible mitigation, measuring the asset change right before and after the asset-transferring functions.

raymogg (Tracer) confirmed but disagreed with severity:

> Most likely not a medium risk as you can do a lot more nasty things than just use rebasing tokens. Since the owner of a market can set their own quote token, this token could be a token they control the supply of allowing them to arbitrarily transfer tokens between accounts, etc.

> As such, this sort of falls outside of our trust model. Market creators should use tokens that behave as "standard" ERC20s. We will make a not that rebasing and deflationary tokens should not be used as quote tokens without weird behaviour.

> Would be better as a low or informational issue due to this.

cemozerr (Judge) downgraded severity from 2 to 1:

> Marking this as low risk as it seems to fall outside of the trust model, yet important enough to communicate to users explicitly.

[M-04] Underflow problems occurring when a token has >18 decimals

Submitted by tensors, also found by s1m0

The contracts assume that all tokens will have <=18 decimals. This isn't necessarily a problem if the Tracer team is the only people deploying the contracts and they keep it in mind. But, If the contracts are to be deployed by other people, this assumption should be made explicit and hard-coded.

We can see that the scaler computations will underflow and be defined when it should not be In L220-L232.

Recommend writing a require check that ensures `tokenDecimals <= 18` before running the above functions.

raymogg (Tracer) confirmed:

> Valid issue and makes sense as a medium.

> Suggested mitigation will be implemented.

[M-05] Add reentrancy protections on function `executeTrade`

Submitted by shw, also found by 0xRajeev

As written in the to-do comments, reentrancy could happen in the `executeTrade` function of `Trader` since the `makeOrder.market` can be a user-controlled external contract. See L121-L126 in `Trader.sol`.

Recommend adding a reentrancy guard (e.g., the implementation from OpenZeppelin) to prevent the users from reentering critical functions.

raymogg (Tracer) disputed:

> Disputing just as while this is important, its quite explicitly stated in the todo comment and as such is already known by the team as a potential issue.

> Realistically shouldn't be too much of a problem with whitelisting of the trader.

cemozerr (Judge) commented:

> Marking this as medium risk as, regardless of being noted by the team, still poses a security threat.

OsmanBran (Tracer) commented:

> Duplicate of #72

[M-06] Single-step process for critical ownership transfer

Submitted by 0xRajeev

The `Tracer Perpetuals Factory` contract is arguably the most critical contract in the project given that it deploys all the markets. The ownership of this contract is transferred to `_governance address` , i.e. TracerDAO, in the constructor. This critical address transfer in one-step is very risky because it is irrecoverable from any mistakes.

The impact is that, if an incorrect address (e.g. one for which the private key is not known) is used accidentally, then it prevents the use of all the `onlyOwner()` functions forever, which includes the changing of various deployer contract addresses and market approvals. This use of an incorrect address may not even be immediately apparent given that these functions are probably not used immediately. When noticed, due to a failing `onlyOwner()` function call, it will force the redeployment of the `factory` contract and require appropriate changes and notifications for switching from the old to new address. This will diminish trust in markets and incur a significant reputational damage. See issue page for proof of concept.

Recommend retaining the deployer ownership in the constructor and then using a two-step address change to `_governance` address separately using setter functions:

1. Approve a new address as a `pendingOwner`
2. A transaction from the `pendingOwner` (TracerDAO) address claims the pending ownership change.

This mitigates risk because if an incorrect address is used in step (1), then it can be fixed by re-approving the correct address. Only after a correct address is used in step (1) can step (2) happen and complete the address/ownership change.

raymogg (Tracer) acknowledged:

> Correct that having the owner be set to a wrong address could be detrimental, however for the first deploy of the factory, this will be owned by the DAO and will be easy to validate on deployment.

> Subsequent ownership transfers will be done via DAO proposal, and will have many eyes across them (due to them being a public Tracer DAO proposal) before function execution happens.

> For this reason it seems like a lot of overhead to have a two step process for this. Not withstanding that the issue you mention could still be possible

[M-07] Malicious owner can arbitrarily change fee to any % value

Submitted by 0xRajeev

The Tracer protocol like any other allows market creators to charge fees for trades. However, a malicious/greedy owner can arbitrarily change fee to any % value and without an event to observe this change or a timelock to react, there is no easy way for users to monitor this via front-end or off-chain monitoring tools.

The impact is that, if the users are trading on a market with 0.1% fees and the owner suddenly changes this to 100%, the users realise this only after their trades are executed. Market loses confidence. Protocol takes a reputational hit.

See similar Medium-severity finding in ConsenSys's Audit of 1inch Liquidity Protocol

Recommend implementing an `Emit` event, and providing a timelock for users to react and establish an upper threshold for fees that is decided across markets by governance.

raymogg (Tracer) confirmed:

> Like the idea of having a timelock for any update parameter update that immediately affects traders

[M-08] Missing events for critical parameter changing operations by owner

Submitted by 0xRajeev

The owner of `TracerPerpetualSwaps` contract, who is potentially untrusted as per specification, can change the market critical parameters such as the addresses of the `Liquidation` / `Pricing` / `Insurance` / `GasOracle` / `FeeReceiver` and also critical values such as `feeRate`, `maxLeverage`, `fundingRateSensitivity`, `deleveragingCliff`, `lowestMaxLeverage`, `insurancePoolSwitchStage` and whitelisting.

None of these setter functions emit events to record these changes on-chain for off-chain monitors/tools/interfaces to register the updates and react if necessary.

The impact of this is that, if a malicious owner changes the critical addresses or values that significantly change the security posture/perception of the protocol. No events are emitted and users lose funds/confidence. The protocol takes a reputation hit.

See similar high-severity finding in OpenZeppelin's Audit of Audius and medium-severity finding OpenZeppelin's Audit of UMA Phase 4.

Recommend to consider emitting events when these addresses/values are updated. This will be more transparent and it will make it easier to keep track of the status of the system.

raymogg (Tracer) marked as duplicate of another (confirmed issue):

> Duplicate of #66

cemozerr (Judge) reopened and removed duplicate label:

> Opening this issue as the event emission seems to be separate from the arbitrarily changing of the values.

[M-09] Wrong funding index in settle when no base?

Submitted by cmichel

The `TracerPerpetualSwaps.settle` function updates the user's last index to `currentGlobalFundingIndex`, however a comment states:

> "// Note: global rates reference the last fully established rate (hence the -1), and not the current global rate. User rates reference the last saved user rate"

The code for the `else` branch also updates the last index to `currentGlobalFundingIndex - 1` instead of `currentGlobalFundingIndex`.

```
    if (accountBalance.position.base == 0) {
        // set to the last fully established index
        // @audit shouldn't this be global - 1 like below?
        accountBalance.lastUpdatedIndex = currentGlobalFundingIndex;
        accountBalance.lastUpdatedGasPrice = IOracle(gasPriceOracle)
    }
```

The impact is that it might be possible for first-time depositors to skip having to pay the first funding rate period as the `accountLastUpdatedIndex + 1 < currentGlobalFundingIndex` check will still return `false` when the funding rates are updated the next time.

Recommend to check if setting it to `currentGlobalFundingIndex` or to `currentGlobalFundingIndex - 1` is correct.

raymogg (Tracer) confirmed but disagreed with severity

[M-10] `prb-math` not audited

Submitted by gpersoon

The library `prb-math` documents have not been audited by a security researcher. This means its more risky to rely on this library.

Recommend considering (crowdsourcing) an audit for `prb-math`.

raymogg (Tracer) confirmed

[M-11] Claim liquidation escrow

Submitted by gpersoon

A liquidator can always claim the liquidation escrow in the following way:

- create a second account
- setup a complimentary trade in that second account, which will result in a large slippage when executed
- call `executeTrade` (which everyone can call), to execute a trade between his own two accounts with a large slippage
- the slippage doesn't hurt because the liquidator owns both accounts
- call `claimReceipt` with the receiptId of the executed order, within the required period (e.g. 15 minutes)

L67

```
function executeTrade(Types.SignedLimitOrder[] memory makers, Ty
```

L394

```
function claimReceipt( uint256 receiptId, Perpetuals.Order[] mem
```

Recommend to perhaps limit who can call `executeTrade`.

raymogg (Tracer) acknowledged and confirmed:

> Valid issue which would allow someone to get reimbursed for slippage against themselves.

> The Trader contract will have whitelisted relayers added to prevent issues like this (similar to #119)

[M-12] avoid paying insurance

Submitted by gpersoon

It's possible to avoid paying insurance in the following way:

- once per hour (at the right moment), do the following:
- using a flash loan, or with a large amount of tokens, call `deposit` of `Insurance.sol` to make sure that the pool is sufficiently filled (`poolHoldings > poolTarget`)
- call the function `executeTrade` of Trader `.sol` with a minimal trade (possibly of value 0, see finding " `executeTrade` with same trades")
- `executeTrade` calls `matchOrders` , which calls `recordTrade`
- `recordTrade` calls `updateFundingRate()` ; (once per hour, so you have to be sure you do it in time before other trades trigger this)
- `updateFundingRate` calls `getPoolFundingRate`
- `getPoolFundingRate` determines the insurance rate, but because the insurance pool is sufficiently full (due to the flash loan), the rate is 0
- `updateFundingRate` stores the 0 rate via `setInsuranceFundingRate` (which is used later on to calculate the amounts for the insurances)
- withdraw from the Insurance and pay back the flash loan

The insurance rates are 0 now and no-one pays insurance. The gas costs relative to the insurance costs + the flash loan fees determine if this is an economically viable

attack. Otherwise it is still a grief attack. This will probably be detected pretty soon because the insurance pool will stay empty. However its difficult to prevent.

See issue page for code referenced in proof of concept.

Recommend setting a timelock on withdrawing insurance.

raymogg (Tracer) confirmed but disagreed with severity:

> Really like this exploit idea. Currently this is possible since the Trader is not whitelisted (eg there is no whitelisted relayer address). With this added, this exploit is no longer possible as only off chain relayers can place orders with the trader.

> Disagree with the severity mainly due to the fact that executing this exploit once would only cause insurance funding to not be paid for a single hour. For insurance funding to never be paid, you would have to time this transaction as the first transaction on each and every hour. This would quickly be noticed. The only affect on this would be insurance depositors miss interest payments for a few periods.

cemozerr (Judge) commented:

> Marking this as medium risk as a front-runner could keep doing this for not paying any funding using a bot.

[M-13] Trader orders can be front-run and users can be denied from trading

Submitted by cmichel, also found by gpersoon and tensors

The `Trader` contract accepts two signed orders and tries to match them. Once they are matched and become filled, they can therefore not be matched against other orders anymore.

This allows for a griefing attack where an attacker can deny any other user from trading by observing the mempool and front-running their trades by creating their own order and match it against the counter order instead.

In this way, a trader can be denied from trading. The cost of the griefing attack is that the trader has to match the order themselves, however depending on the liquidity of the order book and the spread, they might be able to do the counter-trade again afterwards, basically just paying the fees.

It could be useful if the attacker is a liquidator and is stopping a user who is close to liquidation from becoming liquid again.

This seems hard to circumvent in the current design. If the order book is also off-chain, the `executeTrade` could also be a bot-only function.

raymogg (Tracer) disputed (in duplicate)

> Marked as a dispute as this is not really an issue. Tracer will initially maintain an off chain order book that is the entry point for users to make orders (and for market makers to interact with).

> Orders only get propagated on chain once they have been matched, and they will only be propagated on chain by whitelisted relayers. As such nobody can arbitrarily frontrun the orders with their own.

cemozerr (Judge) commented:

> Currently not seeing a whitelisted relayer functionality, so marking this a valid medium risk issue.

Low Risk Findings

[L-01] Zero-address checks are missing

Submitted by 0xRajeev, also found by JMukesh, cmichel, gpersoon, pauliax and shw

Zero-address checks are a best-practice for input validation of critical address parameters. While the codebase applies this to most addresses in setters, there are many places where this is missing in constructors and setters. Accidental use of zero-addresses may result in exceptions, burn fees/tokens or force redeployment of contracts.

Recommend adding zero-address checks.

raymogg (Tracer) confirmed

> Duplicate of #136

> More issues brought up in this one, but falls under the general category of missing zero address checks

[L-02] Can set values to more than 100%

Submitted by cmichel

There are several setter functions that do not check if the amount is less than 100% including:

- `TracerPerpetualSwaps`: `setFeeRate`, `setDeleveragingCliff`, `setInsurancePoolSwitchStage`
- `Insurance`: `setFeeRate`, `setDeleveragingCliff`, `setInsurancePoolSwitchStage`

The impact is that setting values to more than 100% might lead to unintended functionality.

Recommend ensuring that the parameters are less than 100%.

raymogg (Tracer) confirmed

[L-03] LIQUIDATION*GAS*COST may not be a constant

Submitted by 0xRajeev, also found by cmichel and gpersoon

The gas cost for liquidation may change if code is updated/optimized, the compiler is changed or profiling is improved. The developers may forget to update this constant in code. The impact is that the margin validity calculation, which uses this value, may be affected if this changes and hence is not as declared in the constant. This may adversely impact validation.

Because It is safer to make this a constructor-set immutable value that will force usage of an updated accurate value at deployment time.

Recommend evaluating if the sensitivity to this value is great enough to justify a setter to change it if incorrectly initialized at deployment.

raymogg (Tracer) confirmed in (duplicate issue)

[L-04] `Deposit` event should use the converted WAD amount

Submitted by 0xRajeev

The `Deposit` event uses the function parameter amount instead of the `convertedWadAmount` which is what is used to update the user's position and TVL because it prevents any dust deposited in amount. This will also make it consistent with the emit event in the `withdraw` function.

The impact is that the `Deposit` event amount reflects the value with dust while the user position does not. This may lead to confusion.

Recommend using `uint256(convertedWadAmount)` instead of amount in `Deposit` event.

raymogg (Tracer) confirmed

[L-05] TVL calculation in `withdraw()` should use `convertedWadAmount` instead of `amount`

Submitted by 0xRajeev

The TVL calculation in `deposit()` uses `convertedWadAmount` but the one in `withdraw()` uses the parameter `amount`. While `amount` is still in WAD format, it may contain dust which is what the conversion to `rawTokenAmount` and then back to `convertedWadAmount` removes.

The impact of this is that use of `amount` in TVL during `withdraw()` will consider dust while the one in `deposit()` will not, which is inconsistent.

Recommend using `convertedWadAmount` instead of `amount` to be consistent with the increment during `withdraw()` TVL calculation.

raymogg (Tracer) confirmed

[L-06] Lack of a contract existence check may lead to undefined behavior

*Submitted by 0xRajeev, also found by a*delamo and pauliax_

Low-level calls `call` / `delegatecall` / `staticcall` return true even if the account called is non-existent (per EVM design).

Solidity documentation warns:

> "The low-level functions `call`, `delegatecall` and `staticcall` return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed."

Market address may not exist as a contract (e.g. incorrect EOA address used in orders), in which case low-level calls still returns true/success. But the trade is assumed to have been successfully executed.

As a result, `executeTrade()` executes batch orders against a non-existing market contract due to a mistake in the trading interface. The transaction executes successfully without any side-effects because the market doesn't exist. Internal accounting is updated incorrectly.

See related High-severity finding in ToB's Audit of Hermez and ToB's Audit of Uniswap V3. Also see Warning in solidity documentation.

https://github.com/code-423n4/2021-06-tracer/blob/74e720ee100fd027c592ea44f272231ad4dfa2ab/src/contracts/Trader.sol#L121-L129

Recommend that `makeOrder.market` should be checked for contract existence before the low-level call, and then verified to be the actual market contract (but it is not verified as noted in the comment). Evaluate if this is a greater concern than undefined behavior.

[L-07] Using `tx.gasprice` to prevent front-running may lead to failed liquidations

Submitted by 0xRajeev

In `verifyAndSubmitLiquidation()`, the `tx.gasprice` is checked against the `fastGasOracle`'s current gas price presumably to prevent liquidators front-running others for the same market/account by using a gas price exceeding the current prevailing price as indicated by the `fastGasOracle`.

The impact is that, if the gas prices are increasing rapidly due to volatility or network congestion, or if the liquidation engines and `fastGasOracle` are out of sync on gas prices because of consulting different sources, then these liquidations will keep failing. Front-running risk on liquidations is not adequately protected by `tx.gasprice` check.

This logic may also be impacted by the upcoming inclusion of EIP-1559 in London fork which affect gas semantics significantly.

Liquidation bots front-running by monitoring mempool or the use of FlashBots for liquidation MEV, is a systemic challenge and not solved by using gasprice logic in contracts. Would recommend evaluating if the benefits match the failure modes.

raymogg (Tracer) acknowledged:

> Good point on the strict check of fast gas price. As you pointed out this is to prevent gas auctions from occuring and causing liquidations to be extremely expensive, however this could become a bottleneck in times of rapidly changing gas prices.

> Have acknowledged that this could cause problems in certain scenarios. The team will be thinking about a safer implementation of this mechanism.

[L-08] Potential division by zero

Submitted by 0xRajeev

In function `minimumMargin()`, `maximumLeverage` being zero is not handled because it will result in div by zero as `PRBMathUD60x18.div` expects non-zero `diTracer`.

The impact is that various critical market functions will revert if `maximumLeverage` is zero. See issue page for effected code.

Recommend adding checks to make sure `maximumLeverage` is never zero or handle appropriately.

raymogg (Tracer) confirmed

[L-09] Unlocked pragma used in multiple contracts

Submitted by shw, also found by JMukesh

Most of the contracts use an unlocked pragma (e.g., `pragma solidity ^0.8.0`) which is not fixed to a specific Solidity version. Locking the pragma helps ensure that contracts do not accidentally get deployed using a different compiler version with which they have been tested the most. Please use `grep -R pragma .` to find the unlocked pragma statements in the codebase

Recommend locking pragmas to a specific Solidity version. Consider the compiler bugs in the following lists and ensure the contracts are not affected by them. It is also recommended to use the latest version of Solidity when deploying contracts (see Solidity docs).

Solidity compiler bugs: Solidity repo - known bugs Solidity repo - bugs by version

raymogg (Tracer) confirmed but disagreed with severity:

> Disagree with severity as the Solidity version is defined in the project config as well so the risk of the contracts being deployed with the wrong version is low. Should be a 0

cemozerr (Judge) commented:

> Marking this as low risk as unlocked pragma can lead to compiler bugs.

[L-10] `LibMath` fails implicitly

Submitted by cmichel

When `LibMath.abs` is called with -2^255 (`type(int256).min`), it tries to multiply it by `-1` but it'll fail as it exceeds the max signed 256-bit integers. The function will fail with an implicit error that might be hard to locate.

Recommend throwing an error similar to `toInt256` like `int256 overflow.`

raymogg (Tracer) confirmed

[L-11] `LibMath.sumN` can iterate over array

Submitted by cmichel

When `LibMath.sumN` function does not check if `n <= arr.length` and can therefore fail if called with `n > arr.length` . The caller must always check that it's called with an argument that is less than `n` which is inconvenient.

Recommend changing the condition to iterate up to `min(n, arr.length)` .

raymogg (Tracer) confirmed

[L-12] todos left in the code

Submitted by gpersoon

There are several todos left in the code. See Issue page for list. Recommend checking, fixing and removing the todos before it is deployed in production

raymogg (Tracer) confirmed

[L-13] check sign in `calculateSlippage`

Submitted by gpersoon

In function `calculateSlippage` of `LibLiquidation.sol`, the value of `amountToReturn` is calculated by subtracting to numbers. Later on it is checked to see if this value is negative. However, `amountToReturn` is an unsigned integer so it can never be negative. If a negative number would be attempted to be assigned, the code will revert, because solidity 0.8 checks for this. See `LibLiquidation.sol` L106.

```
function calculateSlippage(
...
    uint256 amountToReturn = 0;
    uint256 percentSlippage = 0;
    if (avgPrice < receipt.price && receipt.liquidationSide == I
        amountToReturn = amountExpectedFor - amountSoldFor;
    } else if (avgPrice > receipt.price && receipt.liquidationSi
        amountToReturn = amountSoldFor - amountExpectedFor;
    }
    if (amountToReturn <= 0) {    // can never be smaller than 0
        return 0;
    }
```

Recommend double checking if `amountToReturn` could be negative. If this is the case, change the type of `amountToReturn` to int256 and add the appropriate type casts.

raymogg (Tracer) confirmed:

> Confirmed but think this is a 0 on severity. The check while a bit redundant on the less than case, is actually still needed as we do want to catch the case where `amountToReturn = 0`.

cemozerr (Judge) commented:

> Marking this as low risk as a “Double check if amountToReturn could be negative” seems necessary for the scenarios where the amountToReturn could underflow.

[L-14] The `averagePriceForPeriod` function may revert without proper error message returned

Submitted by shw, also found by gpersoon

The `averagePriceForPeriod` function of `LibPrices` does not handle the case where `j` equals 0 (i.e., no trades happened in the last 24 hours). The transaction reverts due to dividing by 0 without a proper error message returned.

Recommend adding `require(j > 0, "...")` before line 73 to handle this special case.

raymogg (Tracer) confirmed

[L-15] make sure `withdrawFees` always can withdraw

Submitted by gpersoon

If you call the function `withdrawFees` when "TVL" is not enough for the fee, then the code would revert. In this case the fees cannot be withdrawn. Although it is unlikely that the TVL would be wrong, it is probably better to be able to withdraw the remaining fees.

`TracerPerpetualSwaps.sol` L508

```
    function withdrawFees() external override {
        uint256 tempFees = fees;
        fees = 0;
        tvl = tvl - tempFees;

        // Withdraw from the account
        IERC20(tracerQuoteToken).transfer(feeReceiver, tempFees);
        emit FeeWithdrawn(feeReceiver, tempFees);
    }
```

Recommend adding something like `tempFees = min (fees, tvl);` , and changing `fees=0` to `fees -= tempFees;`

raymogg (Tracer) confirmed

[L-16] `matchOrders` could/should check market

Submitted by gpersoon

The function `matchOrders` of `TracerPerpetualSwaps.sol` doesn't check that the contract itself is indeed equal to `order1.market` and `order2.market`.

The function `executeTrade` in `Trader.sol`, which calls the `matchOrders`, can deal with multiple markets.

Suppose there would be a mistake in `executeTrade`, or in a future version, the `matchOrders` would be done in the wrong market.

`TracerPerpetualSwaps.sol` L216

```
    function `matchOrders`( Perpetuals.Order memory order1, Perpetua
```

`Trader.sol` L67

```
     function `executeTrade`(Types.SignedLimitOrder[] memory makers,
    ...
        (bool success, ) = makeOrder.market.call(
        abi.encodePacked(
            ITracerPerpetualSwaps(makeOrder.market).matchOrders.sele
            abi.encode(makeOrder, takeOrder, fillAmount)
        )
    );
```

`LibPerpetuals.sol` L128

```
    function canMatch( Order memory a, uint256 aFilled,Order memory
        ...
            bool marketsMatch = a.market == b.market;
```

Recommend adding something like:

```
require ( order1.market == address(this), "Wrong market");
```

Note: `canMatch` already verifies that `order1.market== order2.market`

raymogg (Tracer) confirmed

[L-17] inclusive check that account is not above minimum margin

Submitted by pauliax

Here the check `currentMargin < Balances.minimumMargin` should be inclusive <= to indicate the account is not above minimum margin:

```
        require(
            currentMargin <= 0 ||
                uint256(currentMargin) < Balances.minimumMargin(pos,
            "LIQ: Account above margin"
        );
```

Recommend using `uint256(currentMargin) <= Balances.minimumMargin`

raymogg (Tracer) confirmed

[L-18] hardcoded `chainID`

Submitted by pauliax, also found by s1m0, shw and 0xRajeev

Hardcoding `chainID` is error-prone (in case you decide to deploy on a different chain and forget to change this, or if the chain forks, etc...): `uint256 public constant override chainId = 1337;` // Changes per chain

Recommend better utilization of global variable `block.chainid`, or you can also retrieve `chainID` via assembly.

raymogg (Tracer) disputed in a duplicate issue

> Disputing as an issue as while the suggested approach is a better solution (dynamically setting ChainID), deploying a Trader contract with the wrong ID does not affect the system. A new Trader can be deployed using the appropriate ChainID if one is accidentally deployed with the wrong ChainID.

> Currently this ChainID is just updated before deployment. The team will implement the dynamic approach moving forward.

Note: *Additional conversation regarding this vulnerability can be found* here

[L-19] `Prices.averagePrice` does not show a difference between no trades and a zero price

Submitted by shw

The `getHourlyAvgTracerPrice` and `getHourlyAvgOraclePrice` functions in `Pricing` return 0 if there is no trade during the given `hour` because of the design of `averagePrice`, which could mislead users that the hourly average price is 0. The same problem happens when emitting the old hourly average in the `recordTrade` function. See `Pricing.sol` L254-L256, L262-L264, and L74.

Recommend returning a special value (e.g., `type(uint256).max`) from `averagePrice` if there is no trade during the specified hour to distinguish from an actual zero price. Handle this particular value whenever the `averagePrice` function is called by others.

raymogg (Tracer) confirmed

[L-20] Margin value is not checked to be non-negative in `leveragedNotionalValue`

Submitted by shw

The `leveragedNotionalValue` function of `LibBalance` gets the margin value of a position (i.e., the `marginValue` variable) to calculate the notional value. However,

the position's margin value is not checked to be non-negative. Margin with a value less than zero is considered invalid and should be specially handled. L80 in `LibBalances.sol`.

Recommend checking whether `marginValue` is less than zero and handle this case.

OsmanBran acknowledged:

> Although in a normal state marginValue should not be negative (due to being liquidated prior to this), this function should still handle negative values for marginValue and result in valid calculations. Reverting the function due to negative margin values will cause undesirable side-effects in the system.

[L-21] The `currentHour` variable in `Pricing` could be out of sync

Submitted by shw

The `recordTrade` function in `Pricing` updates the `currentHour` variable by 1 every hour. However, if there is no trade (i.e., the `recordTrade` is not called) during this hour, the `currentHour` is out of sync with the actual hour. As a result, the `averagePriceForPeriod` function uses the prices before 24 hours and causes errors on the average price. See `Pricing.sol` L90-L94.

Recommend calculating how much time passed (e.g., `(block.timestamp - startLastHour) / 3600`) to update the `currentHour` variable correctly.

raymogg (Tracer) confirmed

[L-22] Potential Out-of-Gas exception due to unbounded loop

Submitted by 0xRajeev

Trading function `executeTrade()` batch executes maker/taker orders against a market. The trader/interface provides arrays of makers/takers which is unbounded. As a result, if the number of orders is too many, there is a risk of this transaction

exceeding the block gas limit (which is 15 million currently). See `Trader.sol` L67 and L78

The impact is that if `executeTrade()` is called with too many orders in the batch, the transaction might exceed block gas limit and revert, resulting in none of the orders are executed.

See similar medium-severity finding from ConsenSys's Audit of Growth DeFi.

Recommend limiting the number or orders executed based on `gasleft()` after every iteration, or estimating the gas cost and enforcing an upper bound on the number of orders allowed in maker/taker arrays.

raymogg (Tracer) confirmed

[L-23] Using array memory parameter without checking its length

Submitted by JMukesh

These array memory parameters can be problematic if not used properly. For example, if the array is very large, it may overlap over other part of memory (`Liquidation.sol` L274).

This an example to show the exploit:

```
// based on https://github.com/paradigm-operations/paradigm-ctf-
pragma solidity ^0.4.24; // only works with low solidity versior

contract test{
struct Overlap {
uint field0;
}
event log(uint);

function mint(uint[] memory amounts) public returns (uint) {
// this can be in any solidity version
Overlap memory v;
v.field0 = 1234;
emit log(amounts[0]); // would expect to be 0 however is 1234
```

```
return 1;
}

function go() public { // this part requires the low solidity ve
uint x=0x8000000000000000000000000000000000000000000000000000000
bytes memory payload = abi.encodeWithSelector(this.mint.selector
bool success=address(this).call(payload);
}
}
```

Recommend checking array length before using it.

raymogg (Tracer) commented:

> Duplicate of #79

sporejack (Tracer) commented:

> So the provided PoC code works (under `solc` 0.4.24) subject to test case:

```
const { expect } = require("chai");

describe("Overlap", async () => {
    describe("go", async () => {
        let overlapFactory;
        let overlap;

        before(async () => {
            overlapFactory = await ethers.getContractFactory("Ov
            overlap = await overlapFactory.deploy();

            await overlap.deployed();
        });

        context("When called", async () => {
          it("Emits `log` event with correct value", async () =>
            var firstAmount = ethers.BigNumber.from("1234");

            await expect(overlap.go()).to.emit(overlap, "log").w
          });
        });
    });
```

```
});
```

[sporejack (Tracer) confirmed]():

> My assessment is:

Impact	Difficulty	Overall
Low	Low	Low

> With rationale:

- Unclear (*a priori*) exactly *how* PoC constitutes an exploit
- PoC payload will likely cause unexpected behaviour in production codebase
- Relatively easy for adversary to craft viable payload (simple overflow)

[L-24] Wrong token approval

Submitted by cmichel

The pool holdings of `Insurance` (`publicCollateralAmount` and `bufferCollateralAmount`) is in WAD (18 decimals) but it's used as a raw token value in `drainPool`

```
// amount is a mix of pool holdings, i.e., 18 decimals
// this requires amount to be in RAW! if tracerMarginToken has >
tracerMarginToken.approve(address(tracer), amount);
// this requires amount to be in WAD which is correct
tracer.deposit(amount);
```

If `tracerMarginToken` has less than 18 decimals, the approval approves orders of magnitude more tokens than required for the `deposit` call that follows. If `tracerMarginToken` has more than 18 decimals, the `deposit` that follows would fail as fewer tokens were approved, but the protocol seems to disallow tokens in general with more than 18 decimals.

Recommend converting the `amount` to a "raw token value" and approve this one instead.

raymogg (Tracer) confirmed but disagreed with severity:

> The issue is correct in pointing out that the wrong approve amount is used, however disagree with the severity.

> It is common practice to approve the maximum amount of tokens for a contract to spend already. This bug simply allows more tokens to be approved (to a trusted contract in the system), than was intended. This is only exploitable if paired with another bug in the Tracer contracts. As is, no users would be affected.

cemozerr (Judge) lowered severity from 2 to 1:

> Marking this as low-risk as it would only pose a security threat coupled with another bug.

Non-Critical Findings (20)

- [N-01] Missing checks for `lowestMaxLeverage` < `maxLeverage` and `insurancePoolSwitchStage` < `deleveragingCliff`
- [N-02] Superfluous `verifySignature` function
- [N-03] State variable not used
- [N-04] Change `claimEscrow()` to external
- [N-05] Only one constructor with an emit
- [N-06] Use constants for numbers
- [N-07] alternative solidity coding
- [N-08] Comment for formula `calcEscrowLiquidationAmount` different than code
- [N-09] Comment in `partialLiquidationIsValid` misleading
- [N-10] use try catch
- [N-11] Comment in `claimEscrow`
- [N-12] Use immutable keyword29
- [N-13] Close-ended time ranges may confuse users/interfaces
- [N-14] Unnecessary type conversions

- [N-15] `setDecimals` can be set by anyone and not used
- [N-16] Event log poisoning/griefing in `withdrawFees()`
- [N-17] `claimEscrow` with not yet existing id
- [N-18] orders and `orderToSig` mappings
- [N-19] Dangerous use of storage data location specifier
- [N-20] Missing length check on array could lead to undefined behavior

Gas Optimizations (12)

- [G-01] Gas savings in `getPoolFundingRate()`
- [G-02] Gas savings in `verifyAndSubmitLiquidation()`
- [G-03] Unused State variable
- [G-04] state variable which can be declared as immutable
- [G-05] function which can declared as external
- [G-06] Use EIP-1167 in order to deploy new perpetual swap contracts
- [G-07] Variables that can be converted into immutables
- [G-08] [Gas] Change some function parameters from `memory` to `calldata`
- [G-09] [Gas] Use at least 0.8.0 instead of 0.8.4
- [G-10] `amountToReturn` > `receipt.escrowedAmount` could be inclusive
- [G-11] recalculation of 10**18
- [G-12] `executionPrice`, `newMakeAverage` and `newTakeAverage` before calling the market

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct

formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top