Tracer contest Findings & Analysis Report

2021-11-15

Table of contents

Overview

About C4

Code4rena (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 Tracer's smart contract system written in Solidity. The code contest took place between October 7—October 13 2021.

Wardens

6 Wardens contributed reports to the Tracer contest code contest:

1. cmichel
2. WatchPug
3. loop
4. pauliax
5. yeOlde
6. AC0611

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities and 29 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity, 3 received a risk rating in the category of MEDIUM severity, and 4 received a risk rating in the category of LOW severity.

C4 analysis also identified 11 non-critical recommendations and 11 gas optimizations.

Scope

The code under review can be found within the C4 Tracer contest repository, and is composed of 22 smart contracts written in the Solidity programming language and includes 1,365 lines of Solidity code and 0 lines of JavaScript.

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.

Medium Risk Findings (3)

[M-01] Wrong keeper reward computation

Submitted by cmichel.

The `PoolKeeper.keeperReward` computation mixes WADs and Quads which leads to issues.

1. Note that `keeperTip` returns values where `1 = 1%`, and `100 = 100%`, the same way `BASE_TIP = 5 = 5%`. Thus `_tipPercent =`

`ABDKMathQuad.fromUInt(keeperTip)` is a Quad value of this keeper tip, and not in "wad units" as the comment above it says.

```
// @audit 👇 this comment is not correct, it's in Quad units
// tip percent in wad units
bytes16 _tipPercent = ABDKMathQuad.fromUInt(keeperTip(_savedPrev
```

2. Now the `wadRewardValue` interprets `_tipPercent` as a WAD + Quad value which ultimately leads to significantly fewer keeper rewards:

It tries to compute `_keeperGas + _keeperGas * _tipPercent` and to compute `_keeperGas * _tipPercent` it does a wrong division by `fixedPoint` (1e18 as a quad value) because it thinks the `_tipPercent` is a WAD value (100%=1e18) as a quad, when indeed `100%=100`. It seems like it should divide by `100` as a quad instead.

```
ABDKMathQuad.add(
    ABDKMathQuad.fromUInt(_keeperGas),
    // @audit there's no need to divide by fixedPoint, he wants
    ABDKMathQuad.div((ABDKMathQuad.mul(ABDKMathQuad.fromUInt(_ke
)
```

Impact

The keeper rewards are off as the `_keeperGas * _tipPercent` is divided by 1e18 instead of 1e2. Keeper will just receive their `_keeperGas` cost but the tip part will be close to zero every time.

Recommended Mitigation Steps

Generally, I'd say the contract mixes quad and WAD units where it doesn't have to do it. Usually, you either use WAD or Quad math but not both at the same time. This complicates the code. I'd make `keeperTip()` return a `byte16` Quad value as a percentage where `100% = ABDKMathQuad.fromUInt(1)`. This temporary float result can then be used in a different `ABDKMathQuad` computation.

Alternatively, divide by 100 as a quad instead of 1e18 as a quad because `_tipPercent` is not a WAD value, but simply a percentage where `1 = 1%`.

```
    ABDKMathQuad.add(
        ABDKMathQuad.fromUInt(_keeperGas),
        // @audit there's no need to divide by fixedPoint, he wants
        ABDKMathQuad.div((ABDKMathQuad.mul(ABDKMathQuad.fromUInt(_ke
    )
```

mynameuhh (Tracer) confirmed

Alex the Entreprenerd (judge) commented:

> Agree with the finding and its severity, great find.

[M-02] Deposits don’t work with fee-on transfer tokens

Submitted by cmichel.

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()`. Others are rebasing tokens that increase in value over time like Aave’s aTokens (`balanceOf` changes over time).

Impact

The `PoolCommiter.commit()` function will store the entire `amount` in the commitment but with fee-on-transfer tokens, fewer tokens will be transferred which leads to inconsistencies with the `pool.longBalance()` and in `uncommit`.

Recommended Mitigation Steps

One possible mitigation is to measure the asset change right before and after the asset-transferring routines

rogue developer (Tracer) disputed:

> Only governance (a multisig) can deploy markets, and has complete say over what markets can be deployed (see the `onlyGov` modifier in `PoolFactory.sol#deployPool`). Because new markets being deployed would be done via proposal to the DAO, which include the collateral token being used in a proposed market, markets with fee-on transfer tokens like Aave's aTokens just won't be deployed. I think this is a fairly safe assumption to make and thus we're making it out of scope. In any case, the chances of this happening and slipping past everyone who votes in the proposals *and* not being noticed extremely soon after a market is deployed are extremely low.

Alex the Entreprenerd (judge) commented:

> I think this is a valid finding, the warden has shown a way to tamper with the protocol, extracting value (as such medium severity)

> In terms of mitigation, not using `feeOnTransfer` or `rebasing` tokens is completely legitimate.

[M-03] `uncommit` sends tokens to the wrong user

Submitted by cmichel, also found by WatchPug.

The `PoolCommitter._uncommit` function calls the `ILeveragedPool(leveragedPool).quoteTokenTransfer/mintTokens` function with `msg.sender`. But in `_uncommit`'s case that's the **pool, not the commit owner,** see `onlyPool` modifier on `executeAllCommitments` which calls `_uncommit`.

Impact

Users lose all tokens from their commitments as they are sent / minted to the pool instead.

Recommended Mitigation Steps

Instead of `msg.sender`, use `_commit.owner`:

```
    // minting: return quote tokens to the commit owner
    // @audit msg.sender is pool, should be _commit.owner
    ILeveragedPool(leveragedPool).quoteTokenTransfer(msg.sender, _cc
    // same with mint cases
```

[rogue developer (Tracer) disagreed with severity](#):

> This is a valid issue. However, it requires there to be an underlying bug in the contracts which would make the `executeCommitment` call in `executeAllCommitments` revert (as `_uncommit` is only called in that case). If the warden can find a way for `executeCommitment` to revert, we would consider this to be an issue of this severity but otherwise we disagree with the severity as it requires/needs to be paired with another bug in the contracts.

> It's also worth noting that governance can rescue any funds (rather, quote/collateral tokens) from the `LeveragedPool` contract. So if there were to be a case where there was a bug in the contracts that led to `executeCommitment` reverting and the users having their mints (quote tokens sent back to the `LeveragedPool`), governance could pause the contracts, drain out the equivalent worth and set up another contract where users can burn their tokens/claim them; if they burned the tokens which later uncommitted, then governance could send them an equal amount in collateral tokens. On the same note, if a critical vulnerability in `executeCommitment` were to be happening whereby commits were being uncommitted, then governance could also pause the contracts, rescue the funds and do some combination of the efforts above to ensure users get the funds back securely.

[Alex the Entreprenerd (judge) commented](#):

> Would setting up a commit such that this line will underflow https://github.com/tracer-protocol/perpetual-pools-contracts/blob/646360b0549962352fe0c3f5b214ff8b5f73ba51/contracts/implementation/PoolCommitter.sol#L305 , causing a revert, be a way to cause the function to call `_uncommit` ?

[Alex the Entreprenerd (judge) commented](#):

> Also, to clarify, you're saying you believe the code will never call `_uncommit` as it won't ever revert, right?

[Alex the Entreprenerd (judge) commented](#):

> At this time I believe that if a user mistakenly commits more than the value in `shadowPools[_commitType]` they can cause a silent revert which will trigger the bug

> I think only their own funds are at risk, and either passing along the original committer or storing it in the commitData would allow to safely return them their funds.

> With the information I have this issue sits between medium and high severity, high severity because user funds are at risk, medium because:
>
> ```
> 2 — Med: Assets not at direct risk, but the function of the prot
> ```

> As of now I'll mark as medium and valid.

> Will think it over the weekend

> If the sponsor can let me know their take and reply to the questions above, that can help clarify the severity and validity

[rogue developer (Tracer) commented](#):

> Would setting up a commit such that this line will underflow https://github.com/tracer-protocol/perpetual-pools-contracts/blob/646360b0549962352fe0c3f5b214ff8b5f73ba51/contracts/implementation/PoolCommitter.sol#L305 , causing a revert, be a way to cause the function to call _uncommit ?

> @Alex the Entreprenerd No. `shadowPools` is a mapping of commit types to the sum of pool tokens to be burned (rather, to be executed because they've already been burned), or sum of collateral tokens to be used in minting that haven't been used for minting yet. `executeCommitment` can only be called on `Commit` 's, which are in the `commits` mapping, which can only be added to via the `commit` function where users have to commit to putting up collateral/burning their pool tokens, which is the function that increments `shadowPools` . I realise that sounds

> a bit convoluted, but basically `executeCommitment` and its `_commit` parameter has a direct dependency on users committing via the `commit` function, which increments `shadowPools` by the value of their commit (which they can't game — their collateral tokens get sent to the `LeveragedPool` contract instantly and their tokens get burned instantly and they don't have access to those funds anymore).

> Also, to clarify, you're saying you believe the code will never call `_uncommit` as it won't ever revert, right?

> Yes, that's right. It is there as a fail-safe (so that if there was some bug in a commit that stopped a commit in the queue from being executed, it wouldn't stop the markets). We are refactoring this code nonetheless though.

[Alex the Entreprenerd (judge) commented]():

> Alright from this information I understand that the underflow idea can't happen (gas optimization there would be to use `unsafe` operations I guess)

> I think given the system a refactoring to send the funds back is warranted

> That said the fact that there seems to be no way to get a revert excludes the high severity.

> That leaves us with the finding either being med or low risk

> Low risk would be acceptable as the code doesn't work as it suggests (`_uncommit` is never executed, and if it did it wouldn't reimburse the user)

> The alternative take is Medium: if `_uncommit` where executed it would cause in a loss of funds / funds stuck

> As of now I'll leave it as med, while we don't have a way to trigger `_uncommit` we can still make the claim that if `_uncommit` where to run, it wouldn't reimburse the user

[Alex the Entreprenerd (judge) commented]():

> Going a little deeper for the sake of clarity:

> The math library is programmed to never revert: https://github.com/tracer-protocol/perpetual-pools-contracts/blob/646360b0549962352fe0c3f5b214ff8b5f73ba51/contracts/implementation/PoolSwapLibrary.sol#L256

> The amounts in commit are always greater than 0 so no revert there

> The pool.setter is innocuous

> The only thing I found is the `pool.quoteTokenTransfer(_commit.owner, amountOut);`

> If for an unfortunate reason the pool is drained from the quoteToken and the safeTransfer fail, then the function would revert

> On that note the way to perform this would be to use `function setKeeper(address _keeper) external override onlyGov onlyUnpaused {`
> To change the keeper to a EOA / Malicious account

> And then run `function payKeeperFromBalances(address to, uint256 amount)` with the full amount (or close to it) of `amount <= shortBalance + longBalance`

> This seems to be a permissioned way (admin privilege) to rug funds from the LeveragedPool as well as enabling the `_uncommit` to be triggered

> Given these findings (which I may misunderstand, so feel free to correct me) I highly recommend the sponsor to ensure there's a timelock for changing keeper
> Additionally (and I may be missing something) allowing `payKeeperFromBalances` to take an indiscriminate amount of funds may prove to be a rug vector the sponsor should consider eliminating.

> I'm fairly confident the math library will never revert, even if you input a high fee, which means that while the end state may be unexpected, the function can be used to rug.

> Highly recommend the sponsor to consider having caps on the `amount` parameter for [`payKeeperFromBalances`]() as this function seems to be the way to break the protocol (and the trust of the users)

[rogue developer (Tracer) commented]():

> @Alex the Entreprenerd Yep, that's completely right. However (and this is something we should have made clear in an assumptions section), all markets will be deployed by the DAO/DAO multisig. You can see that the `payKeeperFromBalances` function has a modifier called `onlyGov` — meaning only governance (the DAO/DAO multisig) can change this and that's something that's immutable and can't be changed. Governance also has the ability to take quote tokens out of the `LeveragedPool` contract (with `withdrawQuote`) unlike the hack with changing the keeper (more directly this way) but we've been going under the assumption that the only reason we'd do this is in the rescue of user funds in the case of some hack.

> If the governance multisig is compromised, they have the ability to do a lot of damage but we've been going under the assumption this just won't be the case, which I think is a safe one to make. So you're right in that `uncommit` can send tokens to the wrong user in case the function reverts because of governance draining funds directly or via setting a malicious keeper, but I think that `uncommit` sending tokens to the wrong user is the least of problems if that happens because governance would only ever drain funds in the case of a major hack. If we're going under the assumption that the multisig *can* easily be compromised, then the "centralisation" point around the DAO would be a much bigger point.

> I still think this should be a low/information by virtue of the fact that unless there is an underlying bug in the contracts, `uncommit` can't be called (it may as well be dead code) — if it gets called because governance has rugged, that is then relatively a fairly small problem because all the funds would be at risk in a much more direct way in that case.

[Alex the Entreprenerd (judge) commented]():

> @rogue developer agree that if governance is malicious, the `_uncommit` path is the least problem

> One thing to note is that the `keeper` has the ability of trying to claim a lot of fees, and the modifier seems to be `onlyKeeper` if the keeper were to be a bot, or a human operator they would have the ability of rugging, unless the parameter `amount` was under some check (let's say less than 1% AUM or something)

> I agree that you can set to the `PoolKeeper` contract which seems safe, once potential rug vector, again from governance would be to inject a high `gasPrice` via `setGasPrice` which as a `onlyOwner` modifier

> Contract for PoolKeeper: https://arbiscan.io/address/0x759E817F0C40B11C775d1071d466B5ff5c6ce28e#code

> The owner is the Dev Multisig: https://arbiscan.io/address/0x0f79e82ae88e1318b8cfc8b4a205fe2f982b928a#readContract

> This does give the Dev Multisig admin privileges and a potential for griefing at the very least if not rugging, as they could raise the gas price, and then run `performUpkeepSinglePool` which would eventually call `payKeeperFromBalances`

> That said, this is something I'm flagging up right now and outside of the contest

[Alex the Entreprenerd (judge) commented]:

> As per the finding at this point I believe it's valid and at medium severity, it is not high severity due to need for existing preconditions that are not "usual", see definition from Gitbook: `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

> We can agree that the pre-condition here are drastic and this finding would be the last of the protocols problems

> In terms of mitigation, ensuring that the funds are send back to the address that initiated the commit are more than sufficient

Alex the Entreprenerd (judge) commented:

> Per the “multisig privileges” above, I recommend you replace `owner` on the poolKeeper with a TimeLock as in it’s current state, the gasPrice may be a way to rug funds from depositors

Low Risk Findings (4)

- [L-01] Contradiction in comment/require statement *Submitted by loop.*
- [L-02] No ERC20 `safeApprove` versions called *Submitted by cmichel.*
- [L-03] Revert in `poolUpkeep` *Submitted by cmichel.*
- [L-04] `PoolKeeper.sol#performUpkeepSinglePool()` Wrong implementation allows attacker to interfere the upkeep of pools *Submitted by WatchPug.*

Non-Critical Findings (11)

- [N-01] Unused imports *Submitted by pauliax.*
- [N-02] No constant for maximum tip (PoolKeeper.sol) *Submitted by yeOlde.*
- [N-03] Unused modifer “onlyFeeReceiver” in LeveragedPool.sol *Submitted by yeOlde.*
- [N-04] Unused local variable in _latestRoundData (ChainlinkOracleWrapper.sol) *Submitted by yeOlde.*
- [N-05] Style issues *Submitted by pauliax.*
- [N-06] Inclusive check of frontRunning > updateInterval *Submitted by pauliax.*
- [N-07] PoolKeeper _gasPrice description says in ETH, but is calculated in wei *Submitted by loop.*
- [N-08] BLOCK_TIME of Arbitrum is less than 13 seconds *Submitted by pauliax.*
- [N-09] Validate max fee *Submitted by cmichel.*
- [N-10] Unsafe `int256` casts in `executePriceChange` *Submitted by cmichel.*
- [N-11] Missing parameter validation *Submitted by cmichel, also found by loop.*

Gas Optimizations (12)

- [G-01] Unused Named Returns Can Be Removed *Submitted by yeOlde.*
- [G-02] Minimize Storage Slots (LeveragedPool.sol) *Submitted by yeOlde.*
- [G-03] token out of range check can be simplified *Submitted by pauliax.*
- [G-04] Useless multiplication by 1 *Submitted by pauliax.*
- [G-05] Adding unchecked directive can save gas *Submitted by WatchPug, also found by pauliax.*
- [G-06] Immutable state variables *Submitted by pauliax.*
- [G-07] Unused state variables *Submitted by pauliax.*
- [G-08] LeveragedPool has require statements which are also checked in library *Submitted by loop.*
- [G-09] Gas: Inefficient modulo computation *Submitted by cmichel.*
- [G-10] Gas: `transferGovernance` can save an sload *Submitted by cmichel.*
- [G-11] Gas: shadow pools are only required for burn types *Submitted by cmichel.*

Disclosures