YIELD

Yield micro contest #1 Findings & Analysis Report

2021-09-17

Table of contents

- Overview
    - About C4
    - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (5)
    - [H-01] `CompositeMultiOracle` returns wrong decimals for prices?
    - [H-02] `ERC20Rewards` returns wrong rewards if no tokens initially exist
    - [H-03] `ERC20Rewards` breaks when setting a different token
    - [H-04] Rewards accumulated can stay constant and often not increment
    - [H-05] Exchange rates from Compound are assumed with 18 decimals
- Medium Risk Findings (4)
    - [M-01] No ERC20 safe* versions called
    - [M-02] `TimeLock` cannot schedule the same calls multiple times

    - [M-03] Rewards squatting - setting rewards in different ERC20 tokens opens various economic attacks.
    - [M-04] Use `safeTransfer` instead of `transfer`
- Low Risk Findings (12)
    - [L-01] `updateTime` of get is 0
    - [L-02] Different definition of `beforeMaturity()` and `afterMaturity()` modifier in different file
    - [L-03] Missing input validation to check that `end` > `start`
    - [L-04] Upgrading solc compiler version may help with bug fixes
    - [L-05] Missing emits for events
    - [L-06] Unused `cauldron_` parameter
    - [L-07] Missing check for contract existence
    - [L-08] `_peek` does not work for tokens with > 18 decimals
    - [L-09] `ERC20Rewards` claiming can fail if no reward tokens
    - [L-10] improve safety of role constants
    - [L-11] `EmergencyBrake.sol` : Permissions cannot be re-planned after termination
    - [L-12] `ERC20Rewards.sol` : Have a method to calculate the latest `rewardsPerToken` accumulated value
- Non-Critical findings (9)
- Gas Optimizations (18)
- Disclosures

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 Yield smart contract system written in Solidity. The code contest took place between August 11—August 14 2021.

Wardens

8 Wardens contributed reports to the Yield micro code contest:

1. cmichel
2. shw
3. moose-code
4. 0xRajeev
5. hickuphh3
6. gpersoon
7. Jmukesh
8. PierrickGT

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities and 48 total findings. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 12 received a risk rating in the category of LOW severity.

C4 analysis also identified 9 non-critical recommendations and 18 gas optimizations.

Scope

The code under review can be found within the C4 Yield micro code contest repository is comprised of 61 smart contracts written in the Solidity programming language and includes 4,115 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 (5)

[H-01] `CompositeMultiOracle` returns wrong decimals for prices?

Submitted by cmichel

The `CompositeMultiOracle.peek/get` functions seem to return wrong prices. It’s unclear what decimals `source.decimals` refers to in this case. Does it refer to `source.source` token decimals?

It chains the price arguments through `_peek` function calls and a single price is computed as:

```
(priceOut, updateTimeOut) = IOracle(source.source).peek(base, qu
// @audit shouldn't this divide by 10 ** IOracle(source.source).
priceOut = priceIn * priceOut / (10 ** source.decimals);
```

Assume all oracles use 18 decimals (`oracle.decimals()` returns 18) and `source.decimals` refers to the *token decimals* of `source.source` .

Then going from `USDC -> DAI -> USDT` (`path = [DAI]`) starts with a price of `1e18` in `peek` :

- `_peek(USDC, DAI, 1e18)` : Gets the price of `1e6 USDC` (as USDC has 6 decimals) in DAI with 18 decimals precision (because all oracle precision is set to 18): `priceOut = priceIn * 1e18 / 1e6 = 1e18 * 1e18 / 1e6 = 1e30`
- `_peek(DAI, USDT, 1e30)` : Gets the price of `1e18 DAI` (DAI has 18 decimals) with 18 decimals precision: `priceOut = priceIn * 1e18 / 1e18 = priceIn = 1e30`

It then uses `1e30` as the price to go from `USDC` to `USDT` : `value = price * amount / 1e18 = 1e30 * (1.0 USDC) / 1e18 = 1e30 * 1e6 / 1e18 = 1e18 = 1e12 * 1e6 = 1_000_000_000_000.0 USDT` . Inflating the actual `USDT` amount.

The issue is that `peek` assumes that the final price is in 18 decimals in the `value = price * amount / 1e18` division by `1e18` . But `_peek` (and `_get`) don’t enforce this.

Recommend that `_peek` should scale the prices to `1e18` by doing:

```
(priceOut, updateTimeOut) = IOracle(source.source).get(base, quo
// priceOut will have same decimals as priceIn if we divide by o
priceOut = priceIn * priceOut / (10 ** IOracle(source.source).de
```

It does not need to divide by the `source.source` *token precision* (`source.decimals`), but by the oracle precision (`IOracle(source.source).decimals()`).

alcueca (Yield) acknowledged:

> It’s confusing to deal with all these decimals, I should at least comment the code better, and try to make it easier to understand.

> It’s unclear what decimals source.decimals refers to in this case. Does it refer to source.source token decimals?

> CompositeMultiOracle takes IOracle contracts as sources, so `source.decimals` refers to the token decimals of the oracle, not of the data source one level below.

> It does not need to divide by the source.source token precision (source.decimals), but by the oracle precision (IOracle(source.source).decimals()).

> The source.source token precision would be `IChainlinkAggregatorV3(source.source()).decimals()`, the source oracle precision is `source.decimals()`. CompositeMultiOracle cannot make an assumption on any fields present on `source.source`, and must work only with the underlying `source` IOracles.

> I’m still not disputing this finding. I need to dig further to make sure the decimals are right when different IOracle sources have different decimals, and I’ve hardcoded a few `1e18` in there. Those are code smells.

alcueca (Yield) patched:

> Sent me into a wild goose chase to support IOracle of multiple decimals as sources to CompositeMultiOracle, only to realize that we create all IOracles and we always create them with 18 decimals, converting from the underlying data source if needed.

> Ended up making CompositeMultiOracle require that underlying oracles have 18 decimals. Done.

alcueca (Yield) further patched:

> Further refactored all oracles so that decimals are handled properly, and work on taking an amount of base as input, and returning an amount of quote as output. Our oracles don't have decimals themselves anymore as a state variable, since the return values are in the decimals of quote. This means that CompositeMultiOracle is agnostic with regards to decimals, and doesn't even need to know about them.

[H-02] `ERC20Rewards` returns wrong rewards if no tokens initially exist

Submitted by cmichel

The `ERC20Rewards._updateRewardsPerToken` function exits without updating `rewardsPerToken_.lastUpdated` if `totalSupply` is zero, i.e., if there are no tokens initially.

This leads to an error if there is an active rewards period but no tokens have been minted yet.

Example: `rewardsPeriod.start: 1 month ago`, `rewardsPeriod.end: in 1 month`, `totalSupply == 0`.

The first mint leads to the user (mintee) receiving all rewards for the past period (50% of the total rewards in this case).

- `_mint` is called, calls `_updateRewardsPerToken` which short-circuits. `rewardsPerToken.lastUpdated` is still set to `rewardsPeriod.start` from the constructor. Then `_updateUserRewards` is called and does not currently yield any rewards. (because both balance and the index diff are zero). User has now minted the tokens, `totalSupply` increases and user balance is set.
- User performs a `claim`: `_updateRewardsPerToken` is called and `timeSinceLastUpdated = end - rewardsPerToken_.lastUpdated = block.timestamp - rewardsPeriod.start = 1 month`. Contract "issues" rewards for the past month. The first mintee receives all of it.

The first mintee receives all pending rewards when they should not receive any past rewards. This can easily happen if the token is new, the reward period has already been initialized and is running, but the protocol has not officially launched yet. Note that `setRewards` also allows setting a date in the past which would also be fatal in this case.

Recommend that the `rewardsPerToken_.lastUpdated` field must always be updated in `_updateRewardsPerToken` to the current time (or `end`) even if `_totalSupply == 0`. Don't return early.

alcueca (Yield) confirmed:

> You are right, that's a great finding. For the record, I think that this is what **this line in Unipool.sol** does:

```
function rewardPerToken() public view returns (uint256) {
  if (totalSupply() == 0) {
    return rewardPerTokenStored;
  }
```

> I'll apply the mitigation step suggested, with a conditional to not do the `rewardsPerToken_.accumulated` math that would revert.

> Now I know the feeling of the devs that fork a known project and leave a pesky conditional out, thanks again :D

alcueca (Yield) patched:

> Fix

[H-03] `ERC20Rewards` breaks when setting a different token

Submitted by cmichel

The `setRewards` function allows setting a different token. Holders of a previous reward period cannot all be paid out and will receive **their old reward amount** in the new token.

This leads to issues when the new token is more (less) valuable, or uses different decimals.

Example: Assume the first reward period paid out in `DAI` which has 18 decimals. Someone would have received `1.0 DAI = 1e18 DAI` if they called `claim` now. Instead, they wait until the new period starts with `USDC` (using only 6 decimals) and can `claim` their `1e18` reward amount in USDC which would equal `1e12 USDC`, one trillion USD.

Changing the reward token only works if old and new tokens use the same decimals and have the exact same value. Otherwise, users that claim too late/early will lose out.

Recommend disallowing changing the reward token, or clearing user's pending rewards of the old token. The second approach requires more code changes and keeping track of what token a user last claimed.

alcueca (Yield) confirmed:

> Maybe I should have used stronger language: `// If changed in a new rewards program, any unclaimed rewards from the last one will be served in the new token`

> The issue is known, but you are right in pointing it out. There are few situations in which changing the rewards token would make sense (such as replacing a faulty rewards token by a fixed one). I think it would be best to just disallow changing the token.

alcueca (Yield) patched:

> Fix

[H-04] Rewards accumulated can stay constant and often not increment

Submitted by moose-code

`rewardsPerToken_.accumulated` can stay constant while `rewardsPerToken_.lastUpdated` is continually updated, leading to no actual rewards being distributed. I.e. No rewards accumulate.

Line 115, `rewardsPerToken_.accumulated` could stay constant if there are very quick update intervals, a relatively low `rewardsPerToken_.rate` and a decent supply of the ERC20 token.

I.e. imagine the token supply is 1 billion tokens (quite a common amount, note even if a supply of only say 1 million tokens this is still relevant). i.e. 1e27 wei.

Line 115 has

```
    1e18 * timeSinceLastUpdated * rewardsPerToken_.rate / _totalSupp
```

`timeSinceLastUpdated` can be crafted to be arbitrarily small by simply transferring or burning tokens, so lets exclude this term (it could be 10 seconds etc). Imagine total supply is 1e27 as mentioned.

Therefore, `1e18 * rewardsPerToken_.rate / 1e27`, which shows that if the `rewardsPerToken_.rate` is < 1e9, something which is very likely, then the accumulated amount won’t increment, as there are no decimals in solidity and this line of code will evaluate to adding zero. While this is rounded down to zero, critically, `rewardsPerToken_.lastUpdated = end;` is updated.

The reason I have labelled this as a high risk is the express purpose of this contract is to reward users with tokens, yet a user could potentially quite easily exploit this line to ensure no one ever gets rewards and the accumulated amount never increases.

Given a fairly large token supply, and a relatively low emissions rate is set, that satisfies the above equation, for the entire duration of the rewards period, the user simply sends tokens back and forth every couple seconds (gas limitations, but layer 2), to keep the delta `timeSinceLastUpdated` close to 1.

This way the accumulated amount will never tick up, but time keeps being counted.

Furthermore, I would say this is high risk as this wouldn't even need an attacker. Given the transfer function is likely often being called by users, `timeSinceLastUpdated` will naturally be very low anyways.

Even if not so extreme as the above case, Alberto points out that "rounding can eat into the rewards" which is likely to be prevalent in the current scenario and make a big impact over time on the targeted vs actual distribution.

Again, this problem is more likely to occur in naturally liquid tokens where lots of transfer, mint or burn events occur.

As suggested by Alberto, the simplest it to probably not update the `rewardsPerToken_.lastUpdated` field if `rewardsPerToken_.accumulated` does not change. Although this change should be closely scrutinized to see it doesn't introduce bugs elsewhere.

alcueca (Yield) acknowledged and disagreed with severity:

> While the issue exists, it's not as severe as portrayed, and doesn't need fixing.

> There is an error in the assessment, and it is that the `rate` refers to the rewards amount distributed per second among all token holders. It is not the rewards amount distributed per token per second (that's dynamically calculated).

> Also, it needs to be taken into account that `rewardsPerToken.accumulated` is stored scaled up by 1e18, to avoid losing much ground to rounding.

```
struct RewardsPerToken {
  uint128 accumulated;                             // Accumulate
  uint32 lastUpdated;                              // Last time t
  uint96 rate;                                     // Wei rewarde
}
```

> One of the largest cap tokens is Dai, with a distribution close to 1e28. If ERC20Rewards were to distribute 1 cent/second among all token holders (which wouldn't be very exciting), and block times were of 1 second, the accumulator would still accumulate.

> `accumulator += 1e18 (scaling) * 1 (seconds per block) * 1e16 (Dai wei / second) / 1e28 (Dai total supply)` The increase to the `accumulator` is of 1e6, which gives plenty of precision. I would expect a rewards program on Dai holders would be at least 1e6 larger per second.

> On the other hand, `accumulator` is an `uint128`, which holds amounts of up to 1e38. To overflow it we would need a low cap token (let's say USDC, with 1e15), and a high distribution (1e12 per second, which is unreal), and we run the program for 3 years, or 1e9, to make it easy.

> The accumulator at the end of the ten years would be: `accumulator = 1e18 (scaling) * 1e9 (seconds) * 1e12 (distribution) / 1e15 (supply) = 1e24` Which doesn't overflow.

ghoul-sol (judge) commented:

> I'll keep high risk as there should be no scenario where the math breaks.

[H-05] Exchange rates from Compound are assumed with 18 decimals

Submitted by shw

The `CTokenMultiOracle` contract assumes the exchange rates (borrowing rate) of Compound always have 18 decimals, while, however, which is not true. According to the **Compound documentation**, the exchange rate returned from the `exchangeRateCurrent` function is scaled by `1 * 10^(18 - 8 + Underlying Token Decimals)` (and so does `exchangeRateStored`). Using a wrong decimal number on the exchange rate could cause incorrect pricing on tokens. See `CTokenMultiOracle.sol` **#L110**.

Recommend following the documentation and getting the decimals of the underlying tokens to set the correct decimal of a `Source`.

alcueca (Yield) confirmed:

> Thanks a lot for coming up with this. I had looked into how Compound defined the decimals and couldn't find it.

alcueca (Yield) patched:

> Fix

Medium Risk Findings (4)

[M-01] No ERC20 safe* versions called

Submitted by cmichel, also found by JMukesh and hickuphh3

The `claim` function performs an ERC20 transfer `rewardsToken.transfer(to, claiming);` but does not check the return value, nor does it work with all legacy tokens.

Some tokens (like USDT) don't correctly implement the EIP20 standard and their `transfer`/`transferFrom` function return `void` instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert.

The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do **not** revert if the transfer failed but return `false` instead.

Tokens that don't actually perform the transfer and return `false` are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value.

Recommend using OpenZeppelin's `SafeERC20` versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

alcueca (Yield) confirmed:

> True, thanks for spotting it!

alcueca (Yield) patched:

> Fix

[M-02] `TimeLock` cannot schedule the same calls multiple times

Submitted by cmichel

The `TimeLock.schedule` function reverts if the same `targets` and `data` fields are used as the `txHash` will be the same. This means one cannot schedule the same transactions multiple times.

Imagine the delay is set to 30 days, but a contractor needs to be paid every 2 weeks. One needs to wait 30 days before scheduling the second payment to them.

Recommend also including `eta` in the hash. (Compound's `Timelock` does it as well.) This way the same transaction data can be used by specifying a different `eta`.

alcueca (Yield) confirmed:

> Funny, BoringCrypto was quite negative about including the eta in the txHash. At the time I couldn't think of a reason to repeat the same call with the same data, but you are right that sometimes it might make sense, and storing off-chain the expected eta of each timelocked transaction is something you should do anyway.

> I'll confirm this issue, and will bring it for public discussion once the contest is over.

alcueca (Yield) patched:

> I ended up refactoring the Timelock so that the eta is not included in the parameters, but repeated proposals are allowed.

[M-03] Rewards squatting - setting rewards in different ERC20 tokens opens various economic attacks.

Submitted by moose-code, also found by hickuphh3

Users essentially have an option to either claim currently earned reward amounts on future rewards tokens, or the current rewards token.

Although stated on line 84, it does not take into account the implications the lock in this contract will have on the future value of new tokens able to be issued via rewards.

Smart users will monitor the mempool for `setRewards` transactions. If the new reward token (token b) is less valuable than the old reward token (token a), they can front run this transaction by calling claim. Otherwise, they let their accrued 'token a' roll into rewards of of the more valuable 'token b'.

Given loads of users will likely hold these tokens from day 1, there will potentially be thousands of different addresses squatting on rewards.

Economically, and given the above, it makes sense that the value of new reward tokens, i.e. 'token b' should always be less than that of 'token a'. This is undesirable in a rewards token contract, as there is no reliable way to start issuing a more valuable token at a later stage, unless exposing yourself to a major risk of reward squatting.

i.e. You could not issue a more valuable token in future (for example, if we wanted to run a rewards period issuing an asset like WETH rewards for 10 days) after first initially issuing DAI as a reward. This hamstrings flexibility of the contract.

P.s. This is one of the slickest contracts I've read. Love how awesome it is.Just believe this should be fixed, then its good to go.

It is true you could probably write a script to manually go call `claim` on thousands of squatting token addresses but this is a poor solution.

Recommend instead, that a simple mapping pattern could be used with an index mapping to a reward cycle with a reward token and a new accumulative etc. Users would likely need to be given a period a to claim from old reward cycles before their token balance could no longer reliably used to calculate past rewards. The would still be able to claim everything up until their last action (even though this may be before the rewards cycle ended).

alcueca (Yield) confirmed:

> Thanks! I agree that allowing to change the rewards token is just too troublesome.

alcueca (Yield) patched:

> Fix

[M-04] Use `safeTransfer` instead of `transfer`

Submitted by shw

Tokens not compliant with the ERC20 specification could return `false` from the `transfer` function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

> Callers MUST handle `false` from `returns (bool success)` . Callers MUST NOT assume that `false` is never returned!

See ERC20Rewards.sol L175.

Recommend using the `SafeERC20` library implementation from OpenZeppelin and calling `safeTransfer` or `safeTransferFrom` when transferring ERC20 tokens.

alcueca (Yield) confirmed and patched:

> Fix

Low Risk Findings (12)

[L-01] `updateTime` of get is 0

Submitted by gpersoon, also found by shw

In function `_get` of `CompositeMultiOracle` the `updateTime` is not initialized, so it will be 0.

Function `_get` has the following statement:

```
    updateTimeOut = (updateTimeOut < updateTimeIn) ? updateTimeOu

updateTimeIn ==0 ==>  (updateTimeOut < updateTimeIn)== false ==>
```

So this means the function get will always return `updateTime==0`

The `updateTime` result of the function `get` doesn’t seem to be used in the code so the risk is low. If would only be relevant for future code updates.

```

function get(bytes32 base, bytes32 quote, uint256 amount)  e
...
for (uint256 p = 0; p < path.length; p++) {
  (price, updateTime) = _get(base_, path[p], price, updateTi

function _get(bytes6 base, bytes6 quote, uint256 priceIn, ui
...
  (priceOut, updateTimeOut) = IOracle(source.source).get(bas
  ...
  updateTimeOut = (updateTimeOut < updateTimeIn) ? updateTim
}
```

Recommend adding the following in the beginning of the `_get` function: `updateTime = block.timestamp;`

alcueca (Yield) confirmed and patched:

> Fix

[L-02] Different definition of `beforeMaturity()` and `afterMaturity()` modifier in different file

Submitted by JMukesh

Different definition of `beforeMaturity()` and `afterMaturity()` modifier in `Strategy.sol` L82 and `FYToken.sol` L65 which used `FYTokenFacory()`. See issue page for further elaboration.

See issue page for more.

alcueca (Yield) disagreed with severity:

> It is right that there is a different definition of before and after maturity, and that Strategy.sol should match FYToken.sol, same as Pool.sol does.

> However, there is no impact from this issue. The only thing that could happen is that an user `mint` strategy tokens on an already matured pool, which is harmless.

alcueca (Yield) patched:

> Fix

ghoul-sol (judge) commented:

> per sponsor comment, making this low risk

[L-03] Missing input validation to check that `end > start`

Submitted by OxRajeev

`setRewards()` is missing input validation on parameters `start` and `end` to check if `end > start`. If accidentally set incorrectly, this will allow resetting new rewards while there is an ongoing one (`ERC20Rewards.sol#L74` L88).

Recommend adding a `require()` to check that `end > start`.

alcueca (Yield) confirmed:

> If accidentally set incorrectly, this will allow resetting new rewards while there is an ongoing one.

> I would say that if we set it incorrectly, we would like to reset it as soon as possible :)

> Still, a good check to add, since otherwise it leads to strange behaviour.

alcueca (Yield) patched:

> Fix

[L-04] Upgrading solc compiler version may help with bug fixes

Submitted by 0xRajeev

solc version 0.8.3 and 0.8.4 fixed important bugs in the compiler. Using version 0.8.1 misses these fixes and may cause a vulnerability.

See `ERC20Rewards.sol` L2. Solidity 0.8.4 fixes a bug in the ABI decoder. The release contains an important bugfix. See decoding from memory bug blog post for more details.

Solidity 0.8.3 is a bugfix release that fixes an important bug about how the optimizer handles the Keccak256 opcode. For details on the bug, please see the bug blog post.

Recommend considering upgrading to 0.8.3 or 0.8.4.

alcueca (Yield) confirmed and patched:

> Fix, fix, and fix.

alcueca (Yield) commented:

> I might actually revert the fixes, unless we are affected by the bug fixes. Using solc 0.8.6 forces us to drop the optimizer from 20000 to 5000.

[L-05] Missing emits for events

Submitted by 0xRajeev, also found by cmichel

Few events are missing emits which prevents the intended data from being observed easily by off-chain interfaces (`Strategy.sol#L48` L49).

Recommend adding emits or remove event declarations.

alcueca (Yield) confirmed and patched:

> Fix

[L-06] Unused `cauldron_` parameter

Submitted by 0xRajeev

That `cauldron_` parameter is not used here and `ladle_.cauldron()` is used instead. The `Ladle` constructor initializes its cauldron value and so the only way this could differ from the parameter is if the argument to this function is specified incorrectly. See issue page for referenced code.

Recommend either using parameter, or remove it in favor of the value from `ladle_.cauldron()`.

alcueca (Yield) confirmed and patched:

> Fix

[L-07] Missing check for contract existence

Low-level call returns success even if the contract is non-existent. This requires a contract existence check before making the low-level call (`TimeLock.sol` L93).

See: “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.” from https://docs.soliditylang.org/en/v0.8.7/control-structures.html#error-handling-assert-require-revert-and-exceptions

Recommend checking for target contract existence before call.

alcueca (Yield) confirmed:

> Mmm, we do this in several other places in the codebase, as well as others do in earlier implementations. Will fix.

alcueca (Yield) patched:

> Fix

[L-08] `_peek` does not work for tokens with > 18 decimals

Submitted by cmichel

The `CTokenMultiOracle._peek/_get` function does the following computation on unsigned integers which reverts when `source.decimals > 18`:

```
price = uint(rawPrice) * 10 ** (18 - source.decimals);
```

Recommend instead performing this `price = uint(rawPrice) * 10 ** 18 / 10 ** source.decimals;`. Note that this leads to a loss of precision and the price could end up being `0`.

alcueca (Yield) acknowledged and patched:

> Fix

[L-09] `ERC20Rewards` claiming can fail if no reward tokens

Submitted by cmichel

The `ERC20Rewards` contract assumes that enough `rewardsToken` are in the contract to pay out when `claim` is called but this value is never checked and claiming rewards can fail.

Recommend that, when setting new rewards periods, to make sure that enough `rewardsToken`s are in the contract to cover the entire period.

alcueca (Yield) acknowledged:

> This is a known issue, which we prefer to leave as it is.

> Users can check if there are rewards tokens in the contract to cover the whole period, if they wish. ERC20Rewards.sol is intended to be inherited, so it depends on the implementation of the child contract whether that user check (or the proposed mitigation) could be trusted.

> In our intended use of ERC20Rewards, it is a governance action to make sure that there are funds to cover rewards at all times, which is easy to do since they are evenly distributed over time.

[L-10] improve safety of role constants

Submitted by gpersoon

The contract `Wand` defines a few role constants with `bytes4(keccak256("...function..."))` However if the function template would change slightly, for example when `uint128` is replaced by `uint256`, then this construction isn't valid anymore.

It is safer to use the function selector, as is done in EmergencyBrake.sol

```
    bytes4 public constant JOIN = bytes4(keccak256("join(address,
    bytes4 public constant EXIT = bytes4(keccak256("exit(address,
    bytes4 public constant MINT = bytes4(keccak256("mint(address,
    bytes4 public constant BURN = bytes4(keccak256("burn(address,
```

```

  _grantRole(IEmergencyBrake.plan.selector, planner);
```

Recommend using function selectors in `Wand.sol`.

alcueca (Yield) confirmed and patched:

> Fix

[L-11] `EmergencyBrake.sol` : Permissions cannot be re-planned after termination

Submitted by hickuphh3

Given a configuration of target, contacts, and permissions, calling `terminate()` , will permanently prevent this configuration from being used again because the state becomes `State.TERMINATED` . All other functions require the configuration to be in the other states (UNKNOWN, PLANNED, or EXECUTED).

In other words, the removal of the restoring option for the configuration through `EmergencyBrake` is permanent.

Recommend that, since `EmergencyBrake` cannot reinstate permissions after termination, it would be better to have terminate change its state to UNKNOWN. The TERMINATED state can therefore be removed.

alcueca (Yield) confirmed:

> That's right.

alcueca (Yield) patched:

> Fix

[L-12] `ERC20Rewards.sol` : Have a method to calculate the latest `rewardsPerToken` accumulated value

Submitted by hickuphh3

This would be equivalent to Unipool's `rewardPerToken()` function. Note that `rewardsPerToken.accumulated` only reflects the latest stored accumulated value, but does not account for pending accumulation like Unipool, and is therefore not the same. It possibly might be mistaken to be so, hence the low risk classification.

A possible implementation is given below.

```
function latestRewardPerToken() external view returns (uint256)
        RewardsPerToken memory rewardsPerToken_ = rewardsPerToke
        if (_totalSupply == 0) return rewardsPerToken_.accumulat
        uint32 end = earliest(block.timestamp.u32(), rewardsPeri
        uint256 timeSinceLastUpdated = end - rewardsPerToken_.la
        return rewardsPerToken_.accumulated + 1e18 * timeSinceLa
}
```

alcueca (Yield) confirmed:

> Thanks for the suggestion. Even if there is no risk, it will be nice to have this on frontends.

Non-Critical findings (9)

- [N-01] Incorrect type of uint parameter is used in event
- [N-02] Missing zero-address checks
- [N-03] Missing parameter validation
- [N-04] Multiple solc versions may be allowed
- [N-05] `CTokenMultiOracle.sol` - Add natspec documentation
- [N-06] `CTokenMultiOracle.sol` - require in `_setSource()` seems useless
- [N-07] `CompositeMultiOracle.sol` - Add natspec documentation
- [N-08] double negative in comment
- [N-09] `Timelock.sol` : Indexing targets array might not be useful

Gas Optimizations (18)

- [G-01] Storage slot packing impacts gas efficiency
- [G-02] Changing function visibility from public to external saves gas
- [G-03] Caching state variable in local variables for repeated reads saves gas by converting expensive SLOADs into much cheaper MLOADs

- [G-04] Using parameters or local variables instead of state variables in event emits can save gas
- [G-05] Not using memory data location specifier for external function parameters will save gas
- [G-06] Two functions with same code can be replaced by a single one
- [G-07] Redundant check
- [G-08] Check made redundant by following check
- [G-09] `FYTokenFactory.sol` - `fyToken.ROOT()` can be stored in a variable
- [G-10] `CTokenMultiOracle.sol` - `cTokenIds.length` in `setSources()` can be stored in a variable
- [G-11] `CompositeMultiOracle.sol` - bases.length in `setSources()` and `setPaths()` can be stored in a variable
- [G-12] Gas: `TimeLock.setDelay` reads storage variable for event
- [G-13] Gas: `ERC20Rewards._updateRewardsPerToken` return value is not needed
- [G-14] gas improvement in schedule and cancel of `TimeLock.sol`
- [G-15] gas improvement with `source.decimals`
- [G-16] Combine get and peek
- [G-17] `ERC20Rewards.sol` : `latest()` is unused
- [G-18] Gas optimization on `_updateRewardsPerToken` of `ERC20Rewards`

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