YIELD

Yield
Findings & Analysis Report

2021-07-23

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 Yield smart contract system written in Solidity. The code contest took place between May 26th and June 2nd, 2021.

Wardens

7 Wardens contributed reports to the Yield code contest:

- gpersoon
- Thunder
- shw
- 0xRajeev
- cmichel
- a_delamo
- 0xsomeone

This contest was judged by LSDan (dmvt). The final report was assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 47 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity, 9 received a risk rating in the category of MEDIUM severity, and 18 received a risk rating in the category of LOW severity.

C4 analysis also identified 17 non-critical recommendations.

Scope

The code under review can be found within the C4 Yield contest repository and comprises 55 smart contracts written in the Solidity programming language.

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 (3)

[H-01] Duplication of Balance

It is possible to duplicate currently held `ink` or `art` within a Cauldron, thereby breaking the contract's accounting system and minting units out of thin air.

The `stir` function of the `Cauldron`, which can be invoked via a `Ladle` operation, caches balances in memory before decrementing and incrementing. As a result, if a transfer to self is performed, the assignment `balances[to] = balancesTo` will contain the added-to balance instead of the neutral balance.

This allows one to duplicate any number of `ink` or `art` units at will, thereby severely affecting the protocol's integrity. A similar attack was exploited in the third bZx hack resulting in a roughly 8 million loss.

Recommend that a `require` check should be imposed prohibiting the `from` and `to` variables to be equivalent.

albertocuestacanada (Yield) confirmed:

> It is a good finding and a scary one. It will be fixed. Duplicated with #7.

[H-02] auth collision possible

The auth mechanism of `AccessControl.sol` uses function selectors `(msg.sig)` as a `(unique)` role definition. Also the `_moduleCall` allows the code to be extended.

Suppose an attacker wants to add the innocent-looking function "`left_branch_block(uint32)` "in a new module. Suppose this module is added via `_moduleCall` , and the attacker gets authorization for the innocent function.

This function happens to have a signature of 0x00000000, which is equal to the root authorization. In this way, the attacker could get authorization for the entire project.

Note: it's pretty straightforward to generate function names for any signature value; you can just brute force it because it's only 4 bytes.

Recommend not allowing third parties to define or suggest new modules and double-checking the function signatures of new functions of a new module for collisions.

albertocuestacanada (Yield) confirmed:

> The execution of any `auth` function will only happen after a governance process or by a contract that has gone through a thorough review and governance process.

> We are aware that new modules can have complete control of the Ladle, and for that reason, the addition of new modules would be subject to the highest level of scrutiny. Checking for signature collisions is a good item to add to that process.

> In addition to that, I would implement two changes in `AccessControl.sol` so that giving ROOT access is explicit.

```
    function grantRole(bytes4 role, address account) external vi
        require(role != ROOT, "Not ROOT role");
        _grantRole(role, account);
    }
>
    function grantRoot(address account) external virtual admin(R
        _grantRole(ROOT, account);
    }
```

> However, given that this could be exploited only through a malicious governance exploit, I would reduce the risk to “Low.”

albertocuestacanada (Yield) acknowledged:

> After further thinking, instead of preventing auth collisions in the smart contracts, we will add CI checks for this specific issue instead.

[H-03] YieldMath.sol / Log2: >= or > ?

The V1 version of `YieldMath.sol` contains ”>=” (larger or equal), while the V2 version of `YieldMath.sol` contains ”>” (larger) in the log_2 function. This change doesn’t seem logical and might lead to miss calculations. The difference is present in several adjacent lines.

```
function log_2 (uint128 x)
...
b = b * b >> 127; if (b >= 0x100000000000000000000000000000000)
```

and

```
function log_2(uint128 x)
...
b = b * b >> 127; if(b > 0x100000000000000000000000000000000) {k
```

Recommend checking which version is the correct version and fix the incorrect version.

albertocuestacanada (Yield) confirmed:

> That's entirely my fault, and this is a scary one. We might be having a slightly different or impredictable curve in Pool.sol, and we might notice only after a long while with the Pools being slowly drained. We might never even have found this was the issue.

> I would suggest increasing the severity of this issue to High.

albertocuestacanada (Yield) Resolved:

> Fix

Medium Risk Findings (9)

[M-01] Potential griefing with DoS by front-running vault creation with same `vaultID`

The `vaultID` for a new vault being built is required to be specified by the user building a vault via the `build()` function (instead of being assigned by the Cauldron/protocol). An attacker can observe a `build()` as part of a batch transaction in the mempool, identify the `vaultID` being requested, and front-run that by constructing a malicious batch transaction with only the build operation with that same `vaultID`. The protocol would create a vault with that `vaultID` and assign the attacker as its owner. More importantly, the valid batch transaction in the mempool, which was front-run, will later fail to create its vault because that `vaultID` already exists, as per the check on Line180 of `Cauldron.sol`. As a result, the valid batch transaction fails entirely because of the attacker front-running with the observed `vaultID`.

While the attacker gains nothing except the ownership of an empty vault after spending the gas, this could grief the protocol's real users by preventing them from opening a vault and interacting with the protocol in any manner.

The rationale for Medium-severity impact: While the likelihood of this may be low, the impact is high because valid vaults from the Yield front-end will never be successfully created and will lead to a DoS against the entire protocol's functioning. So, with low likelihood and high impact, the severity (according to OWASP) is medium.

Alice uses Yield's front-end to create a valid batch transaction. Evil Eve observes that in the mempool and identifies the `vaultID` of the vault being built by Alice. Eve submits her own batch transaction (without using the front-end) with only a build operation using Alice's `vaultID`. She uses a higher gas price to front-run Alice's transaction and get's the protocol to assign that `vaultID` to herself. Alice's batch transaction later fails because the `vaultID` she requested is already assigned to Eve. Eve can do this for any valid transaction to grief protocol users by wasting her gas to cause DoS.

Recommend mitigating this DoS vector by having the `Cauldron` assign the `vaultID` instead of the user specifying it in the `build()` operation. This would likely require the `build()` to be a separate non-batch transaction followed by other operations that use the `vaultID` assigned in `build()`. Consider the pros/cons of this approach because it will significantly affect the batching/caching logic in `Ladle`.

Alternatively, consider adding validation logic in `Ladle's` batching to revert batches that have only build or a subset of the operations that do not make sense to the protocol's operations per valid recipes, which could be an attacker's signature pattern.

albertocuestacanada (Yield) confirmed:

> Good find. Fix

[M-02] Uniswap Oracle uses wrong prices

The Uniswap oracle uses a mock contract with hard-coded prices to retrieve the price, which is not feasible in production. Also, note that even when using the "real deal" `@uniswap/v3-periphery/contracts/libraries/OracleLibrary.sol` ... it does not, in fact, return the prices.

The price could change from the set price. Meanwhile, always updating new prices with `set` will be too slow and gas expensive.

Recommend using `cumulativeTicks = pool.observe([secondsAgo, 0]) // [a_t1, a_t2]` and applying **equation 5.5** from the Uniswap V3 whitepaper to compute the token0 TWAP. Note that even the **official** `.consult` **call** seems to only return the averaged cumulative ticks; you'd still need to compute the `1.0001^timeWeightedAverageTick` in the function.

albertocuestacanada (Yield) acknowledged:

> We probably should have not included this contract; it's too confusing since, at the time, the Uniswap v3 OracleLibrary was still a mock, and this hasn't gone real testing.

> The price source in a production version would be a Uniswap v3 pool, not one of our mock oracle sources. We never expected to call `set` in production, but to retrieve the prices from a Uniswap v3 pool using the mentioned library (which was not even merged into main at the start of the contest).

> We will check with the Uniswap team what is the recommended way of using their oracles. The equation 5.5 in the whitepaper is problematic because an exponentiation of two fractional numbers in Solidity is neither trivial nor cheap. Our understanding is that one of the goals of the OracleLibrary was to provide a consistent implementation to this formula.

> From a conversation with @moodysalem, I understand that the code in `getQuoteAtTick` might achieve the same result as the 5.5 equation, so maybe we need to retrieve the average tick with `consult`, and then the actual price with `getQuoteAtTick`.

albertocuestacanada (Yield) commented:

> I'm using the `acknowledged` label for findings that require further investigation to assess.

[M-03] Witch can't give back vault after 2x grab

The `witch.sol` contract gets access to a vault via the grab function in case of liquidation. If the `witch.sol` contract can't sell the debt within a certain amount of time, a second grab can occur.

After the second grab, the information of the original owner of the vault is lost, and the vault can't be returned to the original owner once the debt has been sold.

The `grab` function stores the previous owner in `vaultOwners[vaultId]`, and then the contract itself is the new owner (via `cauldron.grab` and `cauldron._give`). The `vaultOwners[vaultId]` is overwritten at the second grab

The function buy of `Witch`.sol tried to give the vault back to the original owner, which won't succeed after a second grab. **See the issue page for proof of concept and referenced code**

Assuming it's useful to give back the vault to the original owner, recommend making a stack/array of previous owners if multiple instances of the `witch.sol` contract would be used. Or, check if the witch is already the owner (in the `grab` function) and keep the `vaultOwners[vaultId]` if that is the case.

albertocuestacanada (Yield) confirmed:

> This is a good finding and a vulnerability that we will fix. I anticipate that we will store the original owner in `Cauldron.auctions` along with the time at which the auction was started.

albertocuestacanada (Yield) commented:

> Actually, we might remove some overengineering by taking out the feature of allowing multiple competing liquidation engines.

> That would allow us to:

1. Move the `auctions` mapping to the `Witch`.
2. Remove the `auctionInterval` setting
3. Simplify `grab`
4. Stop worrying about more than one `grab`.

Editors note: An alternative submission for this bug was not included in this report but can be found in Issue #30.

[M-04] User can redeem more tokens by artificially increasing the chi accrual

A user can artificially increase the chi accrual (after maturity) by flash borrow on Compound, which affects the exchange rate used by the chi oracle. As a result, the user redeems more underlying tokens with the same amount of fyTokens since the accrual is larger than before.

The `exchangeRateStored` used by chi oracle is calculated based on the `totalBorrows` of `CToken`, which can be artificially increased by a large amount of borrow operated on Compound. Consider a user performing the following steps in a single transaction (assuming that the fyToken is matured):

1. Deposits a large amount of collateral (whether from flash loans or not) and borrow from Compound
2. Burns his fyToken by calling `redeem`.
3. Repays the borrow to Compound

The user only needs to pay for the gas fees of borrowing and repaying (since they happen in the same transaction) but can redeem more underlying tokens than a regular redeem.

Referenced code: CompoundMultiOracle.sol#L46 FYToken.sol#L125 FYToken.sol#L132-L143

Recommend making the chi accrual time-weighted to mitigate the manipulation caused by flash borrow and repay.

albertocuestacanada (Yield) acknowledged:

> If this is true, that means that we don't understand how `exchangeRateStored` works (quite likely).

> Our understanding was that `exchangeRateStored` is an increasing accumulator, same as `chi` in MakerDAO, which is both an exchange rate and an accumulator.

> If `exchangeRateStored` can go down in value, as well as up, we might have to revisit how we source it.

[M-05] Uninitialized or Incorrectly set `auctionInterval` may lead to liquidation engine livelock

The `grab()` function in Cauldron is used by the Witch or other liquidation engines to grab vaults that are under-collateralized. To prevent re-grabbing without sufficient time for auctioning collateral/debt, the logic uses an `auctionInterval` threshold to give a reasonable window to a liquidation engine that has grabbed the vault.

The `grab()` function has a comment on Line 354: "// Grabbing a vault protects it for a day from being grabbed by another liquidator. All grabbed vaults will be suddenly released on the 7th of February 2106, at 06:28:16 GMT. I can live with that." indicating a requirement of the `auctionInterval` being equal to one day. This can happen only if the `auctionInterval` is set appropriately. However, this state variable is uninitialized (defaults to 0) and depends on `setAuctionInterval()` being called with the appropriate `auctionInterval_` value, which is also not validated.

Discussion with the project lead indicated that this comment is incorrect. Nevertheless, it is safer to initialize `auctionInterval` at declaration to a safe default value instead of the current 0, which will allow liquidation engines to re-grab vaults without making any progress on liquidation auction. It is also good to add a threshold check-in `setAuctionInterval()` to ensure the new value meets/exceeds a reasonable default value.

The rationale for Medium-severity impact: While the likelihood of this may be low, the impact is high because liquidation engines will keep re-grabbing vaults from each other and potentially result in liquidation bots entering a live-lock situation without making any progress on liquidation auctions. This will result in collateral

being stuck and impact the entire protocol's functioning. So, with low likelihood and high impact, the severity (according to OWASP) is medium.

See Issue page for proof of concept and referenced code.

Recommend:

1. Initialize `auctionInterval` at declaration with a reasonable default value.
2. Add a threshold check in `setAuctionInterval()` to ensure the new value meets/exceeds a reasonable default value.

albertocuestacanada (Yield) confirmed but disagreed with severity:

> While we support multiple liquidation engines, we are going live with only one, and only one is in the scope of the contest.

> Zero is an acceptable value for `auctionInterval` using just one liquidation engine (the Witch). The issue can be solved with better natspec and better documentation; therefore, a severity of 0 is suggested.

albertocuestacanada (Yield) commented:

> Actually, if we forget to set `auctionInterval` to a reasonable value, an attacker could repeatedly call `grab` with just one Witch, stopping auctions from dropping down in price until an emergency governance action is taken.

> Therefore, I would suggest the severity be reduced to 1 instead of zero.

[M-06] Violation of implicit constraints in batched operations may break protocol assumptions

The Ladle batching of operations is a complex task (as noted by the project lead) with implicit constraints on what operations can be bundled together in a batch. Operations can/have to appear how many times and in what order/sequence etc. Some examples of these constraints are: `Join Ether` should be the first operation, `Exit Ether` the last, and only one `Join Ether` per batch.

All this complexity is managed currently by anticipating all interactions to happen via their authorized front-end, which uses validated (and currently only revealed on-demand) recipes that adhere to these constraints. There is a plan to open the design up to other front-ends and partner integrating protocols that will also test their batch recipes or integrations for these constraints.

Breaking some of these constraints opens up the protocol to failing transactions, undefined behavior, or potentially loss/lock of funds. Defensive programming suggests enforcing such batch operation constraints in the code and documentation and onboarding checks for defense-in-depth. Relying on documentation or external validation may not be sufficient for arguably the most critical aspect of batched operations which is the only authorized way to interact with the protocol.

The rationale for assigning medium-severity is that, while the likelihood of this may be low because of controlled/validated onboarding on new front-ends or integrating protocols, the impact of accidental deviation from implicit constraints is high. This may result in a transaction failing, or tokens getting locked/lost, thus impact the entire protocol's functioning. So, with low likelihood and high impact, the severity (according to OWASP) is medium.

1. A new front-end project comes up claiming to provide a better user interface than the project's authorized front-end. It does not use the recipe book (correctly) and makes Ladle batches with incorrect operations, thus failing the constraints and leading to protocol failures and token lock/loss.
2. An integrating protocol goes through the approved onboarding and validation but has missed bugs in its recipe for batches, thus failing the constraints and leading to protocol failures and token lock/loss.

Recommend enforcing batch operation constraints explicitly in the code (e.g., with tracking counters/booleans for operations) along with documentation and onboarding validation. This may increase the complexity of the batching code but adds fail-safe defense-in-depth for any mistakes in onboarding validation of implicit constraints, which may affect protocol operations significantly.

albertocuestacanada (Yield) disputed:

> I don't think implementing the suggested checks is worth the added complexity and deployment gas.

> Given that this is an issue disclosed by the sponsor and widely discussed, I would find it unfair to the other wardens if accepted.

dmvt (Judge) commented:

> The warden brings up a valid concern. As to the sponsor's objection, the sponsor has written, "The Ladle is a complex contract, with wide-ranging powers over the protocol. **It is the riskiest contract, and at the same time, the one that has more room for a bug to hide. It is of the highest importance to find as many bugs as we can in it.**" (emphasis mine). I highly recommend the issue be addressed by the sponsor.

albertocuestacanada (Yield) commented:

> I still disagree. I disclosed this issue myself first in the C4 discord, and it is a design choice.

> Users are not expected to build batches themselves and won't be provided with any tools to do so. If a user decides to ignore all advice and go to the extreme length of interacting with the smart contracts himself or via an unauthorized front-end, it's completely on him if he loses funds.

> It is impossible that a user will execute a non-reverting batch without careful research on the batching system. To do that, he would need to exactly match one or more action codes with abi-encoded arrays of parameters of the right types.

> My concerns on the batching system are not on users or integrators building a bad batch and losing their funds.

> My concerns are that the batching system is complex, and there could be the chance that a batch could be built with a negative effect on the protocol or for other users. Since the Ladle has sweeping powers over the protocol, that would be a real issue. Such a batch has not been found.

> So, to reiterate. The issue being brought up was brought up in the discord by myself, and it is a conscious trade-off we made of usability for flexibility. No undisclosed issue has been found.

[M-07] Possible DoS attack when creating `Joins` in `Wand`

It is possible for an attacker to intendedly create a fake `Join` corresponding to a specific token beforehand to make `Wand` unable to deploy the actual `Join`, causing a DoS attack.

The address of `Join` corresponding to an underlying `asset` is determined as follows and thus unique:

```
Join join = new Join{salt: keccak256(abi.encodePacked(asset))}()
```

Besides, the function `createJoin` in the contract `JoinFactory` is permissionless: Anyone can create the `Join` corresponding to the `asset`. An attacker could then deploy many `Joins` with different common underlying assets (e.g., DAI, USDC, ETH) before the `Wand` deploying them. The attempt of deploying these `Joins` by `Wand` would fail since the attacker had occupied the desired addresses with fake `Joins`, resulting in a DoS attack.

Moreover, the attacker can also perform DoS attacks on newly added assets: He monitors the mempool to find transactions calling the function `addAsset` of `Wand` and front-runs them to create the corresponding `Join` to make the benign transaction fail.

Referenced code: JoinFactory.sol#L64-L75 Wand.sol#L53

Recommend enabling access control in `createJoin` (e.g., adding the `auth` modifier) and allowing `Wand` to call it.

albertocuestacanada (Yield) confirmed:

> The issue exists, and we appreciate raising it.

> The solution can't be adding `auth` to `createJoin`, since anyone can use CREATE2 to deploy a Join with their own factory, but with our Join bytecode, occupying the same address we would.

> The proper mitigation, in our opinion, is to ditch CREATE2 and deploy Joins using CREATE instead.

> As for the risk, such a DoS attack wouldn't cause a loss of funds or an interruption on user service. It would cause a governance action to revert, which would be quickly fixed by deploying a new JoinFactory and replacing the Wand. Fortunately, no contract uses the Wand as a Join registry (maybe we should!).

> I suggest the risk is downgraded to 1.

[M-08] Users can avoid paying borrowing interest after the fyToken matures

According to the protocol design, users have to pay borrowing interest when repaying the debt with underlying tokens after maturity. However, a user can give his vault to `Witch` and then buy all his collateral using underlying tokens to avoid paying the interest. Besides, this bug could make users less incentivized to repay the debt before maturity and hold the underlying tokens until liquidation.

1. A user creates a new vault and opens a borrowing position as usual.
2. The maturity date passed. If the user wants to close the position using underlying tokens, he has to pay a borrowing interest (line 350 in `Ladle`), which is his debt multiplied by the rate accrual (line 373).
3. Now, the user wants to avoid paying the borrowing interest. He gives his vault to `Witch` by calling the function `batch` of `Ladle` with the operation `GIVE`.
4. He then calls the function `buy` of `Witch` with the corresponding `vaultId` to buy all his collateral using underlying tokens.

In the last step, the `elapsed` time (line 61) is equal to the current timestamp since the vault is never grabbed by `Witch` before, and thus the auction time of the vault, `cauldron.auctions(vaultId)`, is 0 (the default mapping value). Therefore, the collateral is sold at a price of `balances_.art/balances_.ink` (line 74). The user can buy `balances_.ink` amount of collateral using `balances_.art` but not paying for borrowing fees.

Recommend not allowing users to `give` vaults to `Witch`. And to be more careful, requiring `vaultOwners[vaultId]` and `cauldron.auctions(vaultId)` to be non-zero at the beginning of function `buy`.

albertocuestacanada (Yield) confirmed:

> That's a good catch. The mitigation steps are right to avoid this being exploited by malicious users, but it would be better to fix the underlying issue.

> The problem is that the Witch always applies a 1:1 exchange rate between underlying and fyToken, which is not true after maturity. As long as this is not fixed, the protocol will lose money after maturity liquidations.

albertocuestacanada (Yield) commented:

> More specifically, `_debtInBase` should be a Cauldron public function, and return (debtInFYToken, debtInBase). The Ladle would save a bit in deployment gas; the Witch would use it to determine the underlying / fyToken exchange rate.

albertocuestacanada (Yield) commented:

> However, since `grab` wouldn't be called on the Witch, the vault owner wouldn't be registered. With the liquidation being a Dutch auction, the vault owner would only get a portion of his collateral back after paying all the debt. The vault with the remaining collateral would be given to address(0).

> There is a small protocol loss from miscalculated vault debt on vaults liquidated after maturity, but no user funds would be at risk.

> I would propose a severity of 2, given there are monetary losses, however slight, to the protocol. The attack described can't happen, but it revealed a real issue.

[M-09] auth only works well with external functions

The auth modifier of `AccessControl.sol` doesn't work as you would expect. It checks if you are authorized for `msg.sig`. However, `msg.sig` is the signature of the first function you have called (not the current function). So, if you call function A, which calls function B, the "auth" modifier of function B checks if you are authorized for function A!

There is a difference between external and public functions. For external functions, this works as expected because a fresh call (with a new `msg.sig`) is always made. However, with public functions called from within the same contract, this doesn't happen as the problem described above occurs.

See the issue page for proof of concept, which shows the problem. In the code, several functions have `public` and `auth` combined; see also in the proof of concept.

In the current codebase, I couldn't find a problem situation; however, this could be accidentally introduced with future changes. It could also be introduced via the `_moduleCall` of Ladle.sol, which allows functions to be defined which might call the public functions.

See **issue #4 page** for proof of concept and a list of occurrences of public auth.

Recommend making sure all auth functions use external (still error-prone) Or, recommend changing the modifier to something like:

```
modifier auth(bytes4 fs) {
require (msg.sig == fs,"Wrong selector");
require (_hasRole(msg.sig, msg.sender), "Access denied");
_;
}
```

```
function setFee(uint256) public auth(this.setFee.selector) {
   .....
}
```

albertocuestacanada (Yield) confirmed:

> While many governance functions have been marked `public` instead of `external` throughout the code, they are never called from any other function in the same contract, and they should never be.

> In all contracts of the protocol, `auth` functions are only called by other contracts or by EOAs, with the latter being governance actions.

> The suggestion of changing all `public auth` functions to `external auth` will be applied; however, no changes will be made to AccessControl.sol, and the suggested severity is 1.

Low Risk Findings (18)

[L-01] Missing checks on debt max/min limits could cause pour to revert

`setDebtLimits()` is used to set the maximum and minimum debt for an underlying and ilk pair. The assumption is that max will be greater than min while setting them because otherwise, the debt checks in `_pour()` for line/dust will fail and revert.

While max and min debt limits can be reset, it is safer to perform input validation on them in `setDebtLimits()`.

A recipe incorrectly interchanges the values of min and max debt, which leads to exceptions in pouring into the vaults.

Recommend adding a check to ensure max > mix.

albertocuestacanada (Yield) confirmed:

[L-02] Return values of batch operations are ignored

Many batched operation functions return values ignored by the caller `batch()`. While this may be acceptable for the front-end, which picks up any state changes from such functions via emitted events, integrating protocols that make a call to `batch()` may require it to package and send back return values of all operations from the batch to react on-chain to the success/failure or other return values from such calls. Otherwise, they will be in the dark on the success/impact of batched operations they've triggered.

See issue #46 for code examples.

Recommend packaging and sending back return values of all batched operations' functions to the caller of `batch()`.

albertocuestacanada (Yield) confirmed and resolved

[L-03] Missing sender address check in `receive()` may lead to locked Ether

Recommend adding an address check in `receive()` of `Ladle.sol` to ensure the only address sending ETH being received in `receive()` is the Weth9 contract (similar to the check in `PoolRouter.sol`) for Ether withdrawal in `_exitEther()` as this will prevent stray Ether from being sent accidentally to this contract and getting locked.

albertocuestacanada (Yield) confirmed

[L-04] Missing reentrancy guard and contract existence check for modules

The protocol allows users to call registered modules via `delegateCall` in Module operation. It is not clear how these modules are validated before registration. If they are malicious, they could cause reentrancy in the `batch()` call because there is no reentrancy guard protection. If they are destructed, `delegateCall` will still return success because low-level calls do not check for contract existence. Both will cause an undetermined level of impact to the protocol, but the likelihood is low given the registration process and assumed validation there.

Eve manages to get her malicious module registered, which causes reentrancy or maliciously affects protocol accounts/operations due to the `delegateCall`.

Alternatively, Alice registers a benign module but then accidentally calls `selfDestruct` on it. The module delegation is successful but without any side effects because it doesn't exist anymore. See issue page for referenced code.

Recommend:

1. Ensure during module registration that modules are trustworthy without any possibility to cause reentrancies or self-destruct.
2. Add reentrancy guard on `batch()` and contract existence check on module before `delegateCall`

albertocuestacanada (Yield) confirmed:

> We know that using modules is inherently dangerous, and their permissioning will be subject to the highest level of scrutiny.

uivlis commented:

> This ticket's fix requires no PR, so I will attach no link for the fix.

[L-05] Prevent the use of LOCK in `setRoleAdmin` to instead force the use of `lockRole`

The LOCK role is special in `AccessControl` because it has itself as the admin role (like ROOT) but no members. This means that calling `setRoleAdmin(msg.sig, LOCK)` "means no one can grant/revoke that msg.sig role anymore, and it gets locked irreversibly. This means it disables admin-based permissioning management of that role and therefore is very powerful in its impact.

Given this, there is a special function `lockRole()`, which is specifically meant to enforce LOCK as the admin for the specified role parameter. For all other role admin creations, the generic `setRoleAdmin()` may be used. However, `setRoleAdmin()` does not prevent specifying the use of LOCK as the admin. If this is accidentally used, it leads to disabling that role's admin management, which is irreversibly similar to the `lockRole()` function.

It is safer to force admins to use `lockRole()` as the only way to set admin to LOCK and prevent the use of LOCK as the `adminRole` parameter in `setRoleAdmin()`, because doing so will make the intention of the caller clearer as `lockRole()` clearly has that functionality specified in its name and that's the only thing it does.

Alice who is the admin for `foo()` wants to give the admin rights to Bob (0xFFFFFFF0) but instead of calling `setRoleAdmin(foo.sig, 0xFFFFFFF0)`, she calls `setRoleAdmin(foo.sig, 0xFFFFFFFF)` where 0xFFFFFFFF is LOCK. This makes LOCK as the admin for `foo()` and prevents any further admin-based access control management for `foo()`.

Recommend preventing the use of LOCK as the `adminRole` parameter in `setRoleAdmin()`.

albertocuestacanada (Yield) confirmed

[L-06] Incompatibility With Rebasing/Deflationary/Inflationary tokens

Yield whitelists a rebasing/deflationary/inflationary token to be used as collateral or underlying by accident. This leads to miscalculations between internal `Cauldron` accounting and the balances in the token contracts.

Yield protocol allows different tokens to be used as collateral or underlying. The `Join` and `Pool` contracts do not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the number of tokens transferred to contracts before and after the actual transfer to infer any fees/interest. These seem to be absent as seen in the `_join()` call of `Join.sol` or in Pool contracts. The impact will be miscalculations between internal `Cauldron` accounting and the balances in the token contracts.

Yield currently manages this by approving only certain tokens to be used as collateral or underlying. Therefore, this is not an issue now as long as the tokens are determined to not be of the concerned kinds. However, this will become an issue if user-supplied tokens are accepted without the existing vetting.

Recommend:

1. Make sure token vetting accounts for any rebasing/inflation/deflation
2. Add support in contracts for such tokens before accepting user-supplied tokens

albertocuestacanada (Yield) acknowledged:

> We'll take option 1, and ban these tokens from Yield.

uivlis commented:

> This ticket's fix requires no PR, so I will attach no link for the fix.

[L-07] `flashFeeFactor` is uninitialized at declaration leading to zero-fee flash loans enabled by default

`flashFeeFactor` is uninitialized at declaration and so zero initially until set by `setFlashFeeFactor()`. As indicated in one of the explainer videos, the idea is to default to uint256.max to disable flash loans by default.

Currently, flash loans are enabled by default with a zero flash fee unless changed by `setFlashFeeFactor()`.

Recommend Initializing at declaration with a reasonable value which could be uint256.max to disable flash loans by default.

albertocuestacanada (Yield) confirmed:

> The intention was to have flash loans disabled by default, but I see no risk in having zero-fee flash loans.

[L-08] Multiple compiler versions allowing a wide range from 0.5.0 to >=0.8.0

The project uses multiple compiler versions with most specifying ^0.8.0, some specifying >=0.8.0, which allows breaking versions >= 0.9.0 in the future if reused/redeployed, and some even allowing much older >= 0.5.0/0.6.0.

The dangers of allowing multiple compilers across breaking revisions are that the security bug fixes and features might differ across different contracts introducing vulnerabilities or giving a false sense of security.

For example, most contracts use ^0.8.0, which means they have default-checked arithmetic to prevent overflows/underflows without using OZ SafeMath. This doesn't apply to the few `(inherited)` contracts that may be compiled with <0.8.0 and have unchecked overflows/underflows.

Recommend:

1. Update all contracts to use pragma solidity ^0.8.0 or better a fixed version like 0.8.4

2. Deploy with the same compiler version which was used for testing

alberocuestacanada (Yield) confirmed

[L-09] Anyone can create a fake pool to trick unauthorized front-ends

While many Yield protocol functions are authorized to be called from other Yield contracts, the `createPool()` function in `PoolFactory` is not authorized and is callable from anyone. This is only supposed to be called from the Wand function `addSeries()`, which takes a base asset, creates a corresponding fyToken, and then a pool with the two of them. Pool contracts are created deterministically using `Create2`, but fyTokens are not. If the fyToken were also created deterministically using `Create2`, or if the Wand created fyToken and the pool in two different transactions, thus making the fyToken address observable to an attacker, the attacker could front-run pool creation to deploy a fake pool and make the real pool creation fail. This is currently not possible.

However, anyone can create a fake pool with the real base asset address but a fake fyToken. If the threat model is extended to integrating contracts/front-ends that may assume all pools created (as indicated by emitted events) by `createPool` to be authentic ones (like the one created from the Wand), they may end up interacting or allow interactions with a fake pool which may lead to users losing funds.

Alice creates a front-end that observes `PoolCreated` events emitted by `createPool` to automatically list them as Yield pools. An attacker creates a fake pool with Dai as base asset and a corresponding fake/malicious fyToken contract. Users of Alice's front-end end up interacting with the fake Yield pool and losing funds.

Recommend considering adding auth to the `createPool` function and permissioning only `Wand` to access this function for creating pools.

albertocuestacanada (Yield) disputed:

> We can't stop users from deploying pools since we use Create2. Anyone can deploy their own PoolFactory, and if they use the same bytecode, their pools will be available when `getPool` in our PoolFactory is used.

> That said, in no way can we make ourselves responsible for unauthorized front-ends.

[L-10] In method `_update` on `Pool.sol` - Divide before multiply

In the `Pool.sol` contract, there is the following code:

```
    function _update(
            uint128 baseBalance,
            uint128 fyBalance,
            uint112 _baseCached,
            uint112 _fyTokenCached
        ) private {
            ....

                cumulativeBalancesRatio +=
                    (scaledFYTokenCached / _baseCached) *
                    timeElapsed;
            ....
        }
```

The multiplication should always be placed at the end to avoid miscalculations like the following one:

```
a = (b/d)*c
0 = (5/10)*2

a = (b * c)/ 2
1 = (5 * 2)/10
```

- albertocuestacanada (Yield) confirmed

[L-11] Implicit unsafe math

`Ladle._close` (and many other occurrences) reverts the transaction on certain signed inputs that are negated and cast to unsigned integers.

```
// Ladle._close calling it with art or ink as type(int128).min w
uint128 amt = _debtInBase(vault.seriesId, series, uint128(-art))
ilkJoin.exit(to, uint128(-ink))

// explanation
int128 art = type(int128).min; // -2^127
uint128 amt = uint128(-art); // this fails as -art=--2^127=2^127
```

Other places: `CauldronMath.add`, `Ladle._pour`, everywhere where `-int*` is used...

One cannot use the actual `type(int128).min` value for function parameters.

Recommend reverting with a meaningful error message as is done in the `/math/Cast*` functions.

albertocuestacanada (Yield) confirmed:

> You are right; I must have got my boundaries confused.

albertocuestacanada (Yield) commented:

> However, this will only cause rare transactions that would revert to still revert, just with a more meaningful message. Is this a non-critical or low risk?

[L-12] Unsafe call to `.decimals`

The `FYToken.constructor` performs an external call to `IERC20Metadata(address(IJoin(join_).asset())).decimals()`. This function was optional in the initial ERC-20 and might fail for old tokens that did not implement it.

FyTokens cannot be created for tokens that implemented the old initial ERC20 without the `decimals` function.

Recommend considering using the helper function in the utils to retrieve it `SafeERC20Namer.tokenDecimals`, the same way the `Pool.constructor` works.

albertocuestacanada (Yield) confirmed:

> Thanks!

[L-13] `_burnInternal` always returns 0 for fy tokens returned

Function `_burnInternal` always returns 0 as a third parameter. It should return `tokensBurnt`, `tokenOut`, `fyTokenOut`.

Recommend returning (`tokensBurned`, `tokenOut`, `fyTokenOut`);

albertocuestacanada (Yield) confirmed:

> True, thanks!

[L-14] ERC20 `approve` is vulnerable to the front-running

The function `approve` is vulnerable to the front-running. This issue is described here: https://blog.smartdec.net/erc20-approve-issue-in-simple-words-a41aaf47bca6. A malicious delegate can scout for a change in approval and front-run that. It is more of a theoretical issue, but still, I want you to be aware of this and that.

Recommend introducing `increaseAllowance` / `decreaseAllowance functions`.

albertocuestacanada (Yield) disputed: see here for full explanation.

[L-15] external function `transferToPool` is pretty useless

External function `transferToPool` is pretty useless and error-prone. It relies on the user not to leave these tokens in a separate tx; otherwise, it will just be feeding the bots. To use it directly, users will have to write their own custom smart contract and chain actions.

It would be better to remove this function, leaving the only way to invoke it via a batch function.

albertocuestacanada (Yield) confirmed

[L-16] function `redeem` should return "redeemed" amount

Function `redeem` in contract `FYToken` should return "redeemed" amount. The return value is not used anywhere, but it's a mistake that it assigns 'redeemed' but returns 'amount'.

Recommend removing return sentence or explicitly returning `redeemed`.

albertocuestacanada (Yield) confirmed

[L-17] Using stale `cToken` exchange rate

The chi oracle in contract `CompoundMultiOracle` calls the function `exchangeRateStored` rather than `exchangeRateCurrent` to get the exchange rate from Compound. However, since the function `exchangeRateStored` does not accrue interest before calculating the exchange rate, the return data could be out-of-date and affect the results of `_mature` and `_accrual` in the contract `FYToken`.

Recommend using `exchangeRateStored` in the `peek` function (since it does not allow transactional operations), and `exchangeRateCurrent` in the `get` function of `CompoundMultiOracle`.

albertocuestacanada (Yield) acknowledged:

> We discussed using stale exchange rates for `chi` and `rate` for Yield v1 in exchange for lower gas consumption. We chose then that stale rates were preferable for our use case, and I would expect this to still be the case. However, we will revisit this issue.

> We might find that the impact of using stale rates is low enough not to constitute a risk at all.

[L-18] Missing zero-address validations

While the codebase does a great job of input validation for parameters of all kinds (especially addresses), there are a few places where zero-address validations are missing. Even though none of them are catastrophic, resulting in obvious reverts,

and can be reset given the permissioned/controlled interactions with the contracts. Nevertheless, it is helpful to add zero-address validations to be consistent and ensure the high availability of the protocol with resistance to accidental misconfigurations. See the issue page for more details.

albertocuestacanada (Yield) confirmed but disputed severity:

> While these checks could be added, I don't think they are worth the gas. To me, it seems something should be checked in the front-end, or for governance actions, on spell review.

Non-Critical Findings (12)

[N-01] Use `.selector` instead of hex number

In the contract `SafeERC20Namer.sol`, a few function selects are encoded as hexadecimal numbers. Solidity also has the keyword `.selector`, making the code easier to read and less error-prone.

Recommend alternative implementations, e.g.:

- **IERC20Metadata.symbol.selector:** `0x95d89b41 = bytes4(keccak256("symbol()"))`
- **IERC20Metadata.name.selector:** `0x06fdde03 = bytes4(keccak256("name()"))`
- **IERC20Metadata.decimals.selector:** `0x313ce567 = bytes4(keccak256("decimals()"))`

albertocuestacanada (Yield) confirmed:

> That's right, thanks for the finding and the suggestion.

[N-02] `PoolFactory` and `JoinFactory` very similar

`PoolFactory` and `JoinFactory` contain very similar but also relatively complicated code. The risk is that future changes/improvements in one contract might not be

updated in the other.

Recommend considering refactoring the code where the core code is put in a library and reused from both contracts.

albertocuestacanada (Yield) confirmed:

> It is true that there is an opportunity to code reuse and the implementation of a generic CREATE2 factory, and we will explore that. However, there is no relationship between JoinFactory and PoolFactory, and therefore no guarantee that changes in one of them need to be considered in the other.

[N-03] Avoid assembly in `getRevertMsg`

The function `getRevertMsg` of `RevertMsgExtractor.sol` uses assembly to retrieve revert information. The latest solidity version has new functions that allow you to retrieve information without assembly.

Recommend piece of code below showing the new functionality (see Issue page for details).

albertocuestacanada (Yield) confirmed:

> Thanks for the finding. We will implement the proposed solution and if the deployment costs in the Pool are reasonable, merge it to `main`.

[N-04] Use constants for numbers

In several locations in the code, numbers like 1e12, 1e18, 1e27 are used. The same goes for values like: `type(uint256).max` It quite easy to make a mistake somewhere, also when comparing values.

So, recommend defining constants for the numbers used throughout the code.

albertocuestacanada (Yield) confirmed:

> Will do, thanks.

[N-05] Several todos left in the code

The code still has some todos, which should be resolved before production, recommend checking and fixing or remove the todos. (see issue page for list)

albertocuestacanada (Yield) confirmed:

> Thanks, will do.

[N-06] Useless `auth` modifier in `setSources`

Function `setSources` in Oracle contracts does not need an 'auth' modifier as it will be checked anyway in function `setSource`. This does not impact the security; it is just a useless check that can be removed. Recommend removing 'auth' modifier from function `setSources`.

albertocuestacanada (Yield) confirmed:

> The issue exists, but I think we'll fix it keeping two `setSource` and `setSources` `external auth` functions and one `_setSource` `internal` function.

[N-07] enum `TokenType` is never used

enum `TokenType` in library `PoolDataTypes` is not used anywhere. Either remove it or use it where intended.

albertocuestacanada (Yield) confirmed:

> Will remove, thank you!

[N-08] no need for `transferToPool` to be payable

Function `transferToPool` is marked as 'payable'. It only transfers ERC20 tokens, no Ether, so there is no need to have 'payable' here. Recommend removing 'payable' modifier from function `transferToPool`.

albertocuestacanada (Yield) confirmed:

> True, thanks!

[N-09] `UniswapV3Oracle` function `_peek` is public

In contract `UniswapV3Oracle`, the function `_peek` has visibility of public while the name and similar functions in other oracles are declared as private.

Recommend giving `_peek` private visibility.

albertocuestacanada (Yield) confirmed:

> Doh!

[N-10] function `build` could explicitly check that seriesId is not 0

It would be helpful if function `build`, explicitly checked that " `seriesId != bytes12(0)` ".

In practice, it is not possible to have a series with an id of 0, so this check will not pass:

```
require ```(ilks[seriesId][ilkId] == true, "Ilk not added to ser
```

Because the error message is not informative, I am suggesting adding an explicit check and recommend requiring "(seriesId != bytes12(0), "Series id is zero");`

albertocuestacanada (Yield) confirmed and resolved:

> Fix

[N-11] Unnecessary `unchecked` keyword is used in `FYToken`

At line 172 in the contract `FYToken`, the `unchecked` keyword is unnecessary since no arithmetic operation is involved.

Recommend considering removing the `unchecked` keyword.

albertocuestacanada (Yield) confirmed

[N-12] Constants “chi” and “rate”

Several implementations of the value of “chi” and “rate” are used, sometimes as constant, and sometimes the direct value is used. The risk is that if it is changed in one place, it might not be changed in another place, leading to bugs. See issue page for proof of concept and referenced code.

Recommend defining the constants for “chi” and “rate” on one location and include this where required.

albertocuestacanada (Yield) confirmed but disagreed with severity:

> We can create a `Constants.sol` contract for others to inherit to. The downside is code readability, but it is probably an acceptable trade-off.

> However, I can’t think of a scenario in which we would want to change these constants, and as a matter of fact, we can’t because they are included in `Cauldron`, and this contract is not upgradable. I would downgrade the risk to 0.

Gas Optimizations

[G-01] Gas optimizations - using external over public

The following methods could be external instead of public (see issue #60)

albertocuestacanada (Yield) confirmed

[G-02] Inefficient `Witch` buy

In `Witch.buy`, there’s the possibility to do one multiplication instead of two divisions: Instead of computing the `_ink price_` as `1 / artPrice` and then dividing by it to get the ink amount as `ink = art/price`, just keep the `_art price_` and multiply it by the `art` amount. These lines need to be changed:

```
solidity
price = term1.wmul(term2); // this is the art price in terms of
ink = uint256(art).wmulup(price); // can just multiply by art pr
```

One saves gas by doing one multiplication instead of two divisions which seems to be an important goal of Yield v2.

albertocuestacanada (Yield) acknowledged:

> I need to confirm the math is right before I confirm.

[G-03] gas improvements `toAsciiString`

The function `toAsciiString` can be improved the be easier to read and use less gas.

See issue page for proof of concept and proposed improved version.

albertocuestacanada (Yield) confirmed

[G-04] unnecessary store

In the function batch of `Ladle.sol`, at the operation `GIVE`, the value of vault is stored and is deleted directly afterward, rendering storage unnecessary. Maybe the solidity compiler already optimizes this.

Recommend removing the “vault =“.

albertocuestacanada (Yield) confirmed:

> Thanks!

[G-05] `peek` and `get` are identical (non-transactional)

In the contract `ChainlinkMultiOracle`, both functions `peek` and `get` are identical. They are declared as views while based on IOracle interface, `get` should be transactional.

albertocuestacanada (Yield) confirmed:

> We will fix it, but I don't think any risk is derived from this bug.

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