Volt Protocol contest

2022-06-29

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 audit 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 audit contest outlined in this document, C4 conducted an analysis of the Volt Protocol smart contract system written in Solidity. The audit contest took place between March 31—April 6 2022.

Wardens

44 Wardens contributed reports to the Volt Protocol contest:

1. cmichel
2. rayn
3. hyh
4. IllIllI
5. catchup
6. georgypetrov
7. robee
8. Jujic
9. Dravee
10. Hawkeye (0xwags and 0xmint)
11. 0xkatana
12. 0xDjango
13. rfa
14. CertoraInc
15. defsec
16. kenta
17. sahar

18. Funen
19. Meta0xNull
20. TerrierLover
21. Kenshin
22. Sleepy
23. 0xkowloon
24. csanuragjain
25. cccz
26. teryanarmen
27. pauliax
28. kyliek
29. hake
30. Kthere
31. hubble (ksk2345 and shri4net)
32. danb
33. async
34. aysha
35. BouSalman
36. shenwilly
37. berndartmueller
38. saian
39. 0v3rf10w
40. 0xNazgul
41. okkothejawa
42. samruna

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 6 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 36 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 25 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Volt Protocol contest repository, and is composed of 17 smart contracts written in the Solidity programming

language and includes 2,430 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/non-critical.

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 (1)

[H-01] Oracle price does not compound

Submitted by cmichel

ScalingPriceOracle.sol#L136 ScalingPriceOracle.sol#L113

The oracle does not correctly compound the monthly APRs - it resets on `fulfill`. Note that the `oraclePrice` storage variable is only set in `_updateCPIData` as part of the oracle `fulfill` callback. It's set to the old price (price from 1 month ago) plus the interpolation from `startTime` to now. However, `startTime` is **reset** in `requestCPIData` due to the `afterTimeInit` modifier, and therefore when Chainlink calls `fulfill` in response to the CPI request, the `timeDelta = block.timestamp - startTime` is close to zero again and `oraclePrice` is updated to itself again.

This breaks the core functionality of the protocol as the oracle does not track the CPI, it always resets to `1.0` after every `fulfill` instead of compounding it. In addition, there should also be a way for an attacker to profit from the sudden drop of the oracle price to `1.0` again.

Proof of Concept

As an example, assume `oraclePrice = 1.0 (1e18)`, `monthlyAPR = 10%`. The time elapsed is 14 days. Calling `getCurrentOraclePrice()` now would return `1.0 + 14/28 * 10% = 1.05`.

- It's now the 15th of the month and one can trigger `requestCPIData`. **This resets `startTime = now`**.
- Calling `getCurrentOraclePrice()` now would return `1.0` again as `timeDelta` (and `priceDelta`) is zero: `oraclePriceInt + priceDelta = oraclePriceInt = 1.0`.
- When `fulfill` is called it sets `oraclePrice = getCurrentOraclePrice()` which will be close to `1.0` as the `timeDelta` is tiny.

Recommended Mitigation Steps

The `oraclePrice` should be updated in `requestCPIData()` not in `fulfill`. Cover this scenario of multi-month accumulation in tests.

ElliotFriedman (Volt) commented: > Oracle price does compound per this line of code: https://github.com/code-423n4/2022-03-volt/blob/f1210bf3151095e4d371c9e9d7682d9031860bbd/contracts/oracle/ScalingPriceOracle.sol#L197-L198

ElliotFriedman (Volt) confirmed and commented: > This is the only valid critical finding we have seen so far! Great work cmichel!

Medium Risk Findings (6)

[M-01] `vcon` address change not persistent across protocol components

Submitted by rayn

Core.sol#L27 CoreRef.sol#L22 CoreRef.sol#L199

`vcon` address is allowed to be updated by `GOVERNOR` in `Core`, however, this change will not be reflected in `CoreRef._vcon`. Moreover, since `CoreRef._vcon` cannot be updated due to contract design, it is also impossible to fix this manually. We are not yet sure how `vcon` will be used throughout the volt protocol, since details have not yet been made clear and code does not include related implementations. Consequently, it is impossible to estimate the exact impact. However, this desync between contracts seem dangerous enough to raise our attention, hence this report to inform the volt team about it.

Proof of Concept

In `Core`, `vcon` is allowed to be updated by GOVERNORs

```
function setVcon(IERC20 _vcon) external onlyGovernor {
    vcon = _vcon;

    emit VconUpdate(_vcon);
```

```
}
```

But in `CoreRef`, a contract inherited by several other ones including `NonCustodialPSM`, `GlobalRateLimitedMinter`, `ERC20CompountPCVDeposit` and `Volt`, `_vcon` is fixed upon initialization and cannot be further updated

```
IERC20 private immutable _vcon;
...
constructor(address coreAddress) {
    ...
    _vcon = ICore(coreAddress).vcon();
    ...
}
```

Thus if `GOVERNORS` ever updated `vcon` in `Core`, the state between `Core` and all other Volt protocol components will mismatch.

Currently `_vcon` is not used in any place within the Volt protocol, but judging from the description in whitepapaer, future governance will be based on it, thus any potential desync will be devastating.

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

There are several possible solutions.

The first is to dynamically fetch `vcon` from the `Core` whenever `CoreRef` uses it, and avoid storing a static copy locally.

```
function vcon() public view override returns (IERC20) {
    return _volt.vcon();
}
```

The second is to expose a public API to update `_vcon` in `CoreRef`, however, this approach might not be especially favorable since many components will require updates at once, and it is highly possible that future GOVERNORs miss some of them while doing updates.

ElliotFriedman (Volt) disagreed with High severity and commented:
> Agreed this is an issue; however, if the VCON address is updated, contracts that need to reference the new value will need to be redeployed to cache this new address when CoreRef is instantiated.

Jack the Pug (judge) decreased severity to Medium and commented:
> Based on the severity of the impact, I'm downgrading this to `Medium`.

[M-02] Setting new buffer does not reduce current buffer to cap

Submitted by cmichel, also found by rayn and catchup

RateLimited.sol#L142

The `RateLimited.setBufferCap` function first updates the buffer and then sets the new cap, but does not apply the new cap to the updated buffer. Meaning, the updated buffer value can be larger than the new buffer cap which should never be the case. Actions consuming more than the new buffer cap can be performed.

```
function _setBufferCap(uint256 newBufferCap) internal {
    // @audit still uses old buffer cap, should set buffer first
    _updateBufferStored();

    uint256 oldBufferCap = bufferCap;
    bufferCap = newBufferCap;


    emit BufferCapUpdate(oldBufferCap, newBufferCap);
}
```

Recommended Mitigation Steps

Update the buffer after setting the new cap:

```
function _setBufferCap(uint256 newBufferCap) internal {
-   _updateBufferStored();
    uint256 oldBufferCap = bufferCap;
    bufferCap = newBufferCap;

+   _updateBufferStored();

    emit BufferCapUpdate(oldBufferCap, newBufferCap);
}
```

ElliotFriedman (Volt) confirmed

[M-03] Div by 0

Submitted by robee, also found by georgypetrov, cmichel, and IllIllI

Deviation.sol#L23

Division by 0 can lead to accidentally revert, (An example of a similar issue - https://github.com/code-423n4/2021-10-defiprotocol-findings/issues/84)

https://github.com/code-423n4/2022-03-volt/tree/main/contracts/utils/Deviation.sol#L23 a

It's internal function but since it is used in another internal functions that are used in public and neither of them has this protection I thought it can be considered as medium (e.g. isWithinDeviationThreshold).

ElliotFriedman (Volt) confirmed

Jack the Pug (judge) commented: > It's a real issue but just like many other findings, it's unlikely to be triggered in practice.

Jack the Pug (judge) commented: > In the particular context of this project, which most of the findings won't lead to a code update, I'll keep this as a `Med`.

[M-04] `OracleRef` assumes backup oracle uses the same normalizer as main oracle

Submitted by cmichel

OracleRef.sol#L104

The `OracleRef` assumes that the backup oracle uses the same normalizer as the main oracle. This generally isn't the case as it could be a completely different oracle, not even operated by Chainlink.

If the main oracle fails, the backup oracle could be scaled by a wrong amount and return a wrong price which could lead to users being able to mint volt cheap or redeem volt for inflated underlying amounts.

Recommended Mitigation Steps

Should there be two scaling factors, one for each oracle?

ElliotFriedman (Volt) confirmed and commented: > This is a good catch as it exposes some underlying assumptions made about backup oracles; however, we can assume that both oracles will use the same scaling factor and thus we will not need a second value for the backup oracle.

[M-05] Updating rate limit for addresses restores their entire buffer amount

Submitted by cmichel

MultiRateLimited.sol#L280

When the `bufferCap` is updated for an address in `_updateAddress`, the address's allowed buffer (`bufferStored`) is replenished to the entire `bufferCap`.

The address could frontrun the `updateAddress` call and spend their entire buffer, then the buffer is replenished and they can spend their entire buffer a second time.

Recommended Mitigation Steps

Keep the old buffer value, capped by the new `bufferCap`:

```
+ uint256 newBuffer = individualBuffer(rateLimitedAddress);

  rateLimitData.lastBufferUsedTime = block.timestamp.toUint32();
  rateLimitData.bufferCap = _bufferCap;
  rateLimitData.rateLimitPerSecond = _rateLimitPerSecond;
- rateLimitData.bufferStored = _bufferCap;
+ rateLimitData.bufferStored = min(_bufferCap, newBuffer);
```

ElliotFriedman (Volt) confirmed and commented: > Good catch!

[M-06] `NonCustodialPSM` can become insolvent as CPI index rises

Submitted by hyh

NonCustodialPSM.sol#L236-L248

NonCustodialPSM mints and redeems VOLT to a chosen stablecoin at the current market rate minus a fixed fee. It is assumed that the difference to be covered with pcvDeposit funds. That assumption is similar to one used in FEI protocol, but there no rate growth takes place as FEI to USD rate supposed to be stable, while VOLT to USD rate will rise over time.

VOLT market rate is tied to the off-chain published CPI index. The growth of this index can easily surpass the yield of the pcvDeposit used, so its interest cannot be guaranteed to be always greater than CPI index advancement. The contract can end up in the situation when no redeem be possible, i.e. NonCustodialPSM can become insolvent.

For example, let's say the stablecoin is USDC, and now investors are worried about inflation and buy/mint 100m VOLT for 100m USDC. Fast forward 1 year, and investors were generally right, as due to rise of the oil prices happening simultaneously with logistics issues the CPI index growth end up being 30% APR for the year.

Then, inflation fears abated and, say, stocks become stronger, and investors want their funds now to put them there and sell/redeem 100m VOLT expecting 125m USDC in return (for simplicity say 5m USDC goes to mint and redeem fees combined). USDC deposit strategy used in pcvDeposit yielded 10% APR for

the year. The contract cannot redeem all the funds due as it is 125 - 100 * 1.1 = 15m USDC short.

Putting severity to high as the contract serves requests sequentially and the last investors' funds are lost this way, i.e. in the example above all the users, who came in to redeem when contract has 15m USDC in obligations and no funds, will lose their entire deposits.

Proof of Concept

Continuing the example, current low risk USDC deposit rates are circa 2.5 lower than US CPI:

AAVE: https://classic.aave.com/#/markets

Compound: https://compound.finance/markets/USDC

US CPI: https://www.bls.gov/cpi/

NonCustodialPSM.redeem uses current oracle price to determine what amount of stablecoins to be paid for 1 VOLT:

https://github.com/code-423n4/2022-03-volt/blob/main/contracts/peg/NonCustodialPSM.sol#L236

https://github.com/code-423n4/2022-03-volt/blob/main/contracts/peg/NonCustodialPSM.sol#L378-L390

NonCustodialPSM.mint does the same:

https://github.com/code-423n4/2022-03-volt/blob/main/contracts/peg/NonCustodialPSM.sol#L274

https://github.com/code-423n4/2022-03-volt/blob/main/contracts/peg/NonCustodialPSM.sol#L357-L365

For example, FEI protocol use a wide range of pcvDeposits, whose yields vary depending on the underlying strategy:

https://github.com/fei-protocol/fei-protocol-core/blob/develop/protocol-configuration/mainnetAddresses.ts#L164-L568

But there are no PCV deposits whose returns are linked to CPI of any country, so mismatch (basis) risk exists, which has to be addressed.

Recommended Mitigation Steps

Consider providing a way to directly inject funds from a separately held stability fund (possibly shared across all the stablecoins and corresponding pcvDeposits) in addition to pcvDeposit as its strategy alone cannot guarantee the returns needed.

Ideally, the redeem and mint fees collected should go to this stability fund as well, with the possibility to retrieve them when there is a total surplus big enough.

More important, consider limiting the redeem amount to total user's share of the pcvDeposit and its stability fund part, so the deficit be visible and shared between all the users. A user can then choose either to withdraw now, obtaining less than CPI index due to current liquidity situation, or to wait for stability fund to be filled up or for pcvDeposit yield to catch up. This way no user will lose the whole deposit.

ElliotFriedman (Volt) disputed, disagreed with High severity, and commented: > There is a risk of the system not being able to make everyone whole if the PCV grows too rapidly and onchain yields cannot keep up with or outcompete inflation; however, this is not a code issue, so marking as invalid.

Jack the Pug (judge) commented: > This is a valid concern. The warden explained why it's possible to happen and how the current implementation handled it improperly, which can make some of the users suffer an unexpected and unfair loss. > > The recommended fix is reasonable. > > I'll keep this as a `High`.

ElliotFriedman (Volt) commented: > There is a 10m FEI liquidity backstop provided by the Tribe DAO, so the system would need to be live for many years without earning any yield to finally become insolvent.

Jack the Pug (judge) decreased severity to Medium and commented: > With the new information provided by the sponsor, I now agree that the likelihood of the situation described by the warden is low. And therefore, I'm downgrading the issue from High to Medium.

Low Risk and Non-Critical Issues

For this contest, 36 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **rayn** received the top score from the judge.

The following wardens also submitted reports: IllIllI, cmichel, Hawkeye, 0xDjango, georgypetrov, sahar, defsec, rfa, kenta, CertoraInc, cccz, robee, Jujic, Meta0xNull, pauliax, 0xkatana, hake, Dravee, teryanarmen, kyliek, danb, Kenshin, async, shenwilly, Sleepy, catchup, aysha, hubble, TerrierLover, BouSalman, Kthere, Funen, berndartmueller, 0xkowloon, and csanuragjain.

Summary

List of findings: * Roles management * `oraclePrice` will be imprecise * Unnecessary usage of `init()` * Using ecrecover is against best practice * `valid` in `OraclePassThrough/read()` always returns true * Use Address.sendValue()

without import @openzeppelin/contracts/utils/Address.sol * setPublicChainlinkToken only called on chainId 1 and 42 * Unused functions and modifiers

In summary of recommended security practices, complex role management easily creates confusions over the privilege of each role, it's better to simplify the design of roles. Also, it's better to use a verified library like ECDSA, delete unused functions, set unused variables to private, and import contracts correctly for best practice.

[1] Roles management

After reviewing the entire volt protocol, we found that role management has been made unnecessarily complex. While there are no immediate fatal flaws in the current role assignment, from our prior experience in dealing with privilege management, we worry that such a complex system will likely lead to future problems, especially when management gradually moves from the hands of few reliable developers to an open vcon based governance. Due to this, we feel that there is a need to express our concerns as well as highlight a few role assignments that we find "strange".

First. let's review the roles included in Permissions and Tribes. The admin/members shown below are the ones explicitly assigned in contracts.

```
DEFAULT_ADMIN_ROLE                              (native)
    admin DEFAULT_ADMIN_ROLE
    members

MINTER_ROLE/MINTER          "MINTER_ROLE"           (native) (tribe-major)
    admin GOVERN_ROLE
    members

BURNER_ROLE             "BURNER_ROLE"           (native)            unused
    admin GOVERN_ROLE
    members

PCV_CONTROLLER_ROLE/PCV_CONTROLLER  "PCV_CONTROLLER_ROLE"       (native) (tribe-major)
    admin GOVERN_ROLE
    members

GOVERN_ROLE/GOVERNOR            "GOVERN_ROLE"           (native) (tribe-major)
    admin GOVERN_ROLE
    members
        core
        init() caller

GUARDIAN_ROLE/GUARDIAN          "GUARDIAN_ROLE"         (native) (tribe-major)
    admin GOVERN_ROLE
```

```
    members

PARAMETER_ADMIN             "PARAMETER_ADMIN"       (tribe-admin)
    admin DEFAULT_ADMIN_ROLE
    members

ORACLE_ADMIN                "ORACLE_ADMIN_ROLE"     (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

TRIBAL_CHIEF_ADMIN          "TRIBAL_CHIEF_ADMIN_ROLE"   (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

PCV_GUARDIAN_ADMIN          "PCV_GUARDIAN_ADMIN_ROLE"   (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

MINOR_ROLE_ADMIN            "MINOR_ROLE_ADMIN"      (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

FUSE_ADMIN              "FUSE_ADMIN"            (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

VETO_ADMIN              "VETO_ADMIN"            (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

MINTER_ADMIN                "MINTER_ADMIN"          (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

OPTIMISTIC_ADMIN            "OPTIMISTIC_ADMIN"      (tribe-admin)           unused
    admin DEFAULT_ADMIN_ROLE
    members

LBP_SWAP_ROLE               "SWAP_ADMIN_ROLE"       (tribe-minor)           unused
    admin DEFAULT_ADMIN_ROLE
    members

VOTIUM_ROLE             "VOTIUM_ADMIN_ROLE"     (tribe-minor)           unused
    admin DEFAULT_ADMIN_ROLE
    members
```

```
MINOR_PARAM_ROLE            "MINOR_PARAM_ROLE"      (tribe-minor)           unused
    admin DEFAULT_ADMIN_ROLE
    members

ADD_MINTER_ROLE             "ADD_MINTER_ROLE"       (tribe-minor)
    admin DEFAULT_ADMIN_ROLE
    members

PSM_ADMIN_ROLE              "PSM_ADMIN_ROLE"        (tribe-minor)
    admin DEFAULT_ADMIN_ROLE
    members
```

Notice how a several of those roles are not used in any of the contracts (marked as unused). This is the first problem. While it is understandable that the protocol is incomplete yet, introducing redundant roles does not make management easier. AccessControl.sol allows introducing new roles post-deployment, so it might be a better idea to keep a list of dynamically introduced roles instead of listing a lot of unused ones upfront, especially since there are roles with similar names (PCV_GUARDIAN_ADMIN and PCV_CONTROLLER), between which the difference is not made clear.

Next, we carry on to see the design of `Permissions`. We note that this contract is modified from the implementation of fei protocol, but on the other hand, disagree that the implementation is anywhere near optimal. To support our argument, we discuss the logic of role granting and revoking below. So let's look at the `grantMinter` function.

```
    function grantMinter(address minter) external override onlyGovernor {
        grantRole(MINTER_ROLE, minter);
```

The function specifies that it is a `onlyGovernor` function, which should expectedly mean that `GOVERNOR`, and only `GOVERNOR` have the privilege to add `MINTER_ROLE` members. However, if we dig a bit deeper, it is easy to see that this is not the case.

The `grantRole` function defined in AccessControl.sol is as below. Notice that it also has a modifier that specifies only admins of the specific role has privilege to add new members. Additionally, this function is public.

```
    function grantRole(bytes32 role, address account) public virtual override onlyRole(getRo
        _grantRole(role, account);
    }
```

Combining this logic with the `grantMinter` one above, we can see that the workflow essentially becomes

1. caller is allowed to add new users if it only has admin of `MINTER_ROLE` (call `grantRole` directly)
2. caller is allowed to add new users if it has admin of `MINTER_ROLE` and `GOVERNOR` role (call `grantRole` or `grantMinter`)

3. caller is not allowed to add new users if it only has `GOVERNOR` role (blocked by `grantRole`)

It is clear that the `grantMinter` function now becomes semi-useless, since callers with only `GOVERNOR` role cannot do anything without admin of `MINTER_ROLE`, and admin of `MINTER_ROLE` can always call `grantRole` directly even if it does not have `GOVERNOR` role.

Our best guess of the original intention is that `GOVERNOR` has full privilege over role management, while admin of roles are neglected. To realize this concept, it might be better to override the `grantRole` function and make it non-public, so that callers can't circumvent the `GOVERNOR` check. Finally, change `grantMinter` to use the internal function `_grantRole`. Similar modifications should also be done to the `revokeXXX` series of functions.

Now we've gone through `Permissions`, time to look at `CoreRef` and `OracleRef`. One of the more interesting design choice in `CoreRef` is the introduction of `CONTRACT_ADMIN_ROLE`. This allows an additional role to be granted admin over the specific contract. However, throughout the volt protocol, `CONTRACT_ADMIN_ROLE` does not serve any particularly useful purpose. Moreover, in some places, the usage of `CONTRACT_ADMIN_ROLE` does not make much sense.

For instance, let's look at implementation of `RateLimited` and `MultiRateLimited`. `RateLimited` defines a global limit to the `bufferCap` and `rateLimitPerSecond`, and `MultiRateLimited` defines the upper limit of `individualMaxBufferCap` and `individualMaxRateLimitPerSecond`. In our opinion, for the system to make sense, a user granted permission to change `bufferCap` and `rateLimitPerSecond` should also have permission to change `individualMaxBufferCap` and `individualMaxRateLimitPerSecond`. However, it can be seen that `CONTRACT_ADMIN_ROLES` are allowed to change the global limits through `onlyGovernorOrAdmin` modifier, while individual limits can only be changed by `GOVERNOR` since `onlyGovernor` is used. The flexibility introduced through `CONTRACT_ADMIN_ROLE` is not properly utilized in volt protocol, and as we see in the example above, leads to potential role privilege confusions, thus we deemed it more appropriate to remove the `CONTRACT_ADMIN_ROLE` mechanism altogether for simplicity.

The next aspect we would like to discuss is more of a design choice, and not really a management problem. Take it with a grain of salt. Throughout the contract, there are several places that use the `onlyGovernor` modifier. However given that the roles already included controllers/admins for each specific component (`PCV_CONTROLLER_ROLE`, `PSM_ADMIN_ROLE`), it is probably more appropriate to limit `GOVERNOR` to only manage the `Core` contract. If a governor needs to modify stuff from other contracts, add the corresponding admin role to itself and use that role to authenticate further actions. This design can create a clean cut between metadata management, and actually protocol management, at the cost of slightly more gas spent in granting roles. From our limited experience, this kind of management is more robust and greatly lowers the probability of

mis-management in the future.

Now we've discussed all general suggestions we have for role management, we finally note a few role modifier usage that we find "strange", but are uncertain whether intended or not.

1. `NonCustodialPSM.withdrawERC20` uses modifier `onlyPCVController`. We find it strange that a function in the PSM module requires PCVController role. Shouldn't this require `PSM_ADMIN_ROLE` instead?
2. `CompoundPCVDepositBase.withdraw` uses the modifier `onlyPCVController`. This would require `NonCustodialPSM` to have the `PCV_CONTROLLER_ROLE` to call `withdraw`, and while it is not a problem strictly speaking, we find it strange to grant such a role. A more usual implementation would be to have `CompoundPCVDepositBase` do internal bookkeeping on the amount each address deposited, and allow withdraw with respect to those values (similar to implementation of ERC20). This avoids the introduction of an additional role (a.k.a potential point of failure due to mis-management).

Overall, complexity in role management easily creates confusions over the privilege of each role, and in the specific case of volt protocol, does not really introduce any benefits. We thus urge the developers to re-think the current role management system, and preferably simplify the design.

[02] `oraclePrice` will be imprecise

Public `oraclePrice` variable will be imprecise and confused. If other contracts try to get oracle price by calling `oraclePrice()` rather than calling `getCurrentOraclePrice()`, it will get wrong price.

Proof of Concept

ScalingPriceOracle.sol#L36

Recommended Mitigation Steps

Declare `oraclePrice` to private, and only use `getCurrentOraclePrice()` to get the price instead.

[03] Unnecessary usage of `init()`

Since `Core` contract does not go through a proxy, and the caller of `init()` is the same as the deployer in deploy script, it is unnecessary to use init functions.

Proof of Concept

Core.sol#L20

Recommended Mitigation Steps

Consider putting `init()` logic into `constructor` instead and stop inheriting Initializable.

[04] Using ecrecover is against best practice

Using ecrecover is against best practice. Preferably use ECDSA.recover instead. EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make the signature unique. However it should be impossible to be a threat by now.

Proof of Concept

Volt.sol#L89

Recommended Mitigation Steps

Take these implementations into consideration

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/4a9cc8b4918ef3736229a5cc5a310bdc17bf759f/contracts/utils/cryptography/draft-EIP712.sol

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/draft-ERC20Permit.sol

[05] `valid` in `OraclePassThrough/read()` always returns true

In `OraclePassThrough.sol`, `valid` is always true in `read()`. Since `OracleRef` uses `valid` to determine validity. There should be a invalid case. Otherwise, the validity check is meaningless.

Proof of Concept

OraclePassThrough.sol#L41

OracleRef.sol#L102

Recommended Mitigation Steps

Define the invalid case.

[06] Use Address.sendValue() without import @openzeppelin/contracts/utils/Address.sol

Although `@openzeppelin/contracts/proxy/utils/Initializable.sol` is imported in `Core.sol`. A good coding practice should import `@openzeppelin/contracts/utils/Address.sol` when using it.

Proof of Concept

PCVDeposit.sol#L43

Recommended Mitigation Steps

Import `@openzeppelin/contracts/utils/Address.sol` in `PCVDeposit.sol`.

[07] setPublicChainlinkToken only called on chainId 1 and 42

`ScalingPriceOracle` checks `chainId` in contract constructor, it might be better to upfront reject construction if it is not intended to serve other chains.

Proof of Concept

ScalingPriceOracle.sol#L84-L86

Recommended Mitigation Steps

It's better to upfront check and reject construction if it is not intended to serve other chains.

[08] Unused functions and modifiers

These functions are noted as for backward compatibility, but not used anywhere, should probably be removed.

Proof of Concept

CoreRef.sol#L10

Unused functions:

- _burnVoltHeld

Unused modifiers:

- onlyVolt
- hasAnyOfFiveRoles
- hasAnyOfFourRoles
- hasAnyOfThreeRoles
- onlyGovernorOrGuardianOrAdmin
- ifMinterSelf

Recommended Mitigation Steps

Remove these unused functions and modifiers.

Gas Optimizations

For this contest, 25 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **IllIllI** received the top score from the judge.

The following wardens also submitted reports: Jujic, Dravee, 0xkatana, CertoraInc, robee, rfa, catchup, Funen, defsec, georgypetrov, saian, kenta, Kenshin, TerrierLover, Hawkeye, samruna, 0v3rf10w, 0xNazgul, rayn, Sleepy, okkothejawa, 0xkowloon, csanuragjain, and Meta0xNull.

[G-01] Store the timestamp endpoint rather than re-calculating it every time

The `isTimeEnded()` function does a lot of calculations every time it's called. It's called from multiple modifiers so it's important for it to be efficient. Rather than storing the duration, the code can calculate and store the ending timestamp, so `isTimeEnded()` can just be a direct comparison of two `uint256`s. The duration can be calculated by subtracting the start timestamp from the ending timestamp.

1. File: contracts/utils/Timed.sol (lines 38-59)

```
/// @notice return true if time period has ended
function isTimeEnded() public view returns (bool) {
    return remainingTime() == 0;
}

/// @notice number of seconds remaining until time is up
/// @return remaining
function remainingTime() public view returns (uint256) {
    return duration - timeSinceStart(); // duration always >= timeSinceStart which is or
}

/// @notice number of seconds since contract was initialized
/// @return timestamp
/// @dev will be less than or equal to duration
function timeSinceStart() public view returns (uint256) {
    if (!isTimeStarted()) {
        return 0; // uninitialized
    }
    uint256 _duration = duration;
    uint256 timePassed = block.timestamp - startTime; // block timestamp always >= start
    return timePassed > _duration ? _duration : timePassed;
}
```

[G-02] Lots of duplicated code between `RateLimited.sol` and `MultiRateLimited.sol`

The functionality of `RateLimited.sol` can be achieved by using either `address(0)` or `address(this)` as the `rateLimitedAddress` so having a separate `RateLimited.sol` contract is a waste of deployment gas.

1. File: contracts/utils/RateLimited.sol (Various lines throughout the file)
2. File: contracts/utils/MultiRateLimited.sol (Various lines throughout the file)

[G-03] `require()/revert()` strings longer than 32 bytes cost extra gas

See original submission for instances.

[G-04] Use a more recent version of solidity

Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

1. File: contracts/pcv/PCVDeposit.sol (line 2)

```
pragma solidity ^0.8.4;
```

2. File: contracts/oracle/OraclePassThrough.sol (line 2)

```
pragma solidity ^0.8.4;
```

3. File: contracts/oracle/ScalingPriceOracle.sol (line 2)

```
pragma solidity ^0.8.4;
```

4. File: contracts/utils/Timed.sol (line 2)

```
pragma solidity ^0.8.4;
```

5. File: contracts/utils/Deviation.sol (line 2)

```
pragma solidity ^0.8.4;
```

6. File: contracts/utils/GlobalRateLimitedMinter.sol (line 2)

```
pragma solidity ^0.8.4;
```

7. File: contracts/utils/MultiRateLimited.sol (line 2)

```
pragma solidity ^0.8.4;
```

8. File: contracts/utils/RateLimited.sol (line 2)

```
pragma solidity ^0.8.4;
```

9. File: contracts/volt/Volt.sol (line 2)

```
pragma solidity ^0.8.4;
```

10. File: contracts/refs/OracleRef.sol (line 2)

```
pragma solidity ^0.8.4;
```

11. File: contracts/refs/CoreRef.sol (line 2)

```
pragma solidity ^0.8.4;
```

12. File: contracts/core/Core.sol (line 2)

```
pragma solidity ^0.8.4;
```

13. File: contracts/core/TribeRoles.sol (line 2)

```
pragma solidity ^0.8.4;
```

14. File: contracts/core/Permissions.sol (line 2)

```
pragma solidity ^0.8.4;
```

15. File: contracts/peg/NonCustodialPSM.sol (line 2)

```
pragma solidity ^0.8.4;
```

[G-05] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

1. File: contracts/pcv/compound/CompoundPCVDepositBase.sol (line 2)

```
pragma solidity ^0.8.0;
```

2. File: contracts/pcv/compound/ERC20CompoundPCVDeposit.sol (line 2)

```
pragma solidity ^0.8.0;
```

[G-06] Using `bool`s for storage incurs overhead

```
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

1. File: contracts/utils/RateLimited.sol (line 23)

```
bool public doPartialAction;
```

2. File: contracts/refs/OracleRef.sol (line 25)

```
bool public override doInvert;
```

3. File: contracts/peg/NonCustodialPSM.sol (line 52)

```
bool public redeemPaused;
```

4. File: contracts/peg/NonCustodialPSM.sol (line 55)

```
bool public mintPaused;
```

[G-07] `internal` functions only called once can be inlined to save gas

1. File: contracts/oracle/ScalingPriceOracle.sol (line 171)

```
function _updateCPIData(uint256 _cpiData) internal {
```

2. File: contracts/oracle/ScalingPriceOracle.sol (line 218)

```
function _addNewMonth(uint128 newMonth) internal {
```

3. File: contracts/core/Permissions.sol (line 211)

```
function _setupGovernor(address governor) internal {
```

[G-08] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second access of a state variable within a function. Less obvious optimizations include having local storage variables of mappings within state variable mappings or mappings within state variable structs, having local storage variables of structs within mappings, or having local caches of state variable contracts/addresses.

1. File: contracts/pcv/compound/CompoundPCVDepositBase.sol (line 32)

```
require(cToken.isCToken(), "CompoundPCVDeposit: Not a cToken");
```

2. File: contracts/pcv/compound/CompoundPCVDepositBase.sol (line 57)

```
(cToken.balanceOf(address(this)) * exchangeRate) /
```

3. File: contracts/pcv/compound/ERC20CompoundPCVDeposit.sol (line 31)

```
token.approve(address(cToken), amount);
```

4. File: contracts/oracle/ScalingPriceOracle.sol (line 124)

```
percentageChange = (delta * Constants.BP_INT) / int128(previousMonth);
```

5. File: contracts/utils/MultiRateLimited.sol (line 344)

```
rateLimitPerAddress[rateLimitedAddress].lastBufferUsedTime = block
```

6. File: contracts/utils/RateLimited.sol (line 110)

```
emit BufferUsed(usedAmount, bufferStored);
```

7. File: contracts/refs/OracleRef.sol (line 104)

```
    (_peg, valid) = backupOracle.read();
```

8. File: contracts/refs/OracleRef.sol (line 111)

```
    scalingFactor = 10**(-1 * decimalsNormalizer).toUint256();
```

[G-09] Splitting `require()` statements that use `&&` saves gas

See this issue for an example

1. File: contracts/volt/Volt.sol (lines 90-93)

```
require(
    recoveredAddress != address(0) && recoveredAddress == owner,
    "Fei: INVALID_SIGNATURE"
);
```

[G-10] Usage of `uints/ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

See original submission for instances.

[G-11] Expressions for constant values such as a call to `keccak256()`, should use `immutable` rather than `constant`

See this issue for a detail description of the issue.

See original submission for instances.

[G-12] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code

1. File: contracts/oracle/ScalingPriceOracle.sol (line 50)

```
uint256 public constant override TIMEFRAME = 28 days;
```

2. File: contracts/oracle/ScalingPriceOracle.sol (line 55)

```
uint256 public constant override MAXORACLEDEVIATION = 2_000;
```

3. File: contracts/oracle/ScalingPriceOracle.sol (line 61)

```
bytes32 public immutable jobId;
```

4. File: contracts/oracle/ScalingPriceOracle.sol (line 64)

```
uint256 public immutable fee;
```

5. File: contracts/utils/RateLimited.sol (line 11)

```
uint256 public immutable MAX_RATE_LIMIT_PER_SECOND;
```

6. File: contracts/volt/Volt.sol (lines 13-14)

```
bytes32 public constant PERMIT_TYPEHASH =
    0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;
```

7. File: contracts/core/Permissions.sol (line 10)

```
bytes32 public constant override BURNER_ROLE = keccak256("BURNER_ROLE");
```

8. File: contracts/core/Permissions.sol (line 11)

```
bytes32 public constant override MINTER_ROLE = keccak256("MINTER_ROLE");
```

9. File: contracts/core/Permissions.sol (lines 12-13)

```
bytes32 public constant override PCV_CONTROLLER_ROLE =
    keccak256("PCV_CONTROLLER_ROLE");
```

10. File: contracts/core/Permissions.sol (line 14)

```
bytes32 public constant override GOVERN_ROLE = keccak256("GOVERN_ROLE");
```

11. File: contracts/core/Permissions.sol (line 15)

```
bytes32 public constant override GUARDIAN_ROLE = keccak256("GUARDIAN_ROLE");
```

12. File: contracts/peg/NonCustodialPSM.sol (line 49)

```
uint256 public immutable override MAX_FEE = 300;
```

[G-13] Duplicated `require()/revert()` checks should be refactored to a modifier or function

1. File: contracts/utils/MultiRateLimited.sol (lines 305-308)

```
require(
    _rateLimitPerSecond <= MAX_RATE_LIMIT_PER_SECOND,
    "MultiRateLimited: rateLimitPerSecond too high"
);
```

[G-14] State variables only set in the constructor should be declared `immutable`

1. File: contracts/pcv/compound/ERC20CompoundPCVDeposit.sol (line 16)

```
IERC20 public token;
```

[G-15] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables

1. File: contracts/utils/MultiRateLimited.sol (lines 270-273)

```
require(
    _rateLimitPerSecond <= MAX_RATE_LIMIT_PER_SECOND,
    "MultiRateLimited: rateLimitPerSecond too high"
);
```

2. File: contracts/utils/MultiRateLimited.sol (lines 305-308)

```
require(
    _rateLimitPerSecond <= MAX_RATE_LIMIT_PER_SECOND,
    "MultiRateLimited: rateLimitPerSecond too high"
);
```

3. File: contracts/utils/RateLimited.sol (lines 46-49)

```
require(
    _rateLimitPerSecond <= _maxRateLimitPerSecond,
    "RateLimited: rateLimitPerSecond too high"
);
```

[G-16] Use custom errors rather than `revert()/require()` strings to save deployment gas

1. File: contracts/oracle/ScalingPriceOracle.sol (Various lines throughout the file)

2. File: contracts/utils/Timed.sol (Various lines throughout the file)
3. File: contracts/utils/MultiRateLimited.sol (Various lines throughout the file)
4. File: contracts/utils/RateLimited.sol (Various lines throughout the file)
5. File: contracts/volt/Volt.sol (Various lines throughout the file)
6. File: contracts/refs/OracleRef.sol (Various lines throughout the file)
7. File: contracts/refs/CoreRef.sol (Various lines throughout the file)
8. File: contracts/core/Permissions.sol (Various lines throughout the file)
9. File: contracts/peg/NonCustodialPSM.sol (Various lines throughout the file)

[G-17] Not using the named return variables when a function returns, wastes deployment gas

1. File: contracts/oracle/ScalingPriceOracle.sol (line 150)

```
return sendChainlinkRequestTo(oracle, request, fee);
```

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.