Centrifuge
Findings & Analysis Report

2023-10-11

Table of contents

- Disclosures

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 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 outlined in this document, C4 conducted an analysis of the Centrifuge smart contract system written in Solidity. The audit took place between September 8—September 14 2023.

Wardens

85 Wardens contributed reports to the Centrifuge:

1. ciphermarco
2. alexfilippov314
3. 0x3b
4. jaraxxus
5. twicek
6. J4X
7. 0xStalin
8. Aymen0909
9. bin2chen
10. imtybik
11. castle_chain
12. HChang26
13. merlin

14. maanas
15. nobody2018
16. 0xnev
17. mert_eren
18. Bauchibred
19. josephdara
20. BARW (BenRai and albertwh1te)
21. ast3ros
22. Ch_301
23. degensec
24. 0xpiken
25. ravikiranweb3
26. lsaudit
27. catellatech
28. Sathish9098
29. cats
30. kaveyjoe
31. 0xbrett8571
32. 0xfuje
33. rvierdiiev
34. 0xmystery
35. Kow
36. 0xRobsol
37. codegpt
38. 0xkazim
39. nmirchev8
40. gumgumzum
41. T1MOH
42. Kral01

43. pipidu83
44. Vagner
45. KrisApostolov
46. OxTiwa
47. Oxgrbr
48. PENGUN
49. bitsurfer
50. ether_sky
51. Oxklh
52. Phantasmagoria
53. ladboy233
54. deth
55. IceBear
56. OxPacelli
57. MaslarovK
58. grearlake
59. rokinot
60. fouzantanveer
61. K42
62. hals
63. emerald7017
64. foxb868
65. OxLook
66. MohammedRizwan
67. jkoppel
68. jolah1
69. alexzoid
70. sandy
71. btk

72. Kaysoft
73. klau5
74. 0xAadi
75. Bughunter101
76. 0xblackskull
77. SanketKogekar
78. m_Rassska
79. 7ashraf
80. JP_Courses
81. Krace
82. mrudenko
83. fatherOfBlocks
84. 0xHelium

This audit was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 39 reports detailing issues with a risk rating of LOW severity or non-critical.

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

Scope

The code under review can be found within the C4 Centrifuge audit repository, and is composed of 18 smart contracts written in the Solidity programming language and includes 2,657 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **IllIIII-bot** from warden IllIIII, generated the [Automated Findings report]() and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

Medium Risk Findings (8)

[M-01] `onlyCentrifugeChainOrigin()` can't require `msg.sender` equal `axelarGateway`

Submitted by [bin2chen](), *also found by* [maanas](), [nobody2018](), [castle_chain](), [mert_eren](), *and* [merlin]()

In `AxelarRouter.sol`, we need to ensure the legitimacy of the `execute()` method execution, mainly through two methods:

1. `axelarGateway.validateContractCall ()` to validate if the `command` is approved or not.

2. `onlyCentrifugeChainOrigin()` is used to validate that `sourceChain` `sourceAddress` is legal.

Let’s look at the implementation of `onlyCentrifugeChainOrigin()` :

```
        modifier onlyCentrifugeChainOrigin(string calldata sourceCha
    @>      require(msg.sender == address(axelarGateway), "AxelarRou
            require(
                keccak256(bytes(axelarCentrifugeChainId)) == keccak2
                "AxelarRouter/invalid-source-chain"
            );
            require(
                keccak256(bytes(axelarCentrifugeChainAddress)) == ke
                "AxelarRouter/invalid-source-address"
            );
            _;
        }
```

The problem is that this restriction `msg.sender == address(axelarGateway)` .

When we look at the official `axelarGateway.sol` contract, it doesn’t provide any call external contract ‘s `execute()` method.

So `msg.sender` cannot be `axelarGateway` , and the official example does not restrict `msg.sender` .

The security of the command can be guaranteed by `axelarGateway.validateContractCall()` , `sourceChain` , `sourceAddress` .

There is no need to restrict `msg.sender` .

`axelarGateway` code address
https://github.com/axelarnetwork/axelar-cgp-solidity/blob/main/contracts/AxelarGateway.sol

Can’t find anything that calls `router.execute()` .

Impact

`router.execute()` cannot be executed properly, resulting in commands from other chains not being executed， protocol not working properly.

Recommended Mitigation

Remove `msg.sender` restriction

```
        modifier onlyCentrifugeChainOrigin(string calldata sourceCha
-           require(msg.sender == address(axelarGateway), "AxelarRou
            require(
                keccak256(bytes(axelarCentrifugeChainId)) == keccak2
                "AxelarRouter/invalid-source-chain"
            );
            require(
                keccak256(bytes(axelarCentrifugeChainAddress)) == ke
                "AxelarRouter/invalid-source-address"
            );
            _;
        }
```

Assessed type

Context

hieronx (Centrifuge) confirmed

gzeon (judge) increased severity to High and commented:

> This seems High risk to me since the Axelar bridge is a centerpiece of this protocol, and when deployed in a certain way where the AxelarRouter is the only ward, it might cause user deposits to be stuck forever.

gzeon (judge) decreased severity to Medium and commented:

> Reconsidering severity to Medium here since the expected setup would have DelayedAdmin able to unstuck the system.

hieronx (Centrifuge) commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/168

[M-02] `LiquidityPool::requestRedeemWithPermit` transaction can be front run with the different liquidity pool

Submitted by alexfilippov314

The permit signature is linked only to the tranche token. That's why it can be used with any liquidity pool with the same tranche token. Since anyone can call `LiquidityPool::requestRedeemWithPermit` the following scenario is possible:

1. Let's assume that some user has some amount of tranche tokens. Let's also assume that there are multiple liquidity pools with the same tranche token. For example, USDX pool and USDY pool.
2. The user wants to redeem USDX from the USDX pool using `requestRedeemWithPermit`. The user signs the permit and sends a transaction.
3. A malicious actor can see this transaction in the mempool and use the signature from it to request a redemption from the USDY pool with a greater fee amount.
4. Since this transaction has a greater fee amount it will likely be executed before the valid transaction.
5. The user's transaction will be reverted since the permit has already been used.
6. If the user will not cancel this malicious request until the end of the epoch this request will be executed, and the user will be forced to claim USDY instead of USDX.

This scenario assumes some user's negligence and usually doesn't lead to a significant loss. But in some cases (for example, USDY depeg) a user can end up losing significantly.

Proof of Concept

The test below illustrates the scenario described above:

```
    function testPOCIssue1(
        uint64 poolId,
        string memory tokenName,
        string memory tokenSymbol,
        bytes16 trancheId,
        uint128 currencyId,
        uint256 amount
    ) public {
        vm.assume(currencyId > 0);
        vm.assume(amount < MAX_UINT128);
        vm.assume(amount > 1);

        // Use a wallet with a known private key so we can sign the
        address investor = vm.addr(0xABCD);
        vm.prank(vm.addr(0xABCD));

        LiquidityPool lPool =
            LiquidityPool(deployLiquidityPool(poolId, erc20.decimals
        erc20.mint(investor, amount);
        homePools.updateMember(poolId, trancheId, investor, type(uin

        // Sign permit for depositing investment currency
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            0xABCD,
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    erc20.DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            erc20.PERMIT_TYPEHASH(), investor, addre
                        )
                    )
                )
            )
        );

        lPool.requestDepositWithPermit(amount, investor, block.times
        // To avoid stack too deep errors
        delete v;
        delete r;
        delete s;

        // ensure funds are locked in escrow
        assertEq(erc20.balanceOf(address(escrow)), amount);
        assertEq(erc20.balanceOf(investor), 0);

        // collect 50% of the tranche tokens
        homePools.isExecutedCollectInvest(
            poolId,
            trancheId,
            bytes32(bytes20(investor)),
            poolManager.currencyAddressToId(address(erc20)),
            uint128(amount),
            uint128(amount)
        );

        uint256 maxMint = lPool.maxMint(investor);
        lPool.mint(maxMint, investor);

        {
            TrancheToken trancheToken = TrancheToken(address(lPool.s
            assertEq(trancheToken.balanceOf(address(investor)), maxM

            // Sign permit for redeeming tranche tokens
            (v, r, s) = vm.sign(
                0xABCD,
                keccak256(
                    abi.encodePacked(
                        "\x19\x01",
                        trancheToken.DOMAIN_SEPARATOR(),
                        keccak256(
                            abi.encode(
                                trancheToken.PERMIT_TYPEHASH(),
                                investor,
                                address(investmentManager),
                                maxMint,
                                0,
                                block.timestamp
                            )
                        )
                    )
                )
            );
        }

        // Let's assume that there is another liquidity pool with th
        // but a different currency
        LiquidityPool newLPool;
        {
            assert(currencyId != 123);
            address newErc20 = address(_newErc20("Y's Dollar", "USDY
            homePools.addCurrency(123, newErc20);
            homePools.allowPoolCurrency(poolId, 123);
            newLPool = LiquidityPool(poolManager.deployLiquidityPool
        }
        assert(address(lPool) != address(newLPool));
        // Malicious actor can use the signature extracted from the
        // request redemption from the different liquidity pool
        vm.prank(makeAddr("malicious"));
        newLPool.requestRedeemWithPermit(maxMint, investor, block.ti

        // User's transaction will fail since the signature has alre
        vm.expectRevert();
        lPool.requestRedeemWithPermit(maxMint, investor, block.times
    }
```

Recommended Mitigation Steps

One of the ways to mitigate this issue is to add some identifier of the liquidity pool to the permit message. This way permit will be linked to a specific liquidity pool.

hieronx (Centrifuge) confirmed and commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/159

[M-03] Cached `DOMAIN_SEPARATOR` is incorrect for tranche tokens potentially breaking permit integrations

Submitted by Kow, also found by 0xRobsol, codegpt, 0xkazim, josephdara, Aymen0909, 0xpiken (1, 2), bin2chen, nmirchev8, rvierdiiev (1, 2), lsaudit (1, 2), 0xfuje, gumgumzum, T1MOH, and ravikiranweb3 (1, 2, 3)

Attempts to interact with tranche tokens via `permit` may always revert.

Proof of Concept

When new tranche tokens are deployed, the initial `DOMAIN_SEPARATOR` is calculated and cached in the constructor.
https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/token/ERC20.sol#L42-L49

```
        constructor(uint8 decimals_) {
```

```
        ...
        deploymentChainId = block.chainid;
        _DOMAIN_SEPARATOR = _calculateDomainSeparator(block.chai
    }
```

This uses an empty string since `name` is only set after deployment.
https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/util/Factory.sol#L81-L109

```
    function newTrancheToken(
        uint64 poolId,
        bytes16 trancheId,
        string memory name,
        string memory symbol,
        uint8 decimals,
        address[] calldata trancheTokenWards,
        address[] calldata restrictionManagerWards
    ) public auth returns (address) {
        ...
        TrancheToken token = new TrancheToken{salt: salt}(decima

        token.file("name", name);
        ...
    }
```

Consequently, the domain separator is incorrect (when `block.chainid == deploymentChainId` where the domain separator is not recalculated) and will cause reverts when signatures for `permit` are attempted to be constructed using the tranche token's `name` (which will not be empty).

It should also be noted that the tranche token `name` could be changed by a call to `updateTranchTokenMetadata` which may also introduce complications with the domain separator.

Recommended Mitigation Steps

Consider setting the name in the constructor before the cached domain separator is calculated.

hieronx (Centrifuge) confirmed and commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/142

[M-04] You can deposit really small amount for other users to DoS them

Submitted by 0x3b, *also found by* jaraxxus *and* twicek

Deposit and mint under LiquidityPool lack access control, which enables any user to **proceed** the mint/deposit for another user. Attacker can deposit (this does not require tokens) some wai before users TX to DoS the deposit.

Proof of Concept

deposit and mint do processDeposit/processMint which are the secondary functions to the requests. These function do not take any value in the form of tokens, but only send shares to the receivers. This means they can be called for free.

With this an attacker who wants to DoS a user, can wait him to make the request to deposit and on the next epoch front run him by calling deposit with something small like 1 wei. Afterwards when the user calls `deposit`, his TX will inevitable revert, as he will not have enough balance for the full deposit.

Recommended Mitigation Steps

Have some access control modifiers like withApproval used also in redeem.

```
-    function deposit(uint256 assets, address receiver) public r
+    function deposit(uint256 assets, address receiver) public r
         shares = investmentManager.processDeposit(receiver, asse
         emit Deposit(address(this), receiver, assets, shares);
     }

-    function mint(uint256 shares, address receiver) public retu
+    function mint(uint256 shares, address receiver) public retu
         assets = investmentManager.processMint(receiver, shares)
         emit Deposit(address(this), receiver, assets, shares);
     }
```

Assessed type

Access Control

hieronx (Centrifuge) confirmed and commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/136

[M-05] Investors claiming their `maxDeposit` by using the `LiquidityPool.deposit()` will cause other users to be unable to claim their `maxDeposit` / `maxMint`

Submitted by 0xStalin, also found by Aymen0909, imtybik, HChang26, and J4X

Claiming deposits using the `LiquidityPool.deposit()` will cause the Escrow contract to not have enough shares to allow other investors to claim their maxDeposit or maxMint values for their deposited assets.

Proof of Concept

- Before an investor can claim their deposits, they first needs to request the deposit and wait for the Centrigue Chain to validate it in the next epoch.
- Investors can request deposits at different epochs without the need to claim all the approved deposits before requesting a new deposit, in the end, the maxDeposit and maxMint values that the investor can claim will be increased accordingly based on all the request deposits that the investor makes.
- When the requestDeposit of the investor is processed in the Centrifuge chain, a number of TrancheShares will be minted based on the price at the moment when the request was processed and the total amount of deposited assets, this TrancheShares will be deposited to the Escrow contract, and the TrancheShares will be waiting for the investors to claim their deposits.
- When investors decide to claim their deposit they can use the `LiquidityPool.deposit()` function, this function receives as arguments the number of assets that are being claimed and the address of the account to claim the deposits for.

```
function deposit(uint256 assets, address receiver) public returr
    shares = investmentManager.processDeposit(receiver, assets);
    emit Deposit(address(this), receiver, assets, shares);
}
```

- The `LiquidityPool.deposit()` function calls the `InvestmentManager::processDeposit()` which will validate that the amount of assets being claimed doesn't exceed the investor's deposit limits, will compute the deposit price in the `InvestmentManager::calculateDepositPrice()` , which basically computes an average price for all the request deposits that have been accepted in the Centrifuge Chain, each of those request deposits could've been executed at a different price, so, this function, based on the values of maxDeposit and maxMint will estimate an average price for all the unclaimed deposits, later, using this computed price for the deposits will compute the equivalent of TrancheTokens for the CurrencyAmount being claimed, and finally, processDeposit() will transferFrom the escrow to the investor account the computed amount of TranchTokens.

```
function processDeposit(address user, uint256 currencyAmount) pu
    address liquidityPool = msg.sender;
    uint128 _currencyAmount = _toUint128(currencyAmount);
    require(
        //@audit-info => orderbook[][].maxDeposit is updated whe
        //@audit-info => The orderbook keeps track of the numbeı
        (_currencyAmount <= orderbook[user][liquidityPool].maxDe
        "InvestmentManager/amount-exceeds-deposit-limits"
    );

    //@audit-info => computes an average price for all the reque
    uint256 depositPrice = calculateDepositPrice(user, liquidity
    require(depositPrice != 0, "LiquidityPool/deposit-token-pric

    //@audit-info => Based on the computed depositPrice will com
    uint128 _trancheTokenAmount = _calculateTrancheTokenAmount(_

    //@audit-info => transferFrom the escrow to the investor acc
    _deposit(_trancheTokenAmount, _currencyAmount, liquidityPool
    trancheTokenAmount = uint256(_trancheTokenAmount);
}
```

The problem occurs when an investor hasn't claimed their deposits and has requested multiple deposits on different epochs at different prices. The `InvestmentManager::calculateDepositPrice()` function will compute an equivalent/average price for all the requestDeposits that haven't been claimed yet. Because of the different prices that the request deposits where processed at, the computed price will compute the most accurate average of the deposit's price, but there is a slight rounding error that causes the computed value of trancheTokenAmount to be slightly different from what it should exactly be.

- That slight difference will make that the Escrow contract transfers slightly more shares to the investor claiming the deposits by using the `LiquidityPool.deposit()`
- **As a result**, when another investor tries to claim their **maxDeposit** or **maxMint**, now the Escrow contract doesn't have enough shares to make whole the request of the other investor, and as a consequence the other investor transaction will be reverted. That means the second investor won't be able to claim all the shares that it is entitled to claim because the Escrow contract doesn't have all those shares anymore.

Coded PoC

- I used the `LiquidityPool.t.sol` test file as the base file for this PoC, please add the below testPoC to the LiquidityPool.t.sol file
- In this PoC I demonstrate that Alice (A second investor) won't be able to claim her maxDeposit or maxMint amounts after the first investor uses the `LiquidityPool.deposit()` function to claim his **maxDeposit() assets**. The first investor makes two requestDeposit, each of them at a different epoch and at a different price, Alice on the other hand only does 1 requestDeposit in the second epoch.
- Run this PoC two times, check the comments on the last 4 lines, one time we want to test Alice claiming her deposits using LiquidityPool::deposit(), and the second time using LiquidityPool::mint()
    - The two executions should fail with the same problem.

▶ Details

Recommended Mitigation Steps

- I'd recommend to add a check to the computed value of the `_trancheTokenAmount` in the `InvestmentManager::processDeposit()`, if the `_trancheTokenAmount` exceeds the `maxMint()` of the user, update it and set it to be the maxMint(), in this way, the rounding differences will be discarded before doing the actual transfer of shares from the Escrow to the user, and this will prevent the Escrow from not having all the required TranchToken for the other investors

```
function processDeposit(address user, uint256 currencyAmount) pu
    address liquidityPool = msg.sender;
    uint128 _currencyAmount = _toUint128(currencyAmount);
    require(
        (_currencyAmount <= orderbook[user][liquidityPool].maxDe
        "InvestmentManager/amount-exceeds-deposit-limits"
    );

    uint256 depositPrice = calculateDepositPrice(user, liquidity
    require(depositPrice != 0, "LiquidityPool/deposit-token-pric

    uint128 _trancheTokenAmount = _calculateTrancheTokenAmount(_

    //@audit => Add this check to prevent any rounding errors fr
+   if (_trancheTokenAmount > orderbook[user][liquidityPool].max

    _deposit(_trancheTokenAmount, _currencyAmount, liquidityPool
    trancheTokenAmount = uint256(_trancheTokenAmount);
}
```

- After applying the suggested recommendation, you can use the provided PoC on this report to verify that the problem has been solved.

Assessed type

Math

hieronx (Centrifuge) confirmed

hieronx (Centrifuge) commented via duplicate issue `#210`:

> I believe issues 210, 34 and 118 all come down to the same underlying issues, and I will try to respond with our current understanding (although we are still digging

> further).

> There are actually multiple rounding issues coming together in the system right now:

1. If there are multiple executions of an order, there can be loss of precision when these values are added to each other.
2. If there are multiple `deposit` or `mint` calls, there can be loss of precision in the amount of tranche tokens they receive, based on the computed deposit price.
3. If there are multiple `deposit` or `mint` calls, there can be loss of precision in the implied new price through the subtracted `maxDeposit` / `maxMint` values.

> We've written a new fuzz test, building on the work from the reported findings, that clearly shows the underlying issue.

> Unfortunately it also makes clear that this issue cannot be solved by the recommended mitigation steps from the 3 different findings on their own, and it requires deeper changes. We are still investigating this.

> Now, in terms of the severity, it does not directly lead to a loss of funds. As noted in issue 210, the `UserEscrow` logic prevents users from withdrawing more currency/funds than they are entitled to. However, it does lead to users being able to receive more tranche tokens (shares). I will leave it to the judges to make a decision on the severity for this.

> Thanks to the 3 wardens for reporting these issues!

gzeon (judge) commented:

> This issue described a protocol specific issue with multiple deposit/withdrawal, where issue 34 is a generic 4626 rounding issue, and therefore not marked as duplicate of issue 34. In terms of severity, this does not directly lead to a loss of fund but will affect the availability of the protocol, hence Medium.

hieronx (Centrifuge) commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/166

[M-06] DelayedAdmin Cannot `PauseAdmin.removePauser`

Submitted by [ciphermarco]

As per the audit repository's documentation, which is confirmed as up-to-date, there are carefully considered emergency scenarios. Among these scenarios, one is described as follows:

https://github.com/code-423n4/2023-09-centrifuge/blob/main/README.md?plain=1#L74-L77:

```
**Someone controls 1 pause admin and triggers a malicious `pause

* The delayed admin is a `ward` on the pause admin and can trigg
* It can then trigger `root.unpause()`.
```

That makes perfect sense from a security perspective. However the provided `DelayedAdmin` implementation lacks the necessary functionality to execute `PauseAdmin.removePauser` in the case of an emergency.

Striving to adhere to the documented [Severity Categorization], I have categorized this as Medium instead of Low. The reason is that it does not qualify as Low due to representing both a "function incorrect as to spec" issue and a critical feature missing from the project's security model. Without this emergency action for `PauseAdmin`, other recovery paths may have to wait for `Root`'s delay period or, at least temporarily, change the protocol's security model to make a recovery. In my view, this aligns with the "Assets not at direct risk, but the function of the protocol or its availability could be impacted" requirement for Medium severity. With that said, I realize the sponsors and judges will ultimately evaluate and categorise it based on their final risk analysis, not mine. I'm simply streamlining the process by presenting my perspective in advance.

Proof of Concept

In order to remove a pauser from the `PauseAdmin` contract, the `removePause` function must be called:

https://github.com/code-423n4/2023-09-centrifuge/blob/main/src/admins/PauseAdmin.sol#L39-L42:

```
    function removePauser(address user) external auth {
        pausers[user] = 0;
        emit RemovePauser(user);
    }
```

Since it is a short contract, here is the whole `DelayedAdmin` implementation:

https://github.com/code-423n4/2023-09-centrifuge/blob/main/src/admins/DelayedAdmin.sol

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity 0.8.21;

import {Root} from "../Root.sol";
import {Auth} from "./../util/Auth.sol";

/// @title  Delayed Admin
/// @dev    Any ward on this contract can trigger
///         instantaneous pausing and unpausing
///         on the Root, as well as schedule and cancel
///         new relys through the timelock.
contract DelayedAdmin is Auth {
    Root public immutable root;

    // --- Events ---
    event File(bytes32 indexed what, address indexed data);

    constructor(address root_) {
        root = Root(root_);

        wards[msg.sender] = 1;
        emit Rely(msg.sender);
    }

    // --- Admin actions ---
    function pause() public auth {
        root.pause();
    }
    function unpause() public auth {
        root.unpause();
    }

    function scheduleRely(address target) public auth {
        root.scheduleRely(target);
    }

    function cancelRely(address target) public auth {
        root.cancelRely(target);
    }
}
```

No implemented functionalities exist to trigger `PauseAdmin.removePauser`. Additionally, the contract features an unused `event File`, a and fails to record the `PauseAdmin`'s address upon initiation or elsewhere.

As anticipated, `Auth` (inherited) also does not handle this responsibility:

To confirm that I did not misunderstand anything, I thoroughly searched the entire audit repository for occurrences of `removePauser`. However, I could only find them in `PauseAdmin.sol`, where the function to remove a pauser is implemented, and in a test case that directly calls the `PauseAdmin`.

Recommended Mitigation Steps

1. Implement the `PauseAdmin.removePauser` Functionality in `DelayedAdmin.sol` with This Diff:

```
5a6
> import {PauseAdmin} from "./PauseAdmin.sol";
13a15
>     PauseAdmin public immutable pauseAdmin;
18c20
<     constructor(address root_) {
---
>     constructor(address root_, address pauseAdmin_) {
19a22
>         pauseAdmin = PauseAdmin(pauseAdmin_);
41,42c44,48
<     // @audit HM? How can delayed admin call `PauseAdmin.remov
<     // @audit According to documentation: "The delayed admin i
---
>
>     // --- Emergency actions --
>     function removePauser(address pauser) public auth {
>         pauseAdmin.removePauser(pauser);
>     }
```

2. Add This Test Function to `AdminTest` Contract in `test/Admin.t.sol`

```
    function testEmergencyRemovePauser() public {
        address evilPauser = address(0x1337);
        pauseAdmin.addPauser(evilPauser);
        assertEq(pauseAdmin.pausers(evilPauser), 1);

        delayedAdmin.removePauser(evilPauser);
        assertEq(pauseAdmin.pausers(evilPauser), 0);
    }
```

3. Change the `DelayedAdmin` Creation in `script/Deployer.sol` with This diff:

```
55c55
<         delayedAdmin = new DelayedAdmin(address(root));
---
>         delayedAdmin = new DelayedAdmin(address(root), address
```

4. Test

```
$ forge test
```

RaymondFam (lookout) commented:

> The sponsor has highlighted in Discord that these are EOAs capable of triggering to removePauser.

gzeon (judge) commented:

> Since the `removePauser` usecase is explicitly documented in the README and DelayedAdmin.sol is in-scope, I believe this is a valid Medium issue as the warden described.

hieronx (Centrifuge) confirmed and commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/139

[M-07] `trancheTokenAmount` should be rounded UP when proceeding to a withdrawal or previewing a withdrawal

Submitted by pipidu83, *also found by* Vagner, KrisApostolov, 0xTiwa, josephdara, 0xgrbr, PENGUN, Kral01, bitsurfer, ether_sky, 0xklh, merlin, maanas, Phantasmagoria, lsaudit, castle_chain, ladboy233, deth, IceBear, 0xPacelli, *and* MaslarovK

This is good practice when implementing the EIP-4626 vault standard as it is more secure to favour the vault than its users in that case.
This can also lead to issues down the line for other protocol integrating Centrifuge, that may assume that rounding was handled according to EIP-4626 best practices.

Proof of Concept

When calling the `processWithdraw` function, the `trancheTokenAmount` is computed through the `_calculateTrancheTokenAmount` function, which rounds DOWN the number of shares required to be burnt to receive the `currencyAmount` payout/withdrawal.

```
/// @dev Processes user's tranche token redemption after the epc
/// In case user's redempion order was fullfilled on Centrifuge
/// are increased and LiquidityPool currency can be transferred
/// Note: The trancheTokenAmount required to fullfill the redemp
/// @notice trancheTokenAmount return value is type of uint256 t
/// @return trancheTokenAmount the amount of trancheTokens redee
function processWithdraw(uint256 currencyAmount, address receive
public
auth
returns (uint256 trancheTokenAmount)
```

```
    {
    address liquidityPool = msg.sender;
    uint128 _currencyAmount = _toUint128(currencyAmount);
    require(
    (_currencyAmount <= orderbook[user][liquidityPool].maxWithdraw &
    "InvestmentManager/amount-exceeds-withdraw-limits"
    );

    uint256 redeemPrice = calculateRedeemPrice(user, liquidityPool);
    require(redeemPrice != 0, "LiquidityPool/redeem-token-price-0");

    uint128 _trancheTokenAmount = _calculateTrancheTokenAmount(_curr
    _redeem(_trancheTokenAmount, _currencyAmount, liquidityPool, rec
    trancheTokenAmount = uint256(_trancheTokenAmount);
    }



    function _calculateTrancheTokenAmount(uint128 currencyAmount, ac
    internal
    view
    returns (uint128 trancheTokenAmount)
    {
    (uint8 currencyDecimals, uint8 trancheTokenDecimals) = _getPoolI

    uint256 currencyAmountInPriceDecimals = _toPriceDecimals(currenc
    10 ** PRICE_DECIMALS, price, MathLib.Rounding.Down
    );

    trancheTokenAmount = _fromPriceDecimals(currencyAmountInPriceDec
    }
```

As an additional reason the round UP the amount, the computed amount of shares is also used to `_decreaseRedemptionLimits` , which could potentially lead to a rounded UP remaining redemption limit post withdrawal (note that for the same reason it would we wise to round UP the `_currency` amount as well when calling `_decreaseRedemptionLimits`).

The same function is used in the `previewWithdraw` function, where is should be rounded UP for the same reasons.

```
    /// @return trancheTokenAmount is type of uin256 to support the
    function previewWithdraw(address user, address liquidityPool, ui
    public
    view
    returns (uint256 trancheTokenAmount)
    {
    uint128 currencyAmount = _toUint128(_currencyAmount);
    uint256 redeemPrice = calculateRedeemPrice(user, liquidityPool);
    if (redeemPrice == 0) return 0;

    trancheTokenAmount = uint256(_calculateTrancheTokenAmount(currer
    }
```

Tools Used

Visual Studio / Manual Review

Recommended Mitigation Steps

As the we do not always want to round the amount of shares UP in `_calculateTrancheTokenAmount` (e.g. when used in `previewDeposit` or `processDeposit` the shares amount is correctly rounded DOWN), the function would actually require an extra argument like below:

```
    function _calculateTrancheTokenAmount(uint128 currencyAmount, ac
    internal
    view
    returns (uint128 trancheTokenAmount)
    {
    (uint8 currencyDecimals, uint8 trancheTokenDecimals) = _getPoolI

    uint256 currencyAmountInPriceDecimals = _toPriceDecimals(currenc
    10 ** PRICE_DECIMALS, price, MathLib.Rounding.Down
    );

    trancheTokenAmount = _fromPriceDecimals(currencyAmountInPriceDec
    }
```

And be used as

```
    _calculateTrancheTokenAmount(currencyAmount, liquidityPool, rede
```

In `previewWithdraw` and `processWithdraw`

And

```
    _calculateTrancheTokenAmount(_currencyAmount, liquidityPool, dep
```

In `previewDeposit` and `processDeposit`.

Assessed type

Math

hieronx (Centrifuge) confirmed and commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/166

[M-08] The Restriction Manager does not completely implement ERC1404 which leads to accounts that are supposed to be restricted actually having access to do with their tokens as they see fit

Submitted by Bauchibred*, also found by* josephdara, ast3ros, Ch_301, BARW, 0xnev, degensec*, and* J4X

Medium, contract's intended logic is for *blacklisted* users not to be able to interact with their system so as to follow rules set by regulationary bodies in the case where a user does anything that warrants them to be blacklisted, but this is clearly broken since only half the window is closed as current implementation only checks on receiver being blacklisted and not sender.

Proof of Concept

The current implementation of the ERC1404 restrictions within the `RestrictionManager.sol` contract only places restrictions on the receiving address in token transfer instances. This oversight means that the sending addresses are not restricted, which poses a regulatory and compliance risk. Should a user be `blacklisted` for any reason, they can continue to transfer tokens as long as the receiving address is a valid member. This behaviour is contrary to expectations from regulatory bodies, especially say in the U.S where these bodies are very strict and a little in-compliance could land Centrifuge a lawsuit., which may expect complete trading restrictions for such blacklisted individuals.

Within the `RestrictionManager` contract, the method `detectTransferRestriction` only checks if the receiving address (`to`) is a valid member:

[RestrictionManager.sol#L28-L34]

```
    function detectTransferRestriction(address from, address to, uir
        if (!hasMember(to)) {
            return DESTINATION_NOT_A_MEMBER_RESTRICTION_CODE;
        }
        return SUCCESS_CODE;
    }
```

In the above code, the sending address (`from`) is never checked against the membership restrictions, which means blacklisted users can still initiate transfers and when checking the transfer restriction from both `tranchtoken.sol` and the `liquiditypool.sol` it's going to wrongly return true for a personnel that should be false

See [Tranche.sol#L80-L82)]

```
    // function checkTransferRestriction(address from, address to, ι
    //     return share.checkTransferRestriction(from, to, value);
    // }
```

Also [Tranche.sol#L35-L39]

```
        modifier restricted(address from, address to, uint256 value)
            uint8 restrictionCode = detectTransferRestriction(from,
            require(restrictionCode == restrictionManager.SUCCESS_C(
            _;
        }
```

This function suggests that the system's logic may rely on the `detectTransferRestriction` method in other parts of the ecosystem. Consequently, if the restriction manager's logic is flawed, these other parts may also allow unauthorised transfers.

Foundry POC

Add this to the `Tranche.t.sol` contract

```
        function testTransferFromTokensFromBlacklistedAccountWorks(u
            vm.assume(baseAssumptions(validUntil, targetUser));

            restrictionManager.updateMember(targetUser, validUntil);
            assertEq(restrictionManager.members(targetUser), validUn
            restrictionManager.updateMember(address(this), block.tim
            assertEq(restrictionManager.members(address(this)), bloc

            token.mint(address(this), amount);
            vm.warp(block.timestamp + 1);

            token.transferFrom(address(this), targetUser, amount);
            assertEq(token.balanceOf(targetUser), amount);
        }
```

As seen even after `address(this)` stops being a member they could still transfer tokens to another user in as much as said user is still a member, which means a *blacklisted* user could easily do anything with their tokens all they need to do is to delegate to another member.

Recommended Mitigation Steps

Refactor `detectTransferRestriction()` , i.e modify the method to also validate the sending address (`from`). Ensure both `from` and `to` addresses are valid members before allowing transfers.

If this is contract's intended logic then this information should be duly passed to the regulatory bodies that a `user` can't really get blacklisted and more of user's could be stopped from receiving tokens.

Assessed type

Context

hieronx (Centrifuge) commented:

> This is a design decision. Transferring to a valid member is fine if that destination is still allowed to hold the tokens.

gzeon (judge) commented:

> This is a design decision. Transferring to a valid member is fine if that destination is still allowed to hold the tokens.

> This seems to contradict with `Removing an investor from the memberlist in the Restriction Manager locks their tokens. This is expected behaviour.` (See here)

hieronx (Centrifuge) commented:

> Yes you're right, that's unfortunate wording. What I said above is correct, and the text in the README is incorrect. What was meant was that it locks their tranche tokens from being redeemed.

> It is indeed fair to say though that, based on the README text, the above issue is valid.

hieronx (Centrifuge) commented:

> Mitigated in https://github.com/centrifuge/liquidity-pools/pull/138

Low Risk and Non-Critical Issues

For this audit, 39 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **castle_chain** received the top score from the judge.

The following wardens also submitted reports: Bauchibred, 0xpiken, BARW, 0xmystery, 0xfuje, 0xLook, merlin, MohammedRizwan, jkoppel, jolah1, alexzoid, sandy, btk, Kaysoft, klau5, ast3ros, 0xAadi, Bughunter101, 0xblackskull, 0xnev, SanketKogekar, m_Rassska, grearlake, nobody2018, 7ashraf, JP_Courses, Krace, Ch_301, lsaudit, catellatech, imtybik, mrudenko, Sathish9098, rvierdiiev, rokinot, fatherOfBlocks, 0xHelium, *and* degensec.

[L-01] The function `addTranche()` should validate the tranche token decimals to make sure that the decimals are not greater than 18

```
    function addTranche(
        uint64 poolId,
        bytes16 trancheId,
        string memory tokenName,
        string memory tokenSymbol,
        uint8 decimals
    ) public onlyGateway {
        Pool storage pool = pools[poolId];
        require(pool.createdAt != 0, "PoolManager/invalid-pool")
        Tranche storage tranche = pool.tranches[trancheId];
        require(tranche.createdAt == 0, "PoolManager/tranche-alr


        tranche.poolId = poolId;
        tranche.trancheId = trancheId;
@>      tranche.decimals = decimals;
        tranche.tokenName = tokenName;
        tranche.tokenSymbol = tokenSymbol;
        tranche.createdAt = block.timestamp;


        emit TrancheAdded(poolId, trancheId);
    }
```

[L-02] Unlimiting the delay period with minimum threshold allows the delay period to be set too low and allows a

malicious `ScheduleUpgrade` message to be executed on the root contract and gain access on the other contracts

Note: this finding was originally submitted separately by the warden at a higher severity; however, it was downgraded to low severity and considered by the judge in scoring. It is being included here for completeness.

Allowing to modify the delay period without minimum threshold can result in set the `delay` period to a too short period or even zero ,which is allowed according to the code , and this could lead to allowing a malicious `ScheduleUpgrade` message to pass and a malicious account can gain control over all the contracts of the protocols.

Proof of Concept

In the `Root.sol` contract, the protocol implements a maximum threshold in the function `file()` which can modify the `delay` variable, so the possibility of the set the delay period to a very long period exists and also setting the delay period to a very short period is still possible, as shown in the `file()` which will not prevent set the `delay` period to zero or any short period :

```
    function file(bytes32 what, uint256 data) external auth {
        if (what == "delay") {
            require(data <= MAX_DELAY, "Root/delay-too-long");
            delay = data;
        } else {
            revert("Root/file-unrecognized-param");
        }
        emit File(what, data);
```

So if the `delay` was set to a very short period and Someone gains control over a router and triggers a malicious `ScheduleUpgrade` message , the malicious attacker can trigger the `executeScheduledRely()` function after the `delay` period, which is too short, to give his address or any contract an `auth` role on any of the protocol contracts, the attacker can be `auth` on the `escrow` contract and steals all the funds from the protocol.

This attack does not assume that the `auth` admin on the `Root` contract is malicious, but it is all about the possibility of setting the `delay` period to a very short period or even zero, which can be happened mistakenly or intentionally.

Recommended Mitigation Steps

This vulnerability can be mitigated by implementing a minimum threshold `MIN_DELAY` to the `delay` period, so this will prevent setting the `delay` period to zero or even a very short period.

```
+    uint256 private MIN_DELAY = 2 days ;

    function file(bytes32 what, uint256 data) external auth {
        if (what == "delay") {
            require(data <= MAX_DELAY, "Root/delay-too-long");
+           require(data >= MIN_DELAY, "Root/delay-too-short");

            delay = data;
        } else {
            revert("Root/file-unrecognized-param");
        }
        emit File(what, data);
```

[N-01] Unnecessary checks can be removed, to enhance the code

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L310

The `require` keyword can be removed in the function `deployLiquidityPool ()`, because the function `isAllowedAsPoolCurrency` revert on failure in case of the currency is not supported and always return true on success, so if the check fails the function will revert without executing the `require` key word.

```
    function isAllowedAsPoolCurrency(uint64 poolId, address curr
        uint128 currency = currencyAddressToId[currencyAddress];
        require(currency != 0, "PoolManager/unknown-currency");
        require(pools[poolId].allowedCurrencies[currencyAddress]
        return true;


    function deployLiquidityPool(uint64 poolId, bytes16 trancheId,
        Tranche storage tranche = pools[poolId].tranches[tranche
        require(tranche.token != address(0), "PoolManager/tranch
@>      require(isAllowedAsPoolCurrency(poolId, currency), "Pc

        address liquidityPool = tranche.liquidityPools[currency]
        require(liquidityPool == address(0), "PoolManager/liquic
        require(pools[poolId].createdAt != 0, "PoolManager/pool-
```

[N-02] The `destination` address should be emitted in the userEscrow contract in the function `transferOut`

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/UserEscrow.sol#L49

Due to that the receiver can be different address from the user address , the `destination` address should be emitted.

```
    function transferOut(address token, address destination, adc
        require(destinations[token][destination] >= amount, "Use
        require(
            receiver == destination || (ERC20Like(token).allowar
            "UserEscrow/receiver-has-no-allowance"
        );
        destinations[token][destination] -= amount;

        SafeTransferLib.safeTransfer(token, receiver, amount);
        emit TransferOut(token, receiver, amount);
    }
```

[N-03] The `getTranche` function should be created in the `poolManager` contract and used to get the `tranche` from the storage by specifying the `poolId` and `trancheId`

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L308

The function `getTranche()` will increase the simplicity and the modularity of the code instead of get the tranche from the storage each time in different functions.

```
    function getTranche (uint64 poolId , bytes16 trancheId) private
              tranche =  pools[poolId].tranches[trancheId];
```

This function can be used instead of the line of code in the poolManager.

```
            Tranche storage tranche = pools[poolId].tranches[tranche
```

[N-04] The check of the creation of the liquidity pool in the function `deployLiquidityPool()` in the PoolManager contract can be removed

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L309

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L314

The `require` statement that checks that the pool is created can be removed since there is a check before it that make sure that the tranche token is deployed and the address of it is stored in the `pool` struct which be be stored only after the pool has been created, so the check of the deployment of the tranche token do the same check of the creation of the pool, so the second check can be removed.

The check in the function `addTranche()` , which make sure that the pool is created.

```
        function addTranche(
        uint64 poolId,
        bytes16 trancheId,
        string memory tokenName,
        string memory tokenSymbol,
        uint8 decimals
    ) public onlyGateway {
        Pool storage pool = pools[poolId];
@>      require(pool.createdAt != 0, "PoolManager/invalid-pool")
        Tranche storage tranche = pool.tranches[trancheId];
        require(tranche.createdAt == 0, "PoolManager/tranche-alr
```

And the check of the creation of the pool in the function `deployLiquidityPool()` , which make sure that the tranche token is added and deployed.

```
    function deployLiquidityPool(uint64 poolId, bytes16 trancheI
        Tranche storage tranche = pools[poolId].tranches[tranche
        require(tranche.token != address(0), "PoolManager/tranch
        require(isAllowedAsPoolCurrency(poolId, currency), "Pool


        address liquidityPool = tranche.liquidityPools[currency]
        require(liquidityPool == address(0), "PoolManager/liquic
        require(pools[poolId].createdAt != 0, "PoolManager/pool-
```

So the check `require(pools[poolId].createdAt != 0, "PoolManager/pool-does-not-exist");` can be removed.

[N-05] The function `updateMember()` change the state of the contract, but there is no emitted event, so an event should be emitted

The function `updateMember` in the restriction manager contract should emit the event `updatedMember()` with the right parameters.
https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/token/RestrictionManager.sol#L57-L60

```
    function updateMember(address user, uint256 validUntil) publ
```

```
            require(block.timestamp <= validUntil, "RestrictionManag
            members[user] = validUntil;
        }
```

[N-06] The `poolManager` should emit events in case of transfer the tokens to the centrifuge chains

In the functions `transferTrancheTokensToCentrifuge`, `transferTrancheTokensToEVM` and `transfer()` which burn the tokens and transfer the assets, which is a crucial part of the contract that should emit events in case of transfer of the assets and the burn of the tokens.

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L136-L147

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L149-L163

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/PoolManager.sol#L128-L134

[N-07] `checkTransferRestriction` check can be removed because the tranche token `transferFrom` function do this check before transfer the tokens

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/InvestmentManager.sol#L473
https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/InvestmentManager.sol#L474-L477

```
            require(lPool.checkTransferRestriction(msg.sender, user,
            require(
                lPool.transferFrom(address(escrow), user, trancheTok
                "InvestmentManager/trancheTokens-transfer-failed"
            );
```

The `checkTransferRestriction` is performed in the `transferFrom` call of the tranche token.

[N-08] Consider adding `getLpValues()` to get the lpValuse from the orderBook to enhance the code

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/InvestmentManager.sol#L247

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/InvestmentManager.sol#L268

https://github.com/code-423n4/2023-09-centrifuge/blob/512e7a71ebd9ae76384f837204216f26380c9f91/src/InvestmentManager.sol#L561

The contract repeats the code of getting the lpValuse from the storage more than 10 times so the function `getLpValues()` should be added.

```
    function _getLpValues(address user , address liquidityPool) priv
          lpValues =  orderbook[user][liquidityPool];
    }
```

hieronx (Centrifuge) confirmed

Audit Analysis

For this audit, 20 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **ciphermarco** received the top score from the judge.

The following wardens also submitted reports: cats, Sathish9098, kaveyjoe, 0xnev, 0xbrett8571, catellatech, jaraxxus, Kral01, fouzantanveer, 0xmystery, K42, hals, grearlake, rokinot, castle_chain, lsaudit, emerald7017, 0x3b, *and* foxb868.

1. Executive Summary

For this analysis, I will center my attention on the scope of the ongoing audit `2023-09-centrifuge`. I begin by outlining the code audit approach employed for the in-scope contracts, followed by sharing my perspective on the architecture, and concluding with observations about the implementation's code.

Please note that unless explicitly stated otherwise, any architectural risks or implementation issues mentioned in this document are not to be considered vulnerabilities or recommendations for altering the architecture or code solely based on this analysis. As an auditor, I acknowledge the importance of thorough evaluation for design decisions in a complex project, taking associated risks into consideration as one single part of an overarching process. It is also important to recognise that the project team may have already assessed these risks and determined the most suitable approach to address or coexist with them.

2. Code Audit Approach

Time spent: 18 hours

2.1 Audit Documentation and Scope

The initial step involved examining audit documentation and scope to grasp the audit's concepts and boundaries, and prioritise my efforts. It is worth highlighting the good quality of the `README` for this audit, as it provides valuable insights and actionable guidance that greatly facilitate the onboarding process for auditors.

2.2 Setup and Tests

Setting up to execute `forge test` was remarkably effortless, greatly enhancing the efficiency of the auditing process. With a fully functional test harness at our disposal, we not only accelerate the testing of intricate concepts and potential vulnerabilities but also gain insights into the developer's expectations regarding the implementation. Moreover, we can deliver more value to the project by incorporating our proofs of concept into its tests wherever feasible.

2.3 Code review

The code review commenced with understanding the " `ward` pattern" used to manage authorization accross the system. Thoroughly understanding this pattern made understanding the protocol contracts and its relations much smoother. Throughout this stage, I documented observations and raised questions concerning potential exploits without going too deep.

2.4 Threat Modelling

I began formulating specific assumptions that, if compromised, could pose security risks to the system. This process aids me in determining the most effective exploitation strategies. While not a comprehensive threat modeling exercise, it shares numerous similarities with one.

2.5 Exploitation and Proofs of Concept

From this step forward, the main process became a loop conditionally encompassing each of the steps 2.3, 2.4, and 2.5, involving attempts at exploitation and the creation of proofs of concept with a little help of documentation or the ever helpful sponsors on Discord. My primary objective in this phase is to challenge critical assumptions, generate new ones in the process, and enhance this process by leveraging coded proofs of concept to expedite the development of successful exploits.

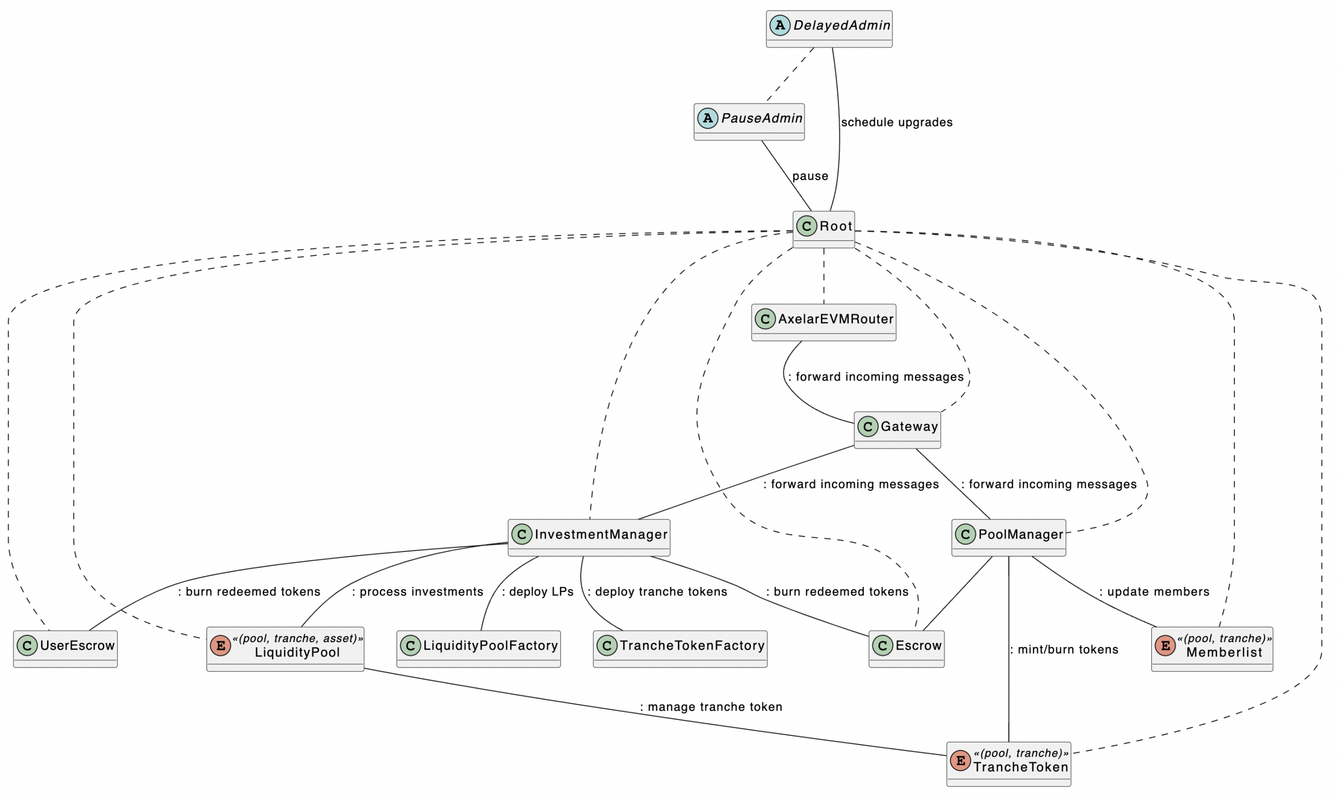
2.6 Report Issues

Alghough this step may seem self-explanatory, it comes with certain nuances. Rushing to report vulnerabilities immediately and then neglecting them is unwise. The most effective approach to bring more value to sponsors (and, hopefully, to auditors) is to document what can be gained by exploiting each vulnerability. This assessment helps in evaluating whether these exploits can be strategically combined to create a more significant impact on the system's security. In some cases, seemingly minor and moderate issues can compound to form a critical vulnerability when leveraged wisely. This has to be balanced with any risks that users may face. In the context of Code4rena audits, zero-days or highly sensitive bugs affecting deployed contracts are given a more cautious and immediate reporting channel.

3. Architecture overview

3.1 `Ward` Pattern

In grasping the audited architecture, one must comprehend some simple yet potent principles. The first concept is the `ward` pattern, employed to oversee authorisation throughout the protocol. I consider it an elegant and secure method. Below is the 'ward' graph supplied by the Centrifuge team:



And, to better contextualise, this is a quote from the audit's `README`:

```
The Root contract is a ward on all other contracts. The PauseAdm
```

So how does this work in practice?

As `Root` is a ward on all other contracts, any ward on `Root` can use its `relyContract` function to become a ward in any of these contracts:

https://github.com/code-423n4/2023-09-centrifuge/blob/main/src/Root.sol#L90-L93:

```
        function relyContract(address target, address user) public a
            AuthLike(target).rely(user);
            emit RelyContract(target, user);
        }
```

The opposite being the `denyContract` function:

https://github.com/code-423n4/2023-09-centrifuge/blob/main/src/Root.sol#L98-L102:

```
        function denyContract(address target, address user) public a
            AuthLike(target).deny(user);
            emit DenyContract(target, user);
        }
    }
```

The term "rely" is employed to signify "becoming a ward." Thus, when I state that contract X relies on contract Y, it implies that contract Y assumes the role of a ward within contract X. Conversely, the verb "deny" is utilised to describe the opposing action.

Contracts within the system are anticipated to inherit (or implement) the `Auth` contract. The `Auth` contract is exceptionally minimal, and for your reference, I have included it below:

```
    // SPDX-License-Identifier: AGPL-3.0-only
    // Copyright (C) Centrifuge 2020, based on MakerDAO dss https://
    pragma solidity 0.8.21;

    /// @title  Auth
    /// @notice Simple authentication pattern
    contract Auth {
        mapping(address => uint256) public wards;

        event Rely(address indexed user);
        event Deny(address indexed user);

        /// @dev Give permissions to the user
        function rely(address user) external auth {
            wards[user] = 1;
            emit Rely(user);
        }

        /// @dev Remove permissions from the user
        function deny(address user) external auth {
            wards[user] = 0;
            emit Deny(user);
        }

        /// @dev Check if the msg.sender has permissions
        modifier auth() {
            require(wards[msg.sender] == 1, "Auth/not-authorized");
            _;
        }
    }
```

With these capabilities at their disposal, any `ward` in `Root` has the ability to invoke `root.allowContract(address target, address user)`. This action designates `user` — whether an Externally Owned Account (EOA) or Contract — as a `ward` within the `contract`. As a result, the `Root`'s existing `wards` can execute functions within any of the contracts to which `Root` serves as a `ward` (i.e. all other contracts in the system).

3.1.1 The `PauseAdmin`

The `PauseAdmin` represents a concise contract equipped with functions to manage `pausers`, those who can call its `pause` function. Its `pause` function, in turn, invokes the `Root.pause` function to suspend protocol operations. We will shortly delve into the mechanics of this pausing process. To facilitate the ability to call `Root.pause`, `PauseAdmin` holds the status of a `ward` within the `Root` contract.

3.1.2 The `DelayedAdmin`

The `DelayedAdmin` is another concise contract with the capability to pause and unpause the protocol. Additionally, it possesses the ability, after a `delay` set in the

`Root` contract by its deployer (the first ward) and other wards, to designate itself or other Externally Owned Accounts (EOAs) or contracts as `wards` within the `Root` contract. This delay is enforced due to the `DelayedAdmin` implementing a function to invoke `root.scheduleRely`. The `root.scheduleRely` function is responsible for scheduling an address to become a `ward` within the `Root` contract.

The `pauseAdmin.scheduleRely` function:

```
// PauseAdmin.sol
function scheduleRely(address target) public auth {
    root.scheduleRely(target);
}
```

Which calls the `root.scheduleRely`:

```
// Root.sol
function scheduleRely(address target) external auth {
    schedule[target] = block.timestamp + delay;
    emit RelyScheduled(target, schedule[target]);
}
```

As per the documentation provided, the initial setting for this `delay` is 48 hours. Once a rely is scheduled, it can subsequently be cancelled using the `delayedAdmin.cancelRely` function:

```
// PauseAdmin.sol
function cancelRely(address target) public auth {
    root.cancelRely(target);
}
```

That function calls the `root.cancelRely`:

```
// Root.sol
function cancelRely(address target) external auth {
    schedule[target] = 0;
    emit RelyCancelled(target);
```

```
        }
```

Or executed to make it effective by `executeScheduledRely` in the `Root` contract after the `delay` has passed:

```
        // Root.sol
        function executeScheduledRely(address target) public {
            require(schedule[target] != 0, "Root/target-not-schedule
            require(schedule[target] < block.timestamp, "Root/target

            wards[target] = 1;
            emit Rely(target);

            schedule[target] = 0;
        }
```

3.1.3 A Problem with `DelayedAdmin`

Whilst conducting a comparison between the audit documentation and the system, I identified a critical discrepancy in the `DelayedAdmin`. The project team outlined several potential emergency scenarios to aid auditors in comprehending the protocol's security model. One of these scenarios is outlined below:

> **"Someone controls 1 pause admin and triggers a malicious** `pause()`

- The delayed admin is a `ward` on the pause admin and can trigger `PauseAdmin.removePauser`.
- It can then trigger `root.unpause()`."

The issue I have identified is that the `DelayedAdmin`, in its current form, does not incorporate any functions to `PauseAdmin.removePauser`. Given that I believe this deviates from a critical aspect of the protocol's security model, I have chosen to report this discovery with a Medium severity rating. However, I acknowledge that it could be argued as more appropriate for a Low severity rating if it is ultimately deemed a mere deviation from the specification. I have expressed my viewpoint in the issue report, and I am confident that it will be evaluated with care and a comprehensive consideration of the real-world risks it may present, which might diverge from my own assessment.

Following discussions with sponsors to gain deeper insights into the implications of a `DelayedAdmin` unable to execute `pauseAdmin.removePause`, they exposed two workarounds to address this issue. These workarounds do not alter my stance on the severity within the context of an audit competition but offer valuable perspectives from the project team to enhance our understanding of the system's security model and its ability to withstand unforeseen failures.

3.1.3.1 The Delayed Rescue

In a delayed rescue the project team could use the `DelayedAdmin` to:

1. Call `root.scheduleRely` to make a contract or EOA (let's call it `delayedRescuer`) an `ward` in the `Root` contract;
2. Wait the set `delay`, initially be set to 48 hours, to pass;
3. Use `Rescuer` to call `root.relyContract(address(pauseAdmin), address(delayedRescuer))` to become a ward on the `pauseAdmin`;
4. Finally call `pauseAdmin.removePauser(address(maliciousPauser))` to remove the malicious `pauser`.

It is elegant but comes with the drawback of having to wait for the `delay` set in the `Root` contract, which is not the intended design of this setup.

3.1.3.2 The Prompt Rescue

In the prompt rescue, a contract (let's call it `promptRescuer`) could be made a ward in the `root` contract to:

1. Call `root.relyContract(address(pauseAdmin), address(promptRescuer))`;
2. Call `pauseAdmin.removePauser(address(maliciousPauser))`;
3. Possibly call `root.denyContract(address(promptRescuer))`;

In `Root`, when one contract designates another as its ward, it always introduces certain risks. However, by employing the Spell pattern, it significantly mitigates at least one of these risks.

3.2 The Spell Pattern

Having understood how these two admins fit into the `ward` pattern, and analysing the rescue operation, shows how simple and potentially powerful the pattern can be. But to realise its power in a more secure manner, we need to enter the Spell pattern.

References to spells can be identified within the codebase's tests, and the sponsors have publicly shared their intentions through the audit competition's Discord channel. When utilising `root.relyContract`, it is essential to anticipate that the intended targets for the `wards` in `Root` will employ the spell pattern from MakerDAO. Besides implementation details and other factors to consider, the utmost detail to retain in the context of this analysis is that *"a spell can only be cast once"*.

By joining the project's `ward` pattern with the Spell pattern, we can fully understand how critical security operations are expected to flow throughout the system.

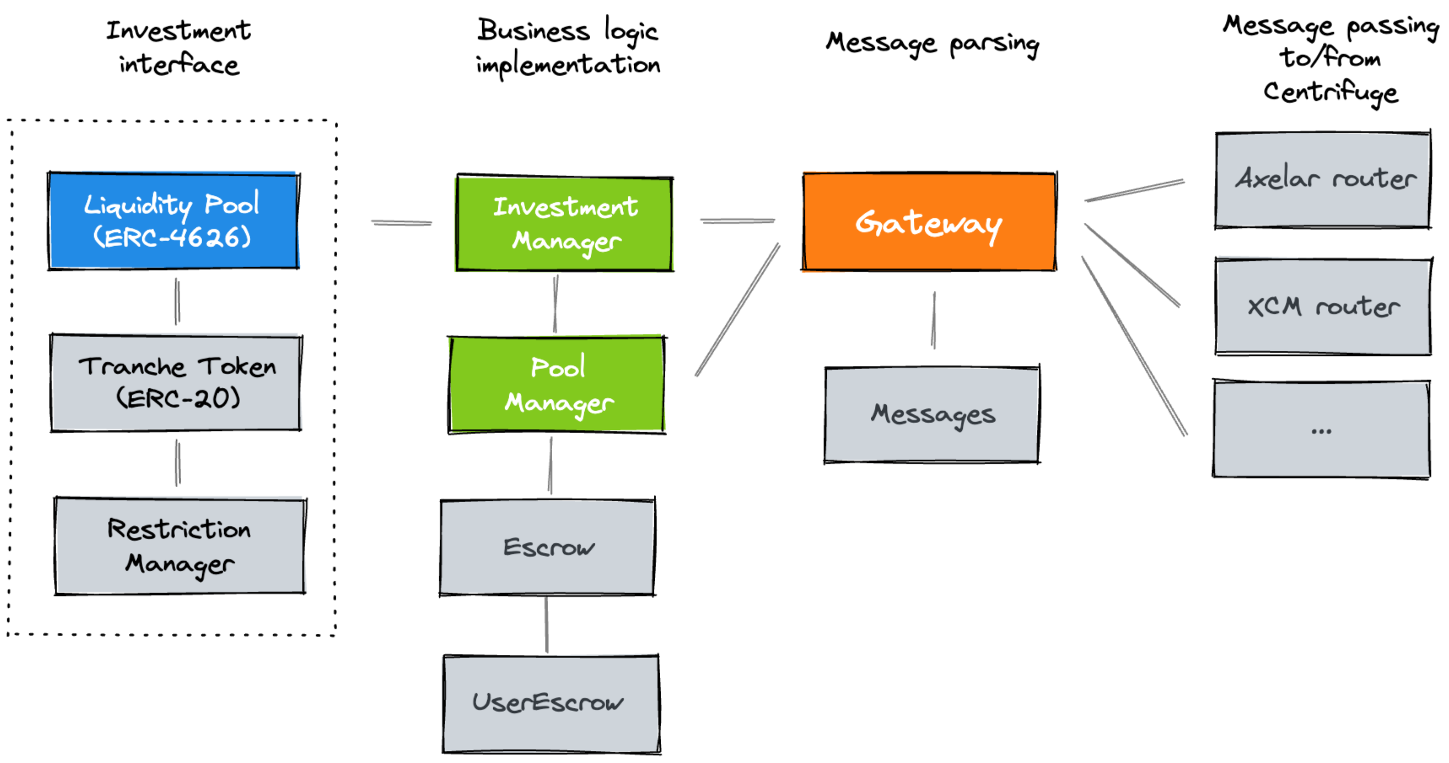
3.3 Pausing the Protocol

As mentioned before, the `PauseAdmin` has the responsibility to manage the `pausers` who can pause the protocol. We have seen that the `pauseAdmin`, triggered by one of its `pausers`, calls the `pause` function in the `Root` contract. The `Root` contract then sets the variable `bool public paused` to `true`. Here is the `root.pause` function's implementation:

```
// Root.sol
function pause() external auth {
    paused = true;
    emit Pause();
}
```

But how can it pause the whole protocol?

The protocol's flow of communication uses a hub-and-spoke model with the `Gateway` in the middle. According to the audit's `README` documentation, the `Gateway` contract is an intermediary contract that encodes and decodes messages using the `Message` contract, and handles routing to and from Centrifuge. This is the graph made available by the project team:



The `Gateway` implements the `pauseable` modifier:

```
// Gateway.sol
modifier pauseable() {
    require(!root.paused(), "Gateway/paused");
    _;
}
```

This modifier is employed in all critical incoming and outgoing operations within the `Gateway`. Then if `root.paused` is set to `true`, it halts the "hub" by blocking these operations, thus interrupting the flow of communication in the protocol.

3.3.1 Redundancy?

It is important to highlight that both `DelayedAdmin` and `PauseAdmin` have the capability to invoke `root.pause` directly in order to pause the protocol. What might initially appear as redundancy seems to be, in fact, a well-considered implementation of secure compartmentalization.

`DelayedAdmin` possesses a broader range of powers beyond the functions to `pause` and `unpause` the protocol. In contrast, `PauseAdmin` primarily oversees `pausers` with the singular purpose of allowing them to pause the protocol. Considering that `DelayedAdmin` has the potential to cause more harm to the protocol, it should be subject to a stricter oversight, while `pausers` may adhere to a less stringent regime. While `PauseAdmin` can disrupt the protocol and cause damage by initiating pauses, this impact pales in comparison to what `DelayedAdmin` could potentially do if not restricted during the `Root`'s `delay`.

3.4 Liquidity Pools

Supported by this architecture lies the liquidity pools. These liquidity pools can be deployed on any EVM-compatible blockchain, whether it's on a Layer 1 or Layer 2, in response to market demand. All of these systems are ultimately interconnected with the Centrifuge chain, which holds the responsibility of managing the Real-World Assets (RWA) pools:

Liquidity Pools

Every tranche, symbolized by a Tranche Token (TT), represents varying levels of risk exposure for investors. For an overview of the authorization relations, you can refer to the Ward Pattern section, and to understand the communication flow, visit the Pausing the Protocol section.

3.5 Upgrading the Protocol

Another interesting aspect of the architecture is the deliberate choice to use contract migrations instead of upgradeable proxy patterns. Upgrades to the protocol can be achieved by utilising the Spell pattern to re-organize `wards` and "rely" links across the system. To initiate an upgrade, a Spell contract is scheduled through the `root.scheduleRely` function. After the specified `delay` period, the upgrade can be executed by using `root.executeScheduledRely` to complete its tasks in a one-shot fashion. Should the need to cancel the upgrade arise, `root.cancelRely` can be called to cancel the scheduled rely operation."

3.6 Centralisation Risks

In the context of this analysis, it is imperative to underscore that the usage of "centralisation" specifically pertains to the potential single point of failure within the project team's assets, such as the scenario where only one key needs to be compromised in order to compromise the whole system. I am not delving into concerns associated with the likelihood of collusion amongst key personnel.

The most significant concern regarding centralisation lies in the `Root` contract's `wards`. Therefore, it is highly recommended to manage these `wards` through multi-signature schemes to mitigate the risk of a single point of failure within the system. This encompasses any contract that could potentially become a `ward` in `Root`, including contracts like `DelayedAdmin` despite its temporary limitation of having to await the `delay` set in the `Root` contract.

Another noteworthy potential single point of failure lies in the `Router` contract, should it become compromised, as mentioned in the audit's `README` documentation. This situation can be rectified through a sequence of operations involving the `PauseAdmin` and `DelayedAdmin`, or even solely relying on the `DelayedAdmin`'s capabilities.

There are other critical entities within the architecture that have the potential to act as single points of failure, leading to varying degrees of damage. However, for the specific use case at hand, I find the existing tools and safeguards to be adequate for mitigating risks, particularly when keys are safeguarded through multi-signature schemes and other strategies that require careful consideration. These aspects are beyond the scope of this analysis; the same applies to what occurs within the Centrifuge chain.

4. Implementation Notes

Throughout the audit, certain implementation details stood out as note worthy, and a portion of these could prove valuable for this analysis.

4.1 General Impressions

The codebase stood to what is promised in the audit's `README`, so I will just paste it here:

> The coding style of the liquidity-pools code base is heavily inspired by MakerDAO's coding style. Composition over inheritance, no upgradeable proxies but rather using contract migrations, and as few dependencies as possible. Authentication uses the ward pattern, in which addresses can be relied or denied to get access. Key parameter updates of contracts are executed through file methods.

It was genuinely enjoyable to review this codebase. The code has been meticulously designed to prioritise simplicity. It strikes the right balance of complexity, avoiding unnecessary complications.

4.2 Composition over Inheritance

The codebase's pivotal choice of favoring composition over inheritance, coupled with thoughtful coding practices, has allowed for the creation of clean and very comprehensible code, free from convoluted inheritance hierarchies that confuse even the project developers themselves. The significance of this choice in enhancing the security of the protocol cannot be overstated. Clear and well-organised code not only prevents the introduction of vulnerabilities but also simplifies the process of detecting and resolving any potential issues. And, my opinion is that opting for composition over inheritance significantly contributed to achieving this objective.

4.3 Tests

As mentioned in **2.2 Setup and Tests**, executing the tests using `forge` was effortless, significantly expediting our work as auditors. Running `forge coverage` also provides a convenient table summarizing the code coverage for these tests:

```
$ forge coverage
[... snip ...]
| File                                     | % Lines             | %
|------------------------------------------|---------------------|---
| script/Axelar.s.sol                      | 0.00% (0/32)        | 0.
| script/Deployer.sol                      | 0.00% (0/40)        | 0.
| script/Permissionless.s.sol              | 0.00% (0/20)        | 0.
| src/Escrow.sol                           | 100.00% (2/2)       | 10
| src/InvestmentManager.sol                | 97.37% (185/190)    | 96
| src/LiquidityPool.sol                    | 100.00% (64/64)     | 10
| src/PoolManager.sol                      | 95.96% (95/99)      | 94
| src/Root.sol                             | 100.00% (22/22)     | 10
| src/UserEscrow.sol                         | 100.00% (8/8)      | 1(
| src/admins/DelayedAdmin.sol                | 100.00% (4/4)      | 1(
| src/admins/PauseAdmin.sol                  | 100.00% (5/5)      | 1(
| src/gateway/Gateway.sol                    | 93.42% (71/76)     | 91
| src/gateway/Messages.sol                   | 59.26% (96/162)    | 5
| src/gateway/routers/axelar/Router.sol | 100.00% (8/8)      | 1(
| src/gateway/routers/xcm/Router.sol    | 0.00% (0/38)       | 0.
| src/token/ERC20.sol                        | 97.40% (75/77)     | 9
| src/token/RestrictionManager.sol           | 100.00% (15/15)    | 1(
| src/token/Tranche.sol                      | 100.00% (18/18)    | 1(
| src/util/Auth.sol                          | 100.00% (4/4)      | 1(
| src/util/BytesLib.sol                      | 0.00% (0/28)       | 0.
| src/util/Context.sol                       | 100.00% (1/1)      | 1(
| src/util/Factory.sol                       | 100.00% (24/24)    | 1(
| src/util/MathLib.sol                       | 0.00% (0/30)       | 0.
| src/util/SafeTransferLib.sol               | 100.00% (7/7)      | 1(
| test/TestSetup.t.sol                       | 0.00% (0/49)       | 0.
| test/mock/AxelarGatewayMock.sol            | 100.00% (4/4)      | 1(
| test/mock/GatewayMock.sol                  | 2.13% (1/47)       | 2.
| test/mock/Mock.sol                         | 25.00% (1/4)       | 2
| Total                                      | 65.86% (710/1078) | 6
```

Coverage, while not providing a comprehensive view of what is tested and how thoroughly, can serve as a valuable indicator of the project's commitment to testing practices. Based on the coverage within the audit's scope and my closer assessment of the test files, I deem the testing practices fitting for this stage of development.

4.4 Comments

As previously mentioned, the code is generally clean and strategically straightforward; even so, comments can further enhance the experience for both auditors and developers. In the context of this codebase, I believe comments are generally effective in delivering value to readers; however, there are a few sections that could benefit from additional comments.

I expect special attention to more detailed comments in mathematical and pricing operations like, such as those found in `InvestmentManager.convertToShares` and `InvestmentManager.converToAssets`. While these operations are not inherently complex, such comments are crucial because there are two phases of finding bugs in mathematical operations in a code. Usually, the easier one for most auditors is to spot a discrepancy between the intention the comments documenting the code transmit and the code itself. The other one is questioning the mathematical foundations and correctness of the formulas used. Being specific when commenting on the expected outcomes of calculations within the code greatly aids to address the former.

4.5 Solidity Versions

When evaluating the quality of a codebase, it is a sensible approach to examine the range of Solidity versions accepted throughout the codebase:

```
$ grep --include \*.sol --exclude-dir forge-std -hr "pragma soli
      1 pragma solidity ^0.8.20;
      1 // pragma solidity 0.8.21;
     47 pragma solidity 0.8.21;
```

The majority of our codebase currently relies on version `0.8.21`, a choice that I consider highly advantageous. And, using `^0.8.20` is obviously not a problem.

Whilst it is true that there are valid points both in favor of and against adopting the latest Solidity version, I believe this debate holds little relevance for the project at this stage. Opting for the most recent version is unquestionably a superior choice over potentially risky outdated versions.

5. Conclusion

Auditing this codebase and its architectural choices has been a delightful experience. Inherently complex systems greatly benefit from strategically implemented simplifications, and I believe this project has successfully struck a harmonious balance between the imperative for simplicity and the challenge of managing complexity. I hope that I have been able to offer a valuable overview of the methodology utilised during the audit of the contracts within scope, along with pertinent insights for the project team and any party interested in analysing this codebase.

Time spent:
18 hours

hieronx (Centrifuge) acknowledged

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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