Centrifuge
Security Review

Cantina Managed review by:
0xLeastwood, Lead Security Researcher
Sujith Somraaj, Associate Security Researcher

October 27, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of low severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or high. Critical findings should be directly vulnerable and have a high likelihood of being exploited. High findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

Centrifuge is the infrastructure that facilitates the decentralized financing of real-world assets natively on-chain, creating a fully transparent market which allows borrowers and lenders to transact without unnecessary intermediaries. The protocol aims to lower the cost of borrowing for businesses around the world, while providing DeFi users with a stable source of collateralized yield that is uncorrelated to the volatile crypto markets.

From Oct 5th to Oct 19th (10 days + 1 day extension) the Cantina team conducted a review of liquidity-pools on commit hash 836f03b3. The team identified a total of **20** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 5
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 11

3 Findings

3.1 Medium Risk

3.1.1 Weighted average prices are unable to handle currency and tranche tokens with varying decimals

Severity: Medium Risk

Context: InvestmentManager.sol#L349-L403

Description: When deposit/redeem orders are decreased, the price needs to be updated to reflect a 1:1 exchange rate of currency or tranche tokens so that they can be readily redeemed. However, if the decimals of the currency and tranche token amounts differ, then the price calculation will be incorrectly stored within `state.depositPrice` and `state.redeemPrice`.

```
function handleExecutedDecreaseInvestOrder(
    uint64 poolId,
    bytes16 trancheId,
    address user,
    uint128 currencyId,
    uint128 currencyPayout,
    uint128 remainingInvestOrder
) public onlyGateway {
    // ...
    // Calculating the price with both payouts as currencyPayout
    // leads to an effective redeem price of 1.0 and thus the user actually receiving
    // exactly currencyPayout on both deposit() and mint()
    state.redeemPrice = _calculatePrice(
        liquidityPool,
        state.maxWithdraw + currencyPayout,
        ((maxRedeem(liquidityPool, user)) + currencyPayout).toUint128()
    );
    // ...
}
```

As noted here, `_calculatePrice()` takes two amount arguments, `currencyAmount` and `trancheTokenAmount`. However, it assumes that both these amounts are according to their respective decimal precision. It is dangerous to assume that `((maxRedeem(liquidityPool, user)) + currencyPayout).toUint128()` will be adding two amounts of the same precision.

The same can be said for when `state.depositPrice` is calculated as the following.

```
function handleExecutedDecreaseRedeemOrder(
    uint64 poolId,
    bytes16 trancheId,
    address user,
    uint128 currencyId,
    uint128 trancheTokenPayout,
    uint128 remainingRedeemOrder
) public onlyGateway {
    // ...
    // Calculating the price with both payouts as trancheTokenPayout
    // leads to an effective redeem price of 1.0 and thus the user actually receiving
    // exactly trancheTokenPayout on both deposit() and mint()
    InvestmentState storage state = investments[liquidityPool][user];
    state.depositPrice = _calculatePrice(
        liquidityPool, _maxDeposit(liquidityPool, user) + trancheTokenPayout, state.maxMint + trancheTokenPayout
    );
    // ...
}
```

This breaks some integration with ERC4626 as `previewDeposit()` and `previewMint()` return inaccurate token amounts.

Recommendation: Modify the wighted average price calculation so that it takes into account the decimals of the tokens.

Centrifuge: Fixed in commit 411494ce.

Cantina: Verified fix. The `handleExecutedDecreaseInvestOrder()` and `handleExecutedDecreaseRedeemOrder()` functions now perform proper conversions when calculating the new redeem and deposit

prices.

3.1.2 Centrifuge only partially supports fee-on-transfer tokens

Severity: Medium Risk

Context: UserEscrow.sol#L28-L49

Description: There have been some changes made to the deposit flow with the intention of supporting fee-on-transfer tokens. While deposit requests record the correct `transferredAmount` when moving funds into the escrow contract and notifying the Centrifuge chain of the increased investment. Upon token redemption, tokens are transferred to a user escrow contract first before processing the redemption when the Centrifuge chain has successfully finalized the intent to exit out of a liquidity pool.

```
function requestDeposit(address liquidityPool, uint256 currencyAmount, address user) public auth {
    // ...

    // Transfer the currency amount from user to escrow (lock currency in escrow)
    // Checks actual balance difference to support fee-on-transfer tokens
    uint256 preBalance = ERC20Like(currency).balanceOf(address(escrow));
    SafeTransferLib.safeTransferFrom(currency, user, address(escrow), _currencyAmount);
    uint256 postBalance = ERC20Like(currency).balanceOf(address(escrow));
    uint128 transferredAmount = (postBalance - preBalance).toUint128();

    InvestmentState storage state = investments[liquidityPool][user];
    state.remainingDepositRequest = state.remainingDepositRequest + transferredAmount;
    state.exists = true;

    gateway.increaseInvestOrder(
        poolId, lPool.trancheId(), user, poolManager.currencyAddressToId(currency), transferredAmount
    );
}
```

The accounting in the user escrow contract records the full `currencyPayout` when handling the finalized redemption message. However, the actual amount that is meant to be recorded should be `currencyPayout` less the fee for transferring between the escrow and user escrow contracts. Ultimately, the fee will be paid out by the last users to withdraw from the user escrow contract.

Recommendation: Correct the accounting issue in `UserEscrow` or consider removing fee-on-transfer tokens altogether. The complexity that atypical ERC20 tokens introduce are exacerbated by multi-chain protocol environments. So the latter proposed change is preferred as the safest approach.

Centrifuge: Fixed in commit 69aa8b5b.

Cantina: Verified fix. Fee-on-transfer support has been removed entirely and tokens are transferred into the escrow contract from the liquidity pool instead.

3.1.3 Frozen and non-member accounts can always redeem tranche tokens

Severity: Medium Risk

Context: InvestmentManager.sol#L185-L221, ERC20.sol#L227-L240

Description: The `RestrictionManager` contract implements a simple restricted token standard (ERC-1404) for which accounts can be frozen and added/removed from a member list. This member list restricts who can be the recipient of token transfers. For example, deposit requests are entirely restricted to this member set as it is not possible to request or process a deposit unless the account is a valid member of the protocol. This is not true when requesting to redeem tranche tokens.

The `_processIncreaseRedeemRequest()` function calls `authTransferFrom()` which does not detect any transfer restrictions as outlined by the ERC-1404 token standard. Therefore, frozen and non-member accounts are able to redeem tranche tokens even when restricted. This is not compliant with regulatory authorities as it is no longer possible to blacklist users where necessary.

Recommendation: Either consider using an alternate implementation of `authTransferFrom()` or add a call to `detectTransferRestriction()` where the `from` and `to` accounts are both `user`.

Centrifuge: Fixed in commit cac2ea22.

Cantina: Verified fix. Freeze checks are now down on the `from` and `to` accounts. When requesting to deposit funds, the `sender` must now be part of the member list and non-frozen. Similarly, requests to redeem tranche tokens ensure that the token `owner` and `operator` accounts are non-frozen. But non-members can always redeem tranche tokens.

It is important to note that users who are not apart of the member list will not be able to process their deposit if they lose membership during an async deposit. To redeem these locked tranche tokens, the pool issuer would have to call `handleTriggerIncreaseRedeemOrder()`.

3.1.4 Tranche token deployments can be overwritten in `PoolManager`

Severity: Medium Risk

Context: PoolManager.sol#L231-L252, PoolManager.sol#L361

Description: It is important to note that the Centrifuge chain has complete autonomy over adding new tranche details, but anyone is able to deploy tranches as it adheres to the details stored in `undeployedTranches`.

However, there is no check when adding new tranche details that `poolId` and `trancheId` does not already belong to a deployed tranche. Therefore, if for whatever reason a new tranche is deployed, it will overwrite the existing `pools[poolId].tranches[trancheId].token` and completely break the redemption of the former tranche token. It is also possible to deploy liquidity pools under this new tranche token for which the state might not be recoverable by upgrading the protocol.

Recommendation: Add a check for `pools[poolId].tranches[trancheId].token == address(0)` in `addTranche()`.

Centrifuge: Addressed in commit a52fb6d0.

Cantina: Verified fix. The recommended check was added into the `addTranche()` function.

3.1.5 Anyone can front-run and deny deposit requests in `requestDepositWithPermit`

Severity: Medium Risk

Context: LiquidityPool.sol#L224-L230, LiquidityPool.sol#L329-L347

Description: The `requestDepositWithPermit` function serves the same purpose as the `requestDeposit` function, except for improved UX, permit is used to avoid a two-step approve and call action. However, because the `owner` and `spender` are arbitrary addresses, anyone can call permit and request to deposit a very small amount.

```
function _withPermit(
    address token,
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) internal {
    try ERC20PermitLike(token).permit(owner, spender, value, deadline, v, r, s) {
        return;
    } catch {
        if (IERC20(token).allowance(owner, spender) >= value) {
            return;
        }
    }
    revert("LiquidityPool/permit-failure");
}
```

As a result, the subsequent call will revert because the owner's token allowance is insufficient.

It's possible to also extend this if `WETH` is used as a pool's currency. Because the fallback function does not revert, the permit function will call `deposit()` with an empty `msg.value` and return back to the `requestDepositWithPermit()` function without performing any strict allowance check.

```
contract WETH9 {
    // ...
    function() public payable {
        deposit();
    }
    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        Deposit(msg.sender, msg.value);
    }
    // ...
}
```

This can be abused to deny all deposit requests.

Recommendation: Consider restricting `owner` to `msg.sender` so it is enforced that they are the only account who can request a deposit. Even if the permit call is front-run, `_withPermit` is able to handle failure and will only check that the allowance is sufficient.

Alternatively, the allowance check in `_withPermit` could be stricter to force a front-runner to create a request for the full approved amount. This should be added to both the `try` and `catch` branches.

Centrifuge: Fixed in commit 6af024f4.

Cantina: The `_withPermit()` function now enforces `IERC20(token).allowance(owner, spender) == value` such that if anyone front-runs the permit call, then the deposit request must use the full approved amount.. However, tokens like `WETH` do not implement a fallback function which reverts when a non-existent `permit()` call is made. Hence, it is necessary to perform a strict allowance check inside both `try` and `catch` branches.

Centrifuge: Addressed in commit 0c01bf4f.

Cantina: Verified fix. The `requestDepositWithPermit()` function now enforces `owner` to be `msg.sender`.

3.2 Low Risk

3.2.1 Asynchronous nature of cross-chain tranche token transfers may leave tokens stuck

Severity: Low Risk

Context: PoolManager.sol#L327-L335

Description: A user may be part of the member list when a transfer is initiated from Centrifuge chain. However, there is no guarantee that holds true after the message has been relayed and executed on Ethereum.

Therefore, `handleTransferTrancheTokens()` may revert because `trancheToken.mint(destinationAddress, amount)` checks that the `destinationAddress` is part of the member list when minting tokens. There is no way to recover these tokens except by re-adding the user back into the member list, which then allows them to generate requests to deposit and redeem tokens which is not always intended.

```
function handleTransferTrancheTokens(uint64 poolId, bytes16 trancheId, address destinationAddress, uint128
↪  amount)
    public
    onlyGateway
{
    TrancheTokenLike trancheToken = TrancheTokenLike(getTrancheToken(poolId, trancheId));
    require(address(trancheToken) != address(0), "PoolManager/unknown-token");

    trancheToken.mint(destinationAddress, amount);
}
```

Recommendation: It may be useful to remove any restriction checks when minting tokens as these should be done when creating a request to deposit or redeem funds instead.

Centrifuge: Acknowledged, but we won't be fixing this for now. The additional complexity of handling this properly is quite significant.

Cantina: Acknowledged.

3.2.2 Pool data cannot be relied upon off-chain or by external integrators

Severity: Low Risk

Context: LiquidityPool.sol#L322-L326, InvestmentManager.sol#L492-L498

Description: Axelar does not enforce message ordering because it opens up the message bridge to DoS attacks. As this order is not enforced, then the execution of these messages cannot be expected to be processed in order. There are several functions (`updateTrancheTokenPrice()`, `handleExecutedCollectInvest()`, `handleExecutedCollectRedeem()`) which update the liquidity pool's latest price.

This cannot in any way be relied upon by protocols integrating with Centrifuge nor can it be used as a source of truth off-chain.

It is also important to note that `state.remainingDepositRequest` and `state.remainingRedeemRequest` are overwritten each time a message is executed in the investment manager contract. As a result, `userDepositRequest()` and `userRedeemRequest()` are not accurate either.

Recommendation: Ensure this is well-documented to users and developers.

Centrifuge: Partial fix in commit 296bdf6a.

Cantina: `PoolManager.updateTrancheTokenPrice()` does not check that `computedAt` is increasing from the last price timestamp before overwriting.

Centrifuge: Fixed in commit 3f0e1022.

Cantina: Verified fix. Latest price data is only updated if it's `computedAt` timestamp is greater than the last price update's timestamp.

3.2.3 Tokens with callbacks can double count deposits

Severity: Low Risk

Context: InvestmentManager.sol#L169-L174

Description: It is unclear what tokens would be approved for use with Centrifuge's liquidity pools, but it's not unlikely to think that some of these tokens might have a callback enabled prior to the actual token transfer. In this case, it would be possible to re-use deposits and trick Centrifuge chain into minting additional tranche tokens to the user for which they could later redeem.

Consider the case where one deposit request is made and it is reentered via callback one more time:

- An initial deposit request of 100 is made.
- `preBalance` would be zero for the first transfer.
- A callback is made prior to tokens being transferred from the `user` account.
- Another deposit request of 100 is made and `preBalance` is still initially zero.
- The reentrancy is unwound and the reentrant function checks `postBalance` which is now 100, setting this as the transferred amount.
- The initial deposit is unwound and has a `postBalance` of 200 which is then passed onto the Centrifuge chain as the transferred amount.
- Even though the currency amount transferred was only 200, the protocol considered the user to have sent 300 tokens.

Recommendation: Add reentrancy protections to specific functions which transfer tokens from or to a user as they may be also relinquishing control over execution too.

Centrifuge: Fixed in commit 69aa8b5b.

Cantina: Verified fix. Pre and post balance checks have been removed so tokens with callbacks will be unable to trick the protocol into minting additional tranche tokens.

3.2.4 Deposits will be halted if the zero address is frozen

Severity: Low Risk

Context: RestrictionManager.sol#L73-L76

Description: Any wards on any token contract which inherits the restriction manager can freeze `address(0)`, blocking token mints until some other ward unfreezes the zero address and removed the malicious ward.

This impact is small in the context of `TrancheToken`, but in `InvestmentManager.handleExecutedCollectInvest()`, a frozen zero address will block all incoming messages from the centrifuge chain, briefly halting deposit operations until the issue is resolved.

Recommendation: Add checks to the `freeze()` function to not accidentally/maliciously freeze `address(0)`.

Centrifuge: Fixed in commit ac440b05.

Cantina: Verified fix.

3.3 Informational

3.3.1 Use Axelar's gas service

Severity: Informational

Context: Router.sol#L88

Description: Axelar has a two-step approach for delivering messages to the destination chain. The first is to call the `callContract()` method of Axelar's gateway followed by the payment call to Axelar's gas service through the `payNativeGasForContractCall()` method.

In Centrifuge, the executor service of Axelar is not called in the same atomic transaction. Instead, the user or an external actor should call the function to deliver the message, which can lead to unexpected delays and affect the robustness of Centrifuge's cross-chain communication layer.

Recommendation: Use Axelar's gas service to deliver the message and invoke the `payNativeGasForContractCall()` method in the same atomic transaction inside Axelar Router's `send()` method. Also, run an off-chain watcher to ensure the gas paid is enough to process the transaction, to avoid intentional delaying of the message by the user.

Centrifuge: As discussed, this is intentional. Users can always use multicall to still complete both steps in 1 atomic transaction.

Cantina: Acknowledged.

3.3.2 Authorized admins of user escrow may grief the protocol

Severity: Informational

Context: UserEscrow.sol#L35-L49

Description: While `transferOut()` has made some considerations for a compromised authorized admin, it does not fully protect users from being impacted.

The current implementation will prevent the receiver account from changing, unless they have been given the full allowance by the original user. However, it is also likely that the liquidity pool contracts have full token approvals, or really any contract for that matter. Consequently, a compromised auth admin could send tokens to any address which has a full approval.

Recommendation: Consider documenting this to users and restricting changes to the recipient of tranche token redemptions to be EOAs only. It is noted that the latter change could potentially break certain smart contract integrations.

Centrifuge: Interesting scenario! IMO The only thing we can do here is properly documenting that users should always be very careful when giving full approval to protocols.

I would propose to mark the issue as informational, as it includes an external unrealistic factor where the admin is compromised. In addition, it is rather unlikely that users have full currency approvals on

protocols where no amount has been already deducted from the max approval. Approval has to be set to max when the malicious call is executed. For example: If a user already invested into a `LiquidityPool` the approval is not max anymore as the invested value got deducted from the approval on transfer.

Cantina: Acknowledged.

3.3.3 Non-existent users check in `InvestmentManager`

Severity: Informational

Context: InvestmentManager.sol#L234, InvestmentManager.sol#L265, InvestmentManager.sol#L223, InvestmentManager.sol#L255

Description: The functions `decreaseDepositRequest()`, `cancelDepositRequest()`, `decreaseRedeemRequest()` and `cancelRedeemRequest()` could flood the network on the Centrifuge chain with invalid requests from the EVM chain due to a lack of input validations.

Since users can call these functions only after a successful `requestRedeem()` or `requestDeposit()` call, adding validations to these functions would be more appropriate.

Recommendation: Consider adding checks to make sure `state.exists == true` for the user calling the affected functions. Also add checks to make sure `state.remainingRedeemRequest > 0` for redemptions and `state.remainingInvestOrder > 0` for deposit related functions.

Centrifuge: `state.exists` is just used to ensure no currency is accidentally sent to the user escrow or an invalid account. Adding the methods to decrease/cancel methods would not add any protections because a user can always set their account into a valid `state.exists` by depositing `1 wei`.

Adding a check that `state.remainingRedeemRequest > 0` unfortunately does not work either as these values cannot be relied upon, so there could be a scenario where `Executed*` messages were swapped leading to an invalid `remaining` value, and this is then blocking the user from decreasing/canceling their request. Considering this as a won't-fix as it is by design.

Cantina: Acknowledged.

3.3.4 Add zero value checks in `handleTriggerIncreaseRedeemOrder`

Severity: Informational

Context: InvestmentManager.sol#L410

Description: If `trancheTokenAmount` is zero, then `handleTriggerIncreaseRedeemOrder()` is executed and will modify the `state.exists` boolean for an invalid redeem request.

Recommendation: Add explicit reverts if `trancheTokenAmount` is zero, in the same way as done in `requestRedeem()`.

Centrifuge: Fixed in commit 899a3048.

Cantina: Verified fix.

3.3.5 Increase inline documentation

Severity: Informational

Context: src/*

Description: The repository has limited comments (inline documentation) within the code, and critical functions lack adequate explanations. This makes understanding the protocol's operations difficult when reviewing the code.

Recommendation: Ensure every function within the protocol has accompanying comments. This will provide clear documentation throughout the codebase.

Centrifuge: This is more about coding style than an actual issue. Other protocols such as MakerDAO also adopt a similar style where comments are limited to 1) external functions (in our case, mainly the liquidity pool contract), or 2) code that is not understandable on its own. I believe we comment where needed, and adding additional comments would clutter the code more.

Cantina: Acknowledged.

3.3.6 `LiquidityPool` does not conform to `ERC4626`

Severity: Informational

Context: LiquidityPool.sol

Description: `LiquidityPool.sol` tries to conform to ERC-4626, a vault standard that makes the vault interface optimized for integrators. However, the current `LiquidityPool` contract behaves in a way that is significantly different from a typical ERC-4626 implementation.

i.e. `previewDeposit()` works on the user's average `state.depositPrice`, but the `deposit()` function will always fail since the user should call `requestDeposit()` before calling the actual `deposit()` function.

This effectively leads to the `previewDeposit()` function returning unrelated values even when the `deposit()` function will revert for all possible combinations of amount inputs.

Similar effects persists on `mint()` and `redeem()` functions as well. This is a risk for integrators as their protocol cannot make any ERC-4626 based assumptions.

Recommendation: Add clear directions to integrators that the vault is not ERC-4626 compliant but just use similar function signatures. Try to make sure the preview functions work as a normal 4626 vault and revert in the following execution function calls [mint, deposit, redeem].

Centrifuge: Fixed in commit bda5ba6e.

Cantina: Verified fix. Preview methods have been removed to be compliant with a new async vault token standard EIP-7540.

3.3.7 Inconsistent use of `state.exists`

Severity: Informational

Context: InvestmentManager.sol#L288, InvestmentManager.sol#L382

Description: The `state.exists` boolean is set to `true` during `requestDeposit()`, `requestRedeem()`, and `handleTriggerIncreaseRedeemOrder()` function calls, which acts as the only source of truth to validate the callback from the Centrifuge chain through Axelar.

However, the flag is not consistently used across all handle functions. For example, `handleExecutedCollectInvest()` and `handleExecutedDecreaseRedeemOrder()` do not check if a user exists already.

Also, there is no way to reset the flag in certain edge-case scenarios. i.e. the user calls `requestDeposit()`, which sets `exists = true` and quickly calls `cancelInvestOrder()` subsequently. As a result, their investment is cancelled, however, they never fully interacted with the liquidity pool, but the boolean flag is still set to `true`.

This expands the trust on the cross-chain message delivered by Axelar enormously and could allow it to enter any callback function without an actual transaction request from the EVM chain. This issue is unlikely to ever happen since Axelar validators are not incentivized to carry out this attack.

Recommendation: Consider checking the `state.exists` boolean value where necessary. It may be worthwhile validating this value before processing Axelar message to reduce damage to certain extent. There are other areas where the protocol trusts the message bridge.

Centrifuge: The Centrifuge team accepts this as expected protocol behaviour.

Cantina: Acknowledged.

3.3.8 Colluded message ordering could result in undesirable state

Severity: Informational

Context: Gateway.sol#L378

Description: The entire security of `Auth` in `Root.sol` is heavily dependent on a spell pattern from MakerDAO. There are two ways to add a new ward to root, one is through `DelayedAdmin.sol` and another way is through `Gateway.sol`. The `Gateway` contract depends on the cross-chain message bridge Axelar to schedule and cancel newly added adapters.

Imagine `scheduleRely()` is triggered from the Centrifuge chain, which then was quickly canceled on the remote chain by calling `cancelRely()`. But when these transactions are relayed, the Axelar bridge is down/broken.

After a while, those two transactions arrive at the EVM chain and the relayers process `cancelRely()` first and later the `scheduleRely()` function. This negates the `cancelRely()` functionality, and if unnoticed, this will add a ward that is canceled on the Centrifuge chain.

Note: Axelar provides no order of message ordering/protection and the message can be executed only once.

Recommendation: Run off-chain watchers, that detect the message ordering impact, who can then initiate a transaction through `DelayedAdmin.sol` and cancel the scheduled transaction. Add an explicit revert if there are no ward additions scheduled.

Centrifuge: Fixed in commit f945184c.

Cantina: Verified fix. `cancelRely()` will revert if a new ward has not been scheduled.

3.3.9 Lack of `MIN_DELAY` in root could invalidate timelock

Severity: Informational

Context: Root.sol#L48

Description: A ward on `Root.sol` can set the delay to 0 and bypass the Timelock assumptions. This scenario is highly unlikely since the only wards on the `Root` contract initially are `DelayedAdmin` and `Gateway` which have no functionality to set the delay at this point.

But this could change in the future with the addition of new wards that are capable of updating delay. This issue can cause serious damage to the protocol but is not possible with the current setup. Exercise caution while adding new wards that can update delay values.

Recommendation: Either make `delay` immutable during the initial deployment or make `delay` bounded between `MIN_DELAY` and `MAX_DELAY`. Refer to Openzeppelin's blog on Timelock for more information on why a minimum delay is required.

Centrifuge: The Centrifuge team accepts this as expected protocol behaviour.

Cantina: Acknowledged.

3.3.10 Unrecoverable states through misconfiguration

Severity: Informational

Context: Auth.sol#L20, DelayedAdmin.sol#L34

Description: An existing ward on `DelayedAdmin.sol` can remove all other wards in this contract, `pause()` the protocol, and renounce itself from `DelayedAdmin.sol` accidentally or maliciously.

Since `DelayedAdmin` and `PauseAdmin` are the only initial admins in the `Root` contract, and only the delayed admin can `unpause()` the protocol or add new wards, this could lead the protocol into an irrecoverable state where critical functionality like pausing and the addition of new wards will be permanently blocked.

Recommendation: Add checks to `DelayedAdmin` to make sure there is at least one active ward available on them and is added/removed through a timelock pattern for robust security.

Centrifuge: Adding additional checks here would lead to a false sense of security. Considering as won't-fix because this is by design.

Cantina: Acknowledged.

3.3.11 Increase test coverage

Severity: Informational

Context: src/*

Description: There is less than complete test coverage of key contracts under review including `InvestmentManager.sol`, `LiquidityPool.sol` and `PoolManager.sol`. Adequate test coverage (both line and branches) is an essential process in ensuring the codebase works as expected. Insufficient code coverage can lead to unexpected issues.

Recommendation: Add to test coverage ensuring all the branches and execution paths are covered. Could also refer Paul R Berg's BTT thread and talk.

Centrifuge: Acknowledged, and will be addressed in the upcoming weeks!

Cantina: Acknowledged.