Adapter Finance Security Review

Pashov Audit Group

Conducted by: Said, Dan Ogurtsov, ast3ros

May 13th 2024 - May 19th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **adapter-fi/AdapterVault** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Adapter Finance

Adapter Finance allows creation of flexible ERC4626 vaults that use registered adapters to integrate with protocols, automating and optimizing liquidity provision for yields. Strategies dictate token distribution via registered adapters using a vault-specific funds-allocator smart contract.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c410c4d7eccb614de864e19faaa271ef0886ae36

fixes review commit hash - 0e826db27b9a99b1903a4a5490323a9e12648d69

Scope

The following smart contracts were in scope of the audit:

- `AdapterVault`
- `FundsAllocator`
- `Governance`
- `PendleAdapter`
- `IAdapter`

7. Executive Summary

Over the course of the security review, Said, Dan Ogurtsov, ast3ros engaged with Adapter Finance to review Adapter Finance. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Adapter Finance
Repository	https://github.com/adapter-fi/AdapterVault
Date	May 13th 2024 - May 19th 2024
Protocol Type	Yield strategies

Findings Count

Severity	**Amount**
Medium	9
Low	3
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Multiple strategies sharing the same asset	Medium	Resolved
[M-02]	submitStrategy() DOS	Medium	Resolved
[M-03]	replaceGovernanceContract is not reset	Medium	Resolved
[M-04]	Send 0 fees to the previous proposer when the proposer is changed	Medium	Resolved
[M-05]	set_strategy will incorrectly remove last_asset_value	Medium	Resolved
[M-06]	min_transfer_balance is checked against the wrong value	Medium	Resolved
[M-07]	PendleAdapter provides zero slippage	Medium	Resolved
[M-08]	Race condition based on TDelay condition	Medium	Resolved
[M-09]	Race condition based on number of guards	Medium	Resolved
[L-01]	Changing guards during a pending voting	Low	Resolved
[L-02]	_getTargetBalancesWithdrawOnly incorrect calculation	Low	Resolved
[L-03]	target_withdraw_balance is not reduced	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Multiple strategies sharing the same asset

Severity

Impact: High

Likelihood: Low

Description

AdapterVault design assumes that each strategy has a unique asset (e.g. PT token). Otherwise, key accounting functions would return overinflated values.

E.g. _totalAssetsNoCache() which goes through each strategy and calls `balanceOf()` for their related underlying and sums them up.

```
@internal
@view
def _totalAssetsNoCache() -> uint256:
    assetqty : uint256 = ERC20(asset).balanceOf(self)
    for adapter in self.adapters:
        assetqty += IAdapter(adapter).totalAssets()

    return assetqty
```

`IAdapter(adapter).totalAssets()` checks the balance of the related underlying on `AdapterVault` and multiply by its oracle price (according to `PendleAdapter`).

But if two strategies share the same underlying, it will double count the same token twice, because `AdapterVault` cannot identify how this balance is split between two strategies.

As a result, it leads to overinflated `totalAssets()` and broken calculations between shares and assets. Transient cache values will also hold the wrong balances.

Recommendations

_addAdapter() should check that `adapter.asset` is not used in current strategies.

[M-02] `submitStrategy()` DOS

Severity

Impact: Medium

Likelihood: Medium

Description

`Governance.submitStrategy()` is strict to have only one pending strategy per vault. Moreover, no new strategy can be accepted for voting. It can only be accepted either after accepting or rejecting a pending strategy or after 6 hours after submission.

In addition, this function is public, so anyone can submit.

As a result, the simplest attack vector is spamming any new strategy as soon as submission is available. New submissions will have to wait 6 hours or governance step in. But after that the same spam attack is possible. It can also be frontrun transactions before submissions to block them.

Recommendations

Consider allowing multiple pending strategies so that the attacker's submissions could be just ignored. Or allow submitting only for trusted addresses.

[M-03] replaceGovernanceContract is not reset

Severity

Impact: Medium

Likelihood: Medium

Description

After the governance is replaced, the VotesGCByVault is not reset to 0. This can lead to a situation where if the governance for the vault is reverted back to a previous contract, the vote count is still recorded, allowing a guard to change it back without proper voting. For example, consider governance contracts A and B: A is voted to be replaced by B. If B has a problem or bug and is replaced back by A, the vote count for B is still recorded. A guard can then vote to change it back to B immediately without proper voting.

```
@external
def replaceGovernance(NewGovernance: address, vault: address):
    ...
    #Add Vote to VoteCount
    for guard_addr in self.LGov:
        if self.VotesGCByVault[vault][guard_addr] == NewGovernance:
            VoteCount += 1

    if len(self.LGov) == VoteCount:
        AdapterVault(vault).replaceGovernanceContract(NewGovernance)
```

Recommendations

Reset `VotesGCByVault` to 0.

[M-04] Send 0 fees to the previous proposer when the proposer is changed

Severity

Impact: Medium

Likelihood: Medium

Description

When `set_strategy` is called and `current_proposer` is not the same as the provided `_proposer`, it will try to send fees to the previous proposer by calling _claim_fees.

```
@internal
def _set_strategy(
  _proposer:address,
  _strategies:AdapterStrategy[MAX_ADAPTERS],
  _min_proposer_payout:uint256,
  pregen_info:DynArray[Bytes[4096],
  MAX_ADAPTERS]
) -> bool:

         assert msg.sender == self.governance, "Only Governance DAO may set a new stra
    assert _proposer != empty(address), "Proposer can't be null address."

    # Are we replacing the old proposer?
    if self.current_proposer != _proposer:

        current_assets : uint256 = self._totalAssetsNoCache
          () # self._totalAssetsCached()

        # Is there enough payout to actually do a transaction?
        yield_fees : uint256 = 0
        strat_fees : uint256 = 0
        yield_fees, strat_fees = self._claimable_fees_available(current_assets)
        # @audit - should be equal
        if strat_fees > self.min_proposer_payout:

            # Pay prior proposer his earned fees.
>>>         self._claim_fees(FeeType.PROPOSER, 0, pregen_info, current_assets)

        self.current_proposer = _proposer
        self.min_proposer_payout = _min_proposer_payout

    # .....

    return True
```

It can be observed that it provides 0 as `_asset_amount`, which will incorrectly be set to `fees_to_claim` if `current_vault_assets` is enough to cover the `min_fees_to_claim`.

```
@internal
def _claim_fees(
  _yield:FeeType,
  _asset_amount:uint256,
  pregen_info:DynArray[Bytes[4096],
  MAX_ADAPTERS],
  _current_assets:uint256=0,
  _min_assets:uint256=0
) -> uint256:
    # self._claim_fees(FeeType.PROPOSER, 0, pregen_info, current_assets)
    yield_fees : uint256 = 0
    strat_fees : uint256 = 0

    yield_fees, strat_fees = self._claimable_fees_by_me
      (_yield, _asset_amount, _current_assets)

    fees_to_claim : uint256 = yield_fees + strat_fees
    if _asset_amount > 0:                 # Otherwise we take it all.

                 fees_to_claim = _asset_amount   # This will be lower than or equal to

    # Account for slippage minimums.
    min_fees_to_claim : uint256 = self._defaultSlippage
      (fees_to_claim, _min_assets)

    # Do we have enough balance locally to satisfy the claim?
    current_vault_assets : uint256 = ERC20(asset).balanceOf(self)

    if current_vault_assets < min_fees_to_claim:

                 # Need to liquidate some shares to fulfill. Insist on withdraw only s

                 # Note - there is a chance that balance adapters could return more th
        #        don't just give it all away in case there's an overage.
        fees_to_claim = min(self._balanceAdapters
          (fees_to_claim, pregen_info, True), fees_to_claim)

                 assert fees_to_claim >= min_fees_to_claim, "Couldn't get adequate ass
    else:
        # @audit - if asset_amount is 0, it will set fees_to_claim to 0
>>>     fees_to_claim = min(_asset_amount, current_vault_assets)
    # Adjust fees proportionally to account for slippage.
    if strat_fees > 0 and yield_fees > 0:
        strat_fees = convert((convert(strat_fees, decimal)/convert(

        )/convert
          (strat_fees+yield_fees, decimal
        yield_fees = fees_to_claim - strat_fees
    elif strat_fees > 0:
        strat_fees = fees_to_claim
    else:
        yield_fees = fees_to_claim

    # .....

    return fees_to_claim
```

This will cause the previous proposer to not receive the deserved fees.

Recommendations

Instead of checking min value using `_asset_amount`, use `fees_to_claim` instead.

```
# .....
    if current_vault_assets < min_fees_to_claim:

                 # Need to liquidate some shares to fulfill. Insist on withdraw only s

                 # Note - there is a chance that balance adapters could return more th
        #        don't just give it all away in case there's an overage.
        fees_to_claim = min(self._balanceAdapters
          (fees_to_claim, pregen_info, True), fees_to_claim)

                 assert fees_to_claim >= min_fees_to_claim, "Couldn't get adequate ass
    else:
-        fees_to_claim = min(_asset_amount, current_vault_assets)
+        fees_to_claim = min(fees_to_claim, current_vault_assets)
# .....
```

[M-05] `set_strategy` will incorrectly remove `last_asset_value`

Severity

Impact: Medium

Likelihood: Medium

Description

`set_strategy` can be called by governance to change the list of an adapter's strategies. This could incorporate a new adapter or change the previous adapter's ratio. However, it will incorrectly remove the `last_asset_value` of the adapter's strategy if the new strategy incorporates a previously used adapter.

```
def _set_strategy(
  _proposer:address,
  _strategies:AdapterStrategy[MAX_ADAPTERS],
  _min_proposer_payout:uint256,
  pregen_info:DynArray[Bytes[4096],
  MAX_ADAPTERS]
) -> bool:

         assert msg.sender == self.governance, "Only Governance DAO may set a new stra
    assert _proposer != empty(address), "Proposer can't be null address."

    # Are we replacing the old proposer?
    if self.current_proposer != _proposer:

        current_assets : uint256 = self._totalAssetsNoCache
          () # self._totalAssetsCached()

        # Is there enough payout to actually do a transaction?
        yield_fees : uint256 = 0
        strat_fees : uint256 = 0
        yield_fees, strat_fees = self._claimable_fees_available(current_assets)
        if strat_fees > self.min_proposer_payout:

            # Pay prior proposer his earned fees.
            self._claim_fees(FeeType.PROPOSER, 0, pregen_info, current_assets)

        self.current_proposer = _proposer
        self.min_proposer_payout = _min_proposer_payout

    # Clear out all existing ratio allocations.
    for adapter in self.adapters:
        self.strategy[adapter] = empty(AdapterValue)

    # Now set strategies according to the new plan.
    for strategy in _strategies:
        plan : AdapterValue = empty(AdapterValue)
        plan.ratio = strategy.ratio
        # @audit - it will clear `last_asset_value` if it previously exist
>>>     self.strategy[strategy.adapter] = plan

    log StrategyActivation(_strategies, _proposer)

    return True
```

`last_asset_value` is a safeguard/protection implemented in case of sudden loss or if the LP is compromised. Setting the value to 0 will remove and bypass the protection once rebalancing happens.

Recommendations

Set `last_asset_value` to the previous value if it previously exists.

[M-06] `min_transfer_balance` is checked against the wrong value

Severity

Impact: Medium

Likelihood: Medium

Description

When `_withdraw` is called a `transfer_balance` is greater than `current_balance` inside the vault, it should check if `current_balance` is enough to cover the `min_transfer_balance`. However, currently, it will check it against `transfer_balance` instead.

```
@internal
def _withdraw(
  _asset_amount:uint256,
  _receiver:address,
  _owner:address,
  pregen_info:DynArray[Bytes[4096],
  MAX_ADAPTERS],
  _min_assets:uint256=0
) -> uint256:
    # _withdraw(assetqty, _receiver, _owner, pregen_info, 0)
    min_transfer_balance : uint256 = self._defaultSlippage
      (_asset_amount, _min_assets)

    # ...


         # Make sure we have enough assets to send to _receiver. Do a withdraw only ba
    self._balanceAdapters(_asset_amount, pregen_info, True )

    # Now account for possible slippage.
    current_balance : uint256 = ERC20(asset).balanceOf(self)
    transfer_balance : uint256 = _asset_amount
    if transfer_balance > current_balance:
        # Didn't get as much as we expected. Is it above the minimum?
        # @audit - this should check against current_balance instead
>>>
       assert transfer_balance >= min_transfer_balance, "ERROR - Unable to meet minimu

        # We'll transfer what we received.
        transfer_balance = current_balance
        log SlippageWithdraw(
          msg.sender,
          _receiver,
          _owner,
          _asset_amount,
          shares,
          transfer_balance
        )

    # Now send assets to _receiver.
    ERC20(asset).transfer(_receiver, transfer_balance)

    # Clear the asset cache for vault but not adapters.
    self._dirtyAssetCache(True, False)

    # Update all-time assets withdrawn for yield tracking.
    self.total_assets_withdrawn += transfer_balance

    log Withdraw(msg.sender, _receiver, _owner, transfer_balance, shares)

    return shares
```

This will cause the call to always pass the check and incorrectly set `transfer_balance` with `current_balance`, even when it doesn't exceed the minimum calculated slippage amount.

Recommendations

Check `min_transfer_balance` against `current_balance` instead.

```diff
if transfer_balance > current_balance:
        # Didn't get as much as we expected. Is it above the minimum?
        # @audit - this should check against current_balance instead
-
-         assert transfer_balance >= min_transfer_balance, "ERROR - Unable to meet min
+
+         assert current_balance >= min_transfer_balance, "ERROR - Unable to meet mini
        # We'll transfer what we received.
        transfer_balance = current_balance
        log SlippageWithdraw(
          msg.sender,
          _receiver,
          _owner,
          _asset_amount,
          shares,
          transfer_balance
        )
```

[M-07] `PendleAdapter` provides zero slippage

Severity

Impact: High

Likelihood: Low

Description

When the vault triggers `_adapter_deposit` and delegates the call to `PendleAdapter.deposit`, it will deposit the provided `_asset_amount` into the configured Pendle market.

```
@external
@nonpayable
def deposit(asset_amount: uint256, pregen_info: Bytes[4096]=empty(Bytes[4096])):
    # ....

    #mint if minting price is better, then sell the YT.
    if pg.mint_returns > pg.spot_returns:
        #Mint PY
        inp: TokenInput = empty(TokenInput)
        inp.tokenIn = asset
        inp.netTokenIn = asset_amount
        inp.tokenMintSy = asset

        netPyOut: uint256 = 0
        netSyInterm: uint256 = 0
        ERC20(asset).approve(pendleRouter, asset_amount)
        #Mint PY+PT using asset
        netPyOut, netSyInterm = PendleRouter(pendleRouter).mintPyFromToken(
            self,
            yt_token,
>>>         0,
            inp
        )

        #Swap any YT gained to PT
        ERC20(yt_token).approve(pendleRouter, netPyOut)

        PendleRouter(pendleRouter).swapExactYtForPt(
            self,
            pendleMarket,
            netPyOut,
>>>         0,
            pg.approx_params_swapExactYtForPt
        )

    else:
        #swapExactTokenForPt
        inp: TokenInput = empty(TokenInput)
        inp.tokenIn = asset
        inp.netTokenIn = asset_amount
        inp.tokenMintSy = asset

        limit: LimitOrderData = empty(LimitOrderData)
        ERC20(asset).approve(pendleRouter, asset_amount)
        PendleRouter(pendleRouter).swapExactTokenForPt(
            self,
            pendleMarket,
>>>         0,
            pg.approx_params_swapExactTokenForPt,
            inp,
            limit
        )
        #NOTE: Not doing any checks and balances, minPtOut=0 is intentional.
        #It's up to the vault to revert if it does not like what it sees.
```

It can be observed that the minimum out return value from all operations is set to 0, and it is noted that not providing slippage is intentional and will be left to the vault to check the slippage. However, inside `_balanceAdapters`, after `_adapter_deposit` is called, there is no slippage check to ensure that the minted/swapped token amount is within the expected range. A sandwich attack can be executed when `_balanceAdapters` is called to extract value from the operation.

Recommendations

Consider adding a minimum accepted return value inside `_balanceAdapters` and `_adapter_deposit` inside the vault.

[M-08] Race condition based on `TDelay` condition

Severity

Impact: Medium

Likelihood: Medium

Description

When a strategy is not rejected or endorsed by the guards, it can be activated after the `TDelay` period.

```
@external
def activateStrategy(Nonce: uint256, vault: address):
    ...
    #Confirm strategy is approved by guards
    assert (len(pending_strat.VotesEndorse) >= (len(self.LGov)/2)+1) or \
           ((


           ) < block.timestamp
    ...
```

However, a strategy can also be replaced by a new one after the `TDelay` period. This creates a situation where a strategy can be both activated and replaced depending on which transaction is executed first. A malicious proposer can front-run the activation transaction and submit a new strategy to deny the current one.

```
@external
def submitStrategy(strategy: ProposedStrategy, vault: address) -> uint256:
    ...
            # First is it the same as the current one?
            # Otherwise has it been withdrawn?
            # Otherwise, has it been short circuited down voted?

                         # Has the period of protection from being replaced expired al
    assert
      (self.CurrentStrategyByVault[vault].Nonce == pending_strat.Nonce) or \
            (pending_strat.Withdrawn == True) or \
            len(pending_strat.VotesReject) > 0 and \
            (len(pending_strat.VotesReject) >= pending_strat.no_guards/2) or \
            (convert(block.timestamp, decimal) > (convert(
              convert
            ) > (convert
              (pending_strat.TSubmitted, decimal
    ...
```

Recommendations

Ensure that the activation and replacement times do not overlap. A strategy should only be replaceable after a period longer than the `TDelay` period. Consider introducing an additional delay period (replacement delay) specifically for replacement to prevent front-running and race conditions.

[M-09] Race condition based on number of guards

Severity

Impact: High

Likelihood: Low

Description

A submitted strategy can be replaced by a new one if it has been short-circuited by downvotes. However, because the number of downvotes required to short-circuit is `no_guards / 2` and rounding down, the number of downvotes required can be less than half of `no_guards`. This can lead to a situation where a strategy is replaced even though it has not been properly short-circuited by downvotes.

Consider the scenario when `no_guards` equals 3:

- 2 guards `endorseStrategy` the strategy. This should be sufficient to activate the strategy (2/3 endorsements).
- Later, 1 guard `rejectStrategy` the strategy and can replace the current pending strategy with a new one (because `no_guards` / 2 is rounded down to 1).

```
def submitStrategy(strategy: ProposedStrategy, vault: address) -> uint256:
    ...


         # Confirm there's no currently pending strategy for this vault so we can repl

            # First is it the same as the current one?
            # Otherwise has it been withdrawn?
            # Otherwise, has it been short circuited down voted?

                         # Has the period of protection from being replaced expired al
    assert
      (self.CurrentStrategyByVault[vault].Nonce == pending_strat.Nonce) or \
            (pending_strat.Withdrawn == True) or \
            len(pending_strat.VotesReject) > 0 and \
            (len
              (pending_strat.VotesReject) >= pending_strat.no_guards/2) or \ # @audit
            (convert(block.timestamp, decimal) > (convert(
              convert
            ) > (convert
              (pending_strat.TSubmitted, decimal
    ...
```

Recommendations

Round up the required number of downvotes to short-circuit the strategy.

```
def submitStrategy(strategy: ProposedStrategy, vault: address) -> uint256:
    ...


         # Confirm there's no currently pending strategy for this vault so we can repl

            # First is it the same as the current one?
            # Otherwise has it been withdrawn?
            # Otherwise, has it been short circuited down voted?

                         # Has the period of protection from being replaced expired al
    assert
      (self.CurrentStrategyByVault[vault].Nonce == pending_strat.Nonce) or \
            (pending_strat.Withdrawn == True) or \
            len(pending_strat.VotesReject) > 0 and \
-           (len(pending_strat.VotesReject) >= pending_strat.no_guards/2) or \
+           (len(pending_strat.VotesReject) >= pending_strat.no_guards/2+1) or \
            (convert(block.timestamp, decimal) > (convert(
              convert
            ) > (convert
              (pending_strat.TSubmitted, decimal
    ...
```

8.2. Low Findings

[L-01] Changing guards during a pending voting

There are two functions that make it possible to remove a guard:

- `swapGuard()`
- `removeGuard()`

But the problem is that the removed guard might have already voted for or against a pending strategy, thus keep presenting in `PendingStrategyByVault[vault].VotesEndorse` `PendingStrategyByVault[vault].VotesReject`.

In this case, `len(pending_strat.VotesEndorse)` and `len(pending_strat.VotesReject)` are incorrect and can include non-gov addresses.

E.g. Consider the case when some guard Alice `swapGuard()` from the old address to the new one given that Alice has already voted for the old address. Now she can vote the second time, and `activateStrategy()` would count two votes.

Consider removing a deleted guard address from `PendingStrategyByVault[vault].VotesEndorse` and `PendingStrategyByVault[vault].VotesReject` during `swapGuard()` and `removeGuard()`.

One of the options could be avoiding `len(pending_strat.VotesEndorse)` and `len(pending_strat.VotesReject)` and instead using a separate calculation that loops through only existing `LGov` addresses and sums them up, so as to ignore addresses that are no longer in `LGov`.

[L-02] `_getTargetBalancesWithdrawOnly` incorrect calculation

In `_getTargetBalancesWithdrawOnly`, it will iterate through the adapters and withdraw from them if necessary to fulfill the target withdrawal balance. However, it will incorrectly add `adapter_assets_allocated` with `adapter.delta * -1` when `adapter.delta` is not 0.

```
@internal
@pure
def _getTargetBalancesWithdrawOnly(
  _vault_balance:uint256,
  _d4626_asset_target:uint256,
  _total_assets:uint256,
  _total_ratios:uint256,
  _adapter_balances:BalanceAdapter[MAX_ADAPTERS]
) -> (uint256, int256, uint256, BalanceAdapter[MAX_ADAPTERS], address[MAX_ADAPTERS]
    # self._getTargetBalancesWithdrawOnly(
      _vault_balance,
      _d4626_asset_target,
      _total_assets,
      _total_ratios,
      _adapter_balances
    )
    # ....

        if adapter.delta != 0:
>>>         adapter_assets_allocated += convert(
  adapter.delta*-1,
  uint256
)    # TODO : eliminate adapter_assets_allocated if never used.
            d4626_delta += adapter.delta * -1
            adapters[tx_count] = adapter
            tx_count += 1

        # NEW
>>>     adapter_result : int256 = convert
  (adapter.current, int256) + adapter.delta

                assert adapter_result >= 0, "Adapter resulting balance can't be less
>>>     adapter_assets_allocated += convert(adapter_result, uint256)


        assert target_withdraw_balance == 0, "ERROR - Unable to fulfill this withdraw


        return adapter_assets_allocated, d4626_delta, tx_count, adapters, blocked_ada
```

It can be observed that adapter `_assets_allocated` should be updated with the adapter's balance, which is added correctly at the end of the function. Consider to remove the unnecessary `adapter_assets_allocated += convert(adapter.delta * -1, uint256)` operation.

[L-03] `target_withdraw_balance` is not reduced

When calculating the target balances for withdrawing only, if the adapter ratio is 0, all the assets in the adapter will be withdrawn back to the vault. However, the withdrawn amount does not reduce the `target_withdraw_balance`. This can lead to over-withdrawal of assets when assets from other adapters also have to be withdrawn, incurring slippage and additional rebalancing costs.

```
def _getTargetBalancesWithdrawOnly(
  _vault_balance:uint256,
  _d4626_asset_target:uint256,
  _total_assets:uint256,
  _total_ratios:uint256,
  _adapter_balances:BalanceAdapter[MAX_ADAPTERS]
) -> (uint256, int256, uint256, BalanceAdapter[MAX_ADAPTERS], address[MAX_ADAPTERS]
    ...
        if adapter.ratio == 0 and adapter.current > 0:
            adapter.target = 0
            adapter.delta = max(convert(
              convert

            )*-1, adapter.max_withdraw
    ...
```

Reduce the `target_withdraw_balance` when withdrawing from an adapter with ratio 0 and a current balance greater than 0. This will ensure accurate calculations of withdrawal targets and prevent over-withdrawal from other adapters.