yAudit Dopex rDPX v2 Review

Review Resources:

- Dopex docs
- rDPX v2 spec

Auditors:

- engn33r
- securerodd

Table of Contents

Review Summary

Dopex rDPX v2

Dopex rDPX v2 provides a new synthetic ETH token. The protocol includes a bonding process where the user receives a receipt token that can later be redeemed for various underlying tokens.

The contracts of the Dopex rDPX v2 Repo were reviewed over 9 days. The code review was performed by 2 auditors between October 24 and November 2, 2023. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit 99372bc2e052ca9a5b7147ff83872c261cd5fba2 for the Dopex rDPX v2 repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- amo/UniV3LiquidityAmo.sol
- receiptToken/ReceiptToken.sol
- receiptToken/RedeemDpxEth.sol

The majority of the rDPX v2 repository was out of scope of this review because it had been reviewed previously. After the findings were presented to the Dopex team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Dopex and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Access controls were applied on nearly every function that modified state variables except for `redeem()` in ReceiptToken.sol. The access control logic was from OpenZeppelin's libraries and were applied properly on the functions. The address that will be assigned the admin privilege was not clear at the time of the audit.
Mathematics	Average	Only simple accounting math was needed in the in-scope contracts. However, some confusion exists around the different decimals used for percentage precision (1e8 vs 1e10 for 100%). Some incorrect math was found in the accounting logic related to rDPX and WETH backing reserves to be paid out during the dpxETH redemption process.
Complexity	Average	The contracts that were in-scope of this audit were not overly complex, but were connected to the overall Dopex protocol which has some complex mechanisms and tokenomics. The aspect that was in-scope was the ReceiptToken redemption process, which involved 3 custom Dopex tokens (rDPX, dpxETH, and the receipt token) that are stacked on top of each other. This complexity coupled with the algorithmic stabilization mechanism could be problem in extreme market situations.

Category	Mark	Description
Libraries	Average	An average number of external libraries were imported into the contracts. The library dependencies could be upgraded to newer versions. It should be noted that solmate and OpenZeppelin ERC20 libraries were imported into ReceiptToken.sol and these libraries sometimes have inconsistencies or incompatibilities between them.
Code stability	Average	Due to tight timelines, the code did not receive an ideal level of test coverage and was not in a very stable state at the start of the audit.
Decentralization	Low	Nearly every function call has an access modifier that lets only a privileged and centralized address call the function. Nearly all functions that store value allow the admin to transfer that value to a different arbitrary address.
Documentation	Low	The NatSpec was incomplete or unclear for many functions. Even if a function is only callable by admins, NatSpec should be added to allow curious users to understand the protocol they are using and to allow future governance members to input the proper values for governance calls.
Monitoring	Low	Events are missing from several functions, including places where the event is defined but not used.
Testing and verification	Low	The test coverage was quite low (around 50-60%) and was not in a finalized state at the start of the audit.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.

- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - Fee calculations in RedeemDpxEth.sol could lead to fees being greater than contract balance

When a Receipt Token is redeemed, fees for the transaction are calculated and collected on the RedeemDpxEth.sol contract. During this calculation, it is possible for the fees to be greater than the actual amount of funds that are sent to the RedeemDpxEth.sol contract. This causes the transaction to revert.

Technical Details

There are 2 underlying causes of this issue within the `redeemDpxEth()` function:

1. The fee calculations take into account a `wethAmount` that is not actually sent to the RedeemDpxEth.sol contract.
2. The fee is taken entirely in WETH even though the redemption occurs in a split of WETH and rDPX.

Assumptions for our walkthrough scenario below:

- `rdpxBacking` is 50% and `rdpxBackingPercentage` is 50%.
- `wethBacking` is 50% and `wethBackingPercentage` is 50%.
- User wants to redeem 2e17 Receipt tokens.
- 1e17 in dpxEth and 1e17 in WETH are removed from the Curve pool when `remove_liqudity()` is called.
- `feePercentage` is 26%.

```
    uint256 backingInRdpx = rdpxBackingInWeth.mul(1e10).div(
      IERC20WithBurn(addresses.dpxEth).totalSupply()
    );
    uint256 backingInWeth = wethBacking.mul(1e10).div(
      IERC20WithBurn(addresses.dpxEth).totalSupply()
    );
```

1 backingInRdpx is 50e8 and backingInWeth is 50e8.

```
(rdpxReceived, wethReceived) = _calculateRdpxAndWethReceived(
      backingInRdpx,
      backingInWeth,
      amount
    );
```

1 The `rdpxReceived` will be calculated as `1e17 * 50e8 * 1e18 / 2e16 / 100e8 = 25e17` and `wethReceived` will be calculated as `1e17 * 50e8 / 100e8 = 5e16`.

```
fee = (wethReceived +
      wethAmount +
      (
        rdpxReceived.mul(IRdpxV2Core(addresses.rdpxV2Core).getRdpxPrice()).div(
          1e18
        )
      )).mul(feePercentage).div(100e8);
```

1 `fee` will be calculated as `(5e16 + 1e17 + (25e17 * 2e16 / 1e18)) * 26e8 / 100e8 = 5.2e16`.

```
    wethReceived = wethReceived - fee;
```

1 `fee` will be 5.2e16 > `wethReceived` which will be 5e16 WETH and the transaction will revert.

POC:

```
  function testFeesTooGreat() public {
    //50e8 for each as initialized
    uint256 rpdxBackingPercent = 50e8;
    uint256 wethBackingPercent = 50e8;

    //26% fee
    uint256 feePercentage = 26e8;

    //backing is 50/50
    uint256 backingInRdpx = 50e8;
    uint256 backingInWeth = 50e8;

    //removed 1e17 of each dpxEth and weth from the curve pool
    uint256 dpxEthAmount = 1e17;
    uint256 wethAmount = 1e17;

    //oracle returns price of rdpx as 2% of eth
    uint256 rdxPrice = 2e16;

    (uint256 rdpxReceived, uint256 wethReceived) = _originalCalculate(rpdxBackingPercent,
wethBackingPercent, backingInRdpx, backingInWeth, dpxEthAmount);

    // calculate fee
    uint256 fee = (wethReceived +
      wethAmount +
      (
        rdpxReceived.mul(rdxPrice).div(
          1e18
        )
      )).mul(feePercentage).div(100e8);

    //will revert due to arithmetic underflow
    vm.expectRevert();
    wethReceived = wethReceived - fee;
  }
```

Impact

High. Users would not be able to redeem their Receipt Tokens.

Recommendation

Consider taking the fee in both rDPX and WETH. In addition, if the fee is only taken on the RedeemDpxEth.sol contract, the calculation should not include the `wethAmount` removed from the LP pool unless this amount is also transferred to the RedeemDpxEth.sol contract. This way, the contract can be sure it has enough funds to properly calculate fees and then transfer the funds to their respective destinations.

Developer Response

The Redemption logic has been changed so that the fee will not exceed the available amounts.

Commit Hash: 68d887cb35afaadbd02051ff1c420d840216eb16.

2. High - Incorrect decimal conversion in `getValueInWeth()`

One instance of 1e18 should be changed to 1e8.

Technical Details

The last line of `getValueInWeth()` includes a division by 1e18 while it should actually divide by 1e8. Other places where `getRdpxPrice()` is used to convert rDPX value to WETH are found in RdpxV2Core.sol use 1e8 (1, 2, 3).

Impact

High. Return value from `getValueInWeth()` has incorrect decimal conversion.

Recommendation

Change 1e18 to 1e8.

Developer Response

This issue has been fixed where all the oracles return the price in 1e18.

Commit Hash: 09fce416dfdbd63495e36aecc33d0c153b783a5a

Medium Findings

1. Medium - Univ3 positions could grow to DoS fee collections

When `collectFees()` is called from the UniV3LiquidityAmo.sol, fees are collected for every single position that the AMO is owns. If enough positions are added by the AMO, any call to `collectFees()` will exceed the block gas limit on Arbitrum.

Technical Details

Within UniV3LiquidityAmo.sol, an array with Position data is appended to each time liquidity is added by the AMO. Collecting fees from these positions is done with the following logic:

```
    for (uint i = 0; i < positions_array.length; i++) {
      Position memory current_position = positions_array[i];
      INonfungiblePositionManager.CollectParams
        memory collect_params = INonfungiblePositionManager.CollectParams(
          current_position.token_id,
          rdpxV2Core,
          type(uint128).max,
          type(uint128).max
        );

      // Send to custodian address
      univ3_positions.collect(collect_params);
    }
```

This operation is gas intensive as it loads values from storage and then performs an external call on Univ3 positions contract to collect the LP fees. If the array grows beyond a certain size, fees cannot be collected until some positions have been removed from the array by calling `removeLiquidity()`. In testing, it only took 1176 positions for calling `collectFees()` on the UniV3LiquidityAmo.sol contract to cost more than the block gas limit (32,000,000 on Arbitrum).

POC:

```
function testUniV3Collect() public {

      // create a v3 pool
  testInitializeV3Amo();

      // test add liquidity
  int24 minTick = (-78245 / int24(10)) * 10 + 10;
  int24 maxTick = (-73136 / int24(10)) * 10;
  uint24 fee = 500;

  UniV3LiquidityAMO.AddLiquidityParams memory params = UniV3LiquidityAMO
    .AddLiquidityParams(
      address(rdpx),
      address(weth),
      minTick,
      maxTick,
      fee,
      1e16,
      1e16,
      0,
      0
    );
  for (uint256 i; i < 1176; i++) {
    uniV3LiquidityAMO.addLiquidity(params);
  }
  uint256 start = gasleft();
  uniV3LiquidityAMO.collectFees();
  uint256 end = gasleft();

  console.log(start - end);
}
```

Impact

Medium. Independent fee collection could become DoS'd. The impact, however, is lowered due to the fact that an admin can call `removeLiquidity()` to remove positions from the array, making it possible to again call `collectFees()`.

Recommendation

Consider adding parameters to the `collectFees()` function so that the array can be "chunked" up and a set amount of positions can have their fees collected from at a time. This ensures that no matter the size of the array, fees can always be collected even if it takes more than one transaction. Additionally, some small steps can be taken to reduce the gas burden of the function logic. First, the array length could be cached in memory to reduce an extra SLOAD each iteration. Second, an entire struct is loaded from storage and into memory each iteration when only one of its components is needed.

Developer Response

Acknowledged, this is a unlikely scenario as the AMO is admin controlled and steps will be taken to make sure this does not happen.

2. Medium - Lack of checkpoints allows gaming of CRV rewards

When `claimRewards()` in ReceiptToken is called, CRV tokens are distributed to the treasury and to a receipt token staking address. If the CRV rewards in the receipt token staking address can be redeemed immediately, the rewards can be gamed by users frontrunning the `claimRewards()` action and owning tokens for only a short time.

Technical Details

There is no checkpointing in `claimRewards()` that verifies token holders have held the token for a certain duration to be eligible to receive these rewards. The result is that an address that has been a token holder for only a few blocks could receive equivalent rewards per token as a long-term token holder. The exact way in which this can be exploited includes contracts that are out of scope of this review, but checkpoints are a standard way of normalizing rewards over a larger timeframe and there is no checkpointing in ReceiptToken.

Impact

Medium. Gaming of rewards can benefit users who game the system and penalizes long-term token holders.

Recommendation

The contract where the CRV reward tokens are sent, the receipt token staking address, is out

of scope of this review. But it should have some locking or checkpointing mechanism to prevent short-term token holders from gaming the system and getting more rewards than they deserve based on their short holding period.

Developer Response

Acknowledged, the contract that is responsible for distributing the rewards handles this.

3. Medium - Reward transfer logic error in ReceiptToken

Different assumptions in two functions of ReceiptToken could lead to CRV reward tokens remaining in ReceiptToken getting transferred to unintended recipients.

Technical Details

The logic in `setPercentage()` requires that `_treasuryPercentage + _receiptTokenPercentage <= 100e8`. A problem can occur in the case that `_treasuryPercentage + _receiptTokenPercentage < 100e8`, because the logic in `claimRewards()` assumes that the condition `_treasuryPercentage + _receiptTokenPercentage == 100e8` is true. The following hypothetical scenario illustrates the problem.

1. The contract admin calls `setPercentage()` such that `_receiptTokenPercentage` is set to 60% and `_treasuryPercentage` is set to 30%, leaving 10% of rewards in the ReceiptToken contract.
2. `claimRewards()` is called and 1000 CRV tokens are received. 600 tokens go to receipt token holders and 300 tokens go to treasury.
3. `claimRewards()` is called and 1000 CRV tokens are received. Because 100 tokens were left in the contract from the first `claimRewards()` call, there are now 1100 CRV rewards tokens in the contract. 660 tokens go to receipt token holders and 330 tokens go to treasury, leaving 110 tokens in the contract. The token split still matches the intended target, but as more tokens accumulate in the ReceiptToken, more rewards get sent for every `claimRewards()`. The rewards are “delayed” compared when less than 100% of reward tokens are distributed compared to a situation where 100% of the rewards are distributed.
4. The contract admin calls `setPercentage()` such that `_receiptTokenPercentage` is set to 70% and `_treasuryPercentage` is set to 30%.
5. `claimRewards()` is called and 1000 CRV tokens are received. Because 110 tokens were left in the contract from the first `claimRewards()` call, there are now 1110 CRV rewards tokens in the contract. 777 tokens go to receipt token holders and 333 tokens go to treasury,

leaving 0 tokens in the contract. The token distribution that was previously delayed results in a sudden boosted payout.

The potential issues from this sequence of events includes:

1. Tokens that remain in the ReceiptToken contract are underutilized, resulting in reduced capital efficiency.
2. Users backrunning any `setPercentage()` call that changes how reward tokens are allocated.

Impact

Medium. Existing logic can result in extra tokens transferred to treasury or receipt token holders.

Recommendation

The simplest solution is to modify `setPercentage()` to require `_treasuryPercentage + _receiptTokenPercentage == 100e8` to remove the case where the sum of percentages is less than 100e8.

Developer Response

Acknowledged, the function to update the percentages is an admin controlled function and steps will be taken to make sure this does not happen.

4. Medium - Admins possess ability to access and impact user funds

The set of smart contracts within the scope of this engagement implemented a highly centralized architecture. Certain roles, particularly the default admin, of the contracts possessed the ability to not only impact the services rendered by the dApp but in some cases the ability to access underlying funds deposited into the dApp by users.

Technical Details

Below is a non-inclusive list of locations and possible courses of action an admin user could take to rug pull users of the application:

- Withdraw user funds.
- Set slippage tolerance too high that redemptions revert.
- Prevent users from redeeming or depositing via pause.
- Set key addresses to any address of their choice.
- Univ3 AMO can rug funds up to the allowance from rdpxv2Core.

Impact

Medium. The number of potential attack vectors and their outcomes is severe, however, exploiting them requires either key compromise or a key-holder to act in bad faith.

Recommendation

In the now, consider providing documentation for users regarding the permissions of various roles within the system and any key management strategies that will be observed to limit the likelihood of issues arising from this. Ensure that the keys for accounts as sensitive as the default admin role are well-secured and that steps are taken to limit a single point of failure, such as by requiring multiple signatures for admin activity. In the future, consider taking steps to decentralize aspects of the protocol to help prevent the possibility of such issues occurring.

Developer Response

Acknowledged, the admin keys are secured and will be a multisig address.

5. Medium - `recoverERC721()` cannot recover ERC721 tokens

`recoverERC721()` in UniV3LiquidityAMO.sol sends the tokens to rdpxV2Core, but the rdpxV2Core contract has no good way to handle the ERC721 tokens that it receives.

Technical Details

Although UniV3LiquidityAMO.sol has a `recoverERC721()` function, the recipient that receives the ERC721 is hard-coded. The hard-coded address is rdpxv2Core, and this contract has no way to recover a trapped ERC721. If the goal is to provide a way for the governance multisig to recover a trapped ERC721, modifications in the current contracts must be made.

Impact

Medium. The likelihood of a trapped ERC721 is low, but the recovering mechanism does not work as intended.

Recommendation

Modify the code to allow a multisig to receive the trapped ERC721. The devs were already aware of this issue and are in the process of mitigating it.

Developer Response

Fixed, the recoverERC721 function has been changed.

Commit Hash: cc2ff5c10bff7ce34b38d0934e886e5a1a2bfcb0

6. Medium - Slippage amount calculated in same transaction it is used

Within ReceiptToken.sol, slippage calculations were made in the same transaction that they were used. When slippage protection is calculated on-chain in the same transaction that it is used, the user is not protected from price changes or transaction ordering issues, such as sandwich attacks.

Technical Details

When a user goes to redeem their ReceiptTokens, ReceiptToken.sol will call `_removeLiquidity()` which in turn calls `remove_liquidity()` on the dpxEth-WETH Curve pool. Before it interacts with the Curve pool, it calculates the slippage it will allow by taking the amount of LP tokens to be redeemed and normalizing an expected output of each individual token based on a preset value for the `liquiditySlippageTolerance` variable.

The problem is that the slippage calculation is made using information unique to the individual transaction. In other words, it doesn't react to any change in the Curve pool's ratio or reserves that occur immediately prior to this transaction, something that one would expect from slippage protection.

This issue is similarly present in the `deposit()` via the `_addLiquidity()` function.

Impact

Medium. Slippage calculations that are made using information unique to the individual transaction do not offer adequate protection for users from price changes and transaction ordering issues.

Recommendation

Allow users to perform off-chain calculations or offer a UI to perform these calculations for them to determine the going rate for the dpxEth-WETH LP token. Then allow users to pass in slippage parameters, so that their transactions are better able to hold up to transaction ordering attacks.

Developer Response

Fixed, min amounts are passed in as parameters in `redeem()` and `removeLiquidity()`.

7. Medium - Slippage allowed in ReceiptToken.sol is the complement of the tolerance

When calculating the minimum tokens that should be received when calling `redeem()`, the expected amount of tokens to be received is multiplied by the `liquiditySlippageTolerance`. Since this translates to the minimum success condition for removing liquidity from the curve pool, the true slippage allowed is the complement of the `liquiditySlippageTolerance`. Initially, the `liquiditySlippageTolerance` will be set to 0.5%, causing the slippage allowed to be 99.5%.

Technical Details

`redeem()` calls `_removeLiquidity()` to exchange LP tokens for their underlying WETH and dpxEth. Within `_removeLiquidity()`, the minimum amount for each underlying token is calculated using the following formulas:

```
min_amounts[0] =
  (token0Amount * liquiditySlippageTolerance * amount) /
  totalSupply /
  1e8;

min_amounts[1] =
  (token1Amount * liquiditySlippageTolerance * amount) /
  totalSupply /
  1e8;
```

This borrows from the formula Curve uses to determine the amount of tokens that will be sent out (example from stETH pool). However, this calculated amount is then multiplied by the slippage tolerance percent instead of reduced by an amount in accordance with the slippage tolerance.

Impact

Medium. Slippage calculations performed as initially set will not adequately protect redemptions from slippage.

Recommendation

As mentioned in issue “Slippage amount calculated in same transaction it is used”, slippage calculations should be done off-chain before the transaction in which they are going to be used. When performing the calculation, ensure that the minimum amount is reduced by the acceptable slippage and not multiplied by the slippage tolerance percent.

Developer Response

Fixed, the min amount calculation has been changed, min amount is passed in as a parameter to `redeem()`.

Low Findings

1. Low - Log emits a deleted mapping entry

An event emit in `removeLiquidity()` of UniV3LiquidityAmo.sol uses a mapping entry that is deleted before emitting. This means the event will always emit zero, which is unintended behavior.

Technical Details

This event log emit in `removeLiquidity()` emits a mapping value that is already deleted by the time the event is emitted. The mapping deletion happens a few lines before the event is emitted. This means the log will always emit 0.

Impact

Low. The event does not emit the intended value, which could lead to difficulties in relying on on-chain event data in the future.

Recommendation

Move the event call to earlier in the function.

```
+  emit log(positions_mapping[pos.token_id].token_id);
    delete positions_mapping[pos.token_id];

    // send tokens to rdpxV2Core
    _sendTokensToRdpxV2Core(tokenA, tokenB);

    emit log(positions_array.length);
-    emit log(positions_mapping[pos.token_id].token_id);
```

Developer Response

Fixed, the event has been moved.

Commit Hash: 533ec0aaadf414530c9abd8bcd55f11c1f832903

2. Low - Inconsistent design between ReceiptToken and RedeemDpxEth

ReceiptToken.sol and RedeemDpxEth.sol both have an Addresses struct that sets the addresses of external contracts called elsewhere in the contract. But the variable storing this struct is handled differently in the two contract, showing inconsistency in adhering to protocol design choices.

Technical Details

In ReceiptToken.sol, the Addresses struct can be updated by calling `setAddresses()`. RedeemDpxEth.sol has no such function. Instead, the same variable in RedeemDpxEth.sol can be declared immutable because they are set only in the constructor and cannot be updated. This could lead to a problem in RedeemDpxEth.sol if values such as `rdpxV2Core` should be possible to update, because the value cannot be modified in RedeemDpxEth.sol. If the inverse is true and values such as `rdpxV2Core` will not be modified after the contract is deployed, then ReceiptToken.sol should be updated to remove `setAddresses()` and move the setting of these values to the constructor like RedeemDpxEth.sol. This would reduce the centralization of the protocol by preventing the admin from updating which external contract are called by ReceiptToken.sol.

Impact

Low. Variables such as `rdpxV2Core` can be updated in ReceiptToken.sol but not in RedeemDpxEth.sol. If the address of `rdpxV2Core` might change, then RedeemDpxEth.sol has no way to process this change. If the address of `rdpxV2Core` is not intended to change, then ReceiptToken.sol should be updated to make the values immutable.

Recommendation

Maintain a consistent approach to setting the Addresses struct variable between ReceiptToken.sol and RedeemDpxEth.sol. Note that other out of scope contracts such as RdpxV2Core.sol also have a `setAddresses()` function like ReceiptToken.sol.

Developer Response

Acknowledged, this is by design.

3. Low - Unbounded fee percentage

The `feePercentage` state variable in RedeemDpxEth.sol has an implied maximum value of 100e8, but `setFeePercentage()` does not verify that this upper limit is not exceeded.

Technical Details

The contract admin can set `feePercentage` to any non-zero value with `setFeePercentage()`. If the admin accidentally sets the fee percentage to a value that exceeds 100%, this will cause problems elsewhere in the contract where `feePercentage` is used in calculation. `feePercentage` should be capped to values equal to or less than 100e8, as anything above this is guaranteed to prevent users from being able to redeem dpxETH.

Impact

Low. The missing check on the upper bound of `feePercentage` would lead to a revert during the dpxETH redemption process.

Recommendation

Modify the `require()` check in `setFeePercentage()` to confirm that `_feePercentage` does not exceed 100e8.

```
require(
-       _feePercentage > 0,
+      _feePercentage > 0 && _feePercentage <= 100e8,
       "RedeemReceiptToken: fee percentage can not be 0"
    );
```

Developer Response

Fixed, the fee percentage is capped at 100e8.

4. Low - `getValueInWeth()` view function returns no values

`getValueInWeth()` is intended to return `totalWethAmount` but does not return any value.

Technical Details

The name of the function `getValueInWeth()` indicates it should return the value of `totalWethAmount`, but there is no return value. This means the view function is useless and does not perform it's intended purpose.

Impact

Low. Function does not work as designed.

Recommendation

Add a return value to `getValueInWeth()`.

```
+  function getValueInWeth() public view returns (uint256 totalWethAmount) {
-  function getValueInWeth() public view {
    Position memory position;
-    uint256 totalWethAmount;
```

Developer Response

Fixed, the function now returns the totalWethAmount.

5. Low - No protection in `addLiquidity()` for wrong token pair

`addLiquidity()` in UniV3LiquidityAMO.sol assumes that tokenA or tokenB is the rDPX token, as evidenced by this line. But there are no checks that confirm that tokenA or tokenB is the rDPX token, so it is possible for `addLiquidity()` to be called when this assumption is not true.

Technical Details

`addLiquidity()` has an implied assumption that only valid tokenA and tokenB values will be used, but there is no logic in the contract to confirm this. A similar contract, the Frax Univ3 AMO, has a check in `addLiquidity()` to confirm tokenA or tokenB are allowed collaterals or are the rDPX-equivalent token.

Impact

Low. The code has a built-in assumption but will not revert if the assumption is broken if the admin forgets about this assumption.

Recommendation

If the decision was made that the admin will not make this mistake, then the check is not necessary because it will require extra gas. But if it is preferred to protect against typos or incorrect admin inputs, a check should be added to confirm rDPX and WETH are the input tokens. Create a new immutable address variable `weth` and add the check:

```
require((params._tokenA == weth && params._tokenB == rdpx) || (params._tokenB == weth &&
params._tokenA == rdpx), "Token pair not allowed");
```

Developer Response

Fixed.

6. Low - UniV3LiquidityAMO should have separation of roles

The UniV3LiquidityAMO contract will most likely be operated by a bot in the future. This bot should be able to perform actions that are part of the standard AMO process, but edge case functions that should be handled by humans should not be accessible to the bot. Therefore, some separation of roles is needed instead of using one role for all access controlled functions in the contract.

Technical Details

The `DEFAULT_ADMIN_ROLE` role is the only privileged role in UniV3LiquidityAMO.sol, and it can access all the functions in the contract. But some functions, such as `approveTarget()`, `recoverERC20()`, `recoverERC721()`, and `execute()` are designed to handle unexpected edge cases and should not be accessible to a bot. These functions should have a different role that can call them.

Impact

Low. A bot should only have control over actions that make sense to automate, not privileged actions that a human should call.

Recommendation

Create another role in UniV3LiquidityAMO.sol that is intended to be a multisig address operated by humans, not bots. Then modify the modified on the functions `approveTarget()`, `recoverERC20()`, `recoverERC721()`, and `execute()` to only allow the human-operated multisig to call these functions.

Developer Response

Acknowledged, the admin role for uni v3 will be a separate address from the other contracts so the risk is minimized.

Gas Saving Findings

1. Gas - Remove `safeApprove()` code

`approveTarget()` has some unnecessary logic that can be removed for gas savings on deployment and on function calls.

Technical Details

The function UniV3LiquidityAMO.sol `approveTarget()` has an if statement that is designed to handle the case of tokens like USDT that require an allowance of zero before calling approve with a non-zero value. But this Univ3 AMO contract is only designed for WETH and rDPX on Arbitrum, neither of which have this special approval requirement. Even USDT on Arbitrum does not have this requirement, unlike USDT on mainnet Ethereum. Even if a zero allowance is needed, the simple `approve()` call can handle the situation, although if it takes two calls of the function (first approving a value of zero, then approving the non-zero value).

Impact

Gas savings.

Recommendation

Simplify `approveTarget()` so that the only line of code in the function is `IERC20WithBurn(_token).approve(_target, _amount);`. The `use_safe_approve` bool function argument can be deleted.

2. Gas - Cache state variables in `redeemDpxEth()`

State variables that are used more than once should be cached in a local variable. This reduces SLOADs and saves gas.

Technical Details

The state variables `rdpxBackingPercentage` and `wethBackingPercentage` are used in four places each in `_calculateRdpxAndWethReceived()`. Cache these state variables into local variables inside the first if statement (indicating overcollateralization) to save gas.

Impact

Gas savings.

Recommendation

Cache state in local variables instead of reading again from storage.

3. Gas - Set `RDPX_V2_CORE_ROLE` in existing function

An address needs the `RDPX_V2_CORE_ROLE` role to call `deposit()` in ReceiptToken. This role is not assigned in any existing function of ReceiptToken, but should be. Avoiding a separate tx would save the admin gas.

Technical Details

In ReceiptToken, the only way to set the address assigned the `RDPX_V2_CORE_ROLE` role is to call `grantRole()` inherited from AccessControl. But it would be better and cost less gas to set the `RDPX_V2_CORE_ROLE` role in `setAddresses()`. This is because `setAddresses()` already has a `_rdpxV2Core` argument to update the rDPX v2 core address. The role could be assigned in the constructor like what is done in RedeemDpxEth, but using `setAddresses()` may be a more sensible choice given the existing function argument.

Impact

Gas savings.

Recommendation

Add this internal call to `setAddresses()`. Consider adding a comment to remind the caller to

revoke the role of the previous `RDPX_V2_CORE_ROLE` address.

```
_setupRole(RDPX_V2_CORE_ROLE, _rdpxV2Core); // remember to revoke previous address with this role
```

4. Gas - Remove useless ReceiptToken functions

ReceiptToken has some functions that are not useful, so they should be removed to save gas on deployment and reduce protocol complexity.

Technical Details

Some functions in ReceiptToken are borrowed from ERC4626, which implies there is some compounding of value. But the design of ReceiptToken does not compound value - instead, the CRV rewards are sent to the treasury and receipt token holders in `claimRewards()`. This means the functions inspired by ERC4626 can be removed.

Impact

Gas savings.

Recommendation

Remove the functions `totalCollateral()`, `previewDeposit()`, `previewRedeem()`, `convertToShares()`, and `convertToAssets()` from ReceiptToken. Remove the variable `_totalCollateral` and use `totalSupply` where a replacement is needed. Additionally, modify these lines in `deposit()` and `redeem()` to remove references to these functions.

```
require(
-      (shares = previewDeposit(liquidityTokensReceived)) != 0,
-      "ReceiptToken: ZERO_SHARES"
+     liquidityTokensReceived != 0, "ReceiptToken: ZERO_SHARES"
    );
- _mint(msg.sender, shares);
+ _mint(msg.sender, liquidityTokensReceived);
```

```
- uint256 lpTokenAmount = previewRedeem(amount);
+ uint256 lpTokenAmount = amount;
```

5. Gas - Immutable variables are cheaper

If a variable is not changed after the constructor, it can be immutable to save gas.

Technical Details

`rdpx`, `rdpxV2Core`, and `uniV3Pool` can be immutable for gas savings in UniV3LiquidityAMO. In RedeemDpxEth, `addresses` can be immutable. In ReceiptToken, `weth`, `dpxEth`, and `collateral` can be immutable.

Impact

Gas savings.

Recommendation

Make variables immutable for gas savings.

6. Gas - Use `block.timestamp` instead of `type(uint256).max` for deadline

A minor gas savings can be achieved using `block.timestamp` instead of `type(uint256).max` in INonfungiblePositionManager.MintParams.

Technical Details

The deadline in the MintParams of `addLiquidity()` is `type(uint256).max`, which means there is no deadline set for the mint action. Instead, consider using `block.timestamp`, which will add a timestamp that will prevent the mint from happening with a delay and also saves gas. Many on-chain contracts using MintParams use `block.timestamp`.

Impact

Gas savings.

Recommendation

Replace `type(uint256).max` with `block.timestamp`. `block.timestamp` is already used for the deadline elsewhere in the AMO, in `DecreaseLiquidityParams`.

7. Gas - Unnecessary calculation in `claimRewards()` in ReceiptToken.sol

An unnecessary subtraction operation should be removed from `claimRewards()` for gas savings.

Technical Details

The CRV rewards are calculated in `claimRewards()` with an unnecessary subtraction of `crvRewards`. When this subtraction happens, the value of `crvRewards` will always be zero, so the subtraction should be removed as shown.

```
    crvRewards =
-       IERC20WithBurn(addresses.crv).balanceOf(address(this)) -
-       crvRewards;
+       IERC20WithBurn(addresses.crv).balanceOf(address(this));
```

Impact

Gas savings.

Recommendation

Remove unnecessary math operation.

8. Gas - Make public functions external where possible

Many public functions are not called internally, meaning they can be declared external instead to save gas on deployment.

Technical Details

UniV3LiquidityAMO.sol has many public functions that are not called internally (1, 2, 3, 4). These instances are examples, there are other instances of this same issue not highlighted here. Changing these functions from public to external saves gas on deployment.

Impact

Gas savings.

Recommendation

Change function visibility for gas savings.

9. Gas - Use constant variables for gas savings

Variables that are set to a known value before deployment and are not changed after deployment can be constant variables to save gas.

Technical Details

The variables `univ3_factory`, `univ3_positions`, and `univ3_router` are set in the constructor of UniV3LiquidityAmo.sol and are not modified after this point. Because the address values are known and not modified after deployment, these variables can be changed to constant variables to save gas.

Impact

Gas savings.

Recommendation

Change variables to constants when possible.

10. Gas - Avoid `&&` logic in require statements

Using `&&` logic in require statements uses more gas than using separate require statements. Dividing the logic into multiple require statements is more gas efficient.

Technical Details

ReceiptToken.sol has two require statements with `&&` logic (1, 2). RedeemDpxEth.sol also has two require statements with `&&` logic (1, 2). Splitting this compound require statement into many require statements can save gas when the function is called and save gas on deployment.

Impact

Gas savings.

Recommendation

Replace require statements that use && by dividing up the logic into multiple require statements. For example:

```
-    require(
-      _rdpxV2Core != address(0) &&
-        _curvePool != address(0) &&
-        _curveGauge != address(0) &&
-        _curveChildGauge != address(0) &&
-        _rdpx != address(0) &&
-        _crv != address(0) &&
-        _treasury != address(0) &&
-        _receiptTokenStaking != address(0) &&
-        _redeemDpxEth != address(0),
-      "ReceiptToken: ZERO_ADDRESS"
-    );
+    require(_rdpxV2Core != address(0));
+    require(_curvePool != address(0));
+    require(_curveGauge != address(0));
+    require(_curveChildGauge != address(0));
+    require(_rdpx != address(0));
+    require(_crv != address(0));
+    require(_treasury != address(0));
+    require(_receiptTokenStaking != address(0));
+    require(_redeemDpxEth != address(0));
```

Informational Findings

1. Informational - Missing event emits

Some functions that transfer values or modify state variables do not have events. Events can assist with analyzing the on-chain history of contracts and are therefore beneficial to add in important functions.

Technical Details

Functions that could have events added include:

- ReceiptToken.sol: `redeem()`, `_removeLiquidity()`, `setPercentage()`
- UniV3LiquidityAMO.sol: `collectFees()`, `addLiquidity()`, `swap()`

Notably there is already a redeem event in ReceiptToken.sol but it is not used.

Impact

Informational.

Recommendation

Add events to the functions listed above.

2. Informational - Incorrect interface definition

IRedeemDpxEth has an incorrect function definition.

Technical Details

IRedeemDpxEth defines `redeemDpxEth()` as returning two uint256 values. The actual definition of `redeemDpxEth()` returns 3 uint256 values.

Impact

Informational.

Recommendation

Fix IRedeemDpxEth by adding the correct number of return arguments, then fix the instances where `redeemDpxEth()` is called.

3. Informational - Misleading function and variable naming

Within `redeemDpxEth()`, a call to `getRdpxAndWethInAmo()` looks like it should return only the collateral held in the AMOs. However, `getRdpxAndWethInAmo()` actually adds the collateral to the passed in parameters and returns the sum. The name of this function should more clearly describe what the function does.

The `backingInRdpx` and `backingInWeth` variables can also receive better names, because these variables store percentage values.

Technical Details

The function `getRdpxAndWethInAmo()` is named in a way that sounds like the function should return amounts only from the AMO contracts. But as seen below, `getRdpxAndWethInAmo()` actually sums the collateral in the AMOs with the passed in arguments and returns the total value. A better name for this function might be `_sumWithRdpxAndWethInAmo()`.

The `backingInRdpx` and `backingInWeth` variables can also be renamed to include the word "percent" in their names, because these variables represent percentage values.

```
function getRdpxAndWethInAmo(
  uint256 _rdpxBacking,
  uint256 _wethBacking
)
  internal
  view
  returns (uint256 rdpxBackingWithAmo, uint256 wethBackingWithAmo)
{
  // add backing in amo's
  (uint256 wethBackingInAmo, uint256 rdpxBackingInAmo) = IRdpxV2Core(
    addresses.rdpxV2Core
  ).getCollateralInAmos();
  rdpxBackingWithAmo = rdpxBackingInAmo + _rdpxBacking;
  wethBackingWithAmo = wethBackingInAmo + _wethBacking;
}
```

Impact

Informational.

Recommendation

Either modify the logic of `getRdpxAndWethInAmo()` to remove the summations at the end of the function (and remove the function arguments) or rename the function to better indicate the summation process that happens. Also consider prepending a '_' to the function name because `getRdpxAndWethInAmo()` is an internal function.

Change the names of `backingInRdpx` and `backingInWeth` variables to `backingPercentInRdpx` and `backingPercentInWeth`.

4. Informational - Loss of precision from multiple division operations

Solidity does not support floating point numbers, resulting in division operations rounding down. If multiple division operations happen in a single calculation, this can lead to a loss in precision. This loss of precision scenario is found in two different functions, `_removeLiquidity()` and `redeemDpxEth()`.

Technical Details

There are at least two places where a loss of precision occurs due to extra division operations. `_removeLiquidity()` in ReceiptToken.sol calculates `min_amount` with multiple division operations, which reduces the precision compare to a reorganized version of the calculation. The same pattern is found at the end of `redeemDpxEth()` in RedeemDpxEth.sol.

Impact

Informational.

Recommendation

Rewrite `min_amount` calculations in `_removeLiquidity()` similar to:

```
min_amounts[0] =
  (token0Amount * liquiditySlippageTolerance * amount) /
-  totalSupply /
-  1e8;
+  (totalSupply * 1e8)
```

The calculations in `redeemDpxEth()` can be modified in the same way.

```
rdpxReceived = amount
  .mul(rdpxBackingPercentage)
  .mul(1e18)
-  .div(rdpxPrice)
-  .div(100e8);
+  .div(rdpxPrice * 100e8);
```

5. Informational - Similar NatSpec has different meanings

NatSpec comments in two different dopex contracts have different meanings. This can lead to confusion about how the values should be used when modifying the contract code in the future.

Technical Details

A comment in ReceiptToken.sol reads `The slipage tolerance in 1e8`. The slippage value is later divided by 1e8, indicating that 100% corresponds to 1e8. Compare this to a comment in the rdpx v2 token contract that reads `Buy-side fees percentage in 1e8 precision.` The same fee value is later divided by 1e10, indicating that 100% corresponds to 1e10.

The meaning of 1e8 precision can mean 100% = 1e8 or it can mean 100% = 1e10 depending on the contract.

Impact

Informational.

Recommendation

Use standard language and precision across different contracts of dopex. The simplest way to do this is to modify `liquiditySlippageTolerance` from `5e5` to `5e7` and change the denominator in the division of this value to 100e8 (1, 2).

6. Informational - Replace magic numbers with constants

Constant variables should be used in place of magic numbers to prevent typos. Using a constant adds a description to the value to explain the purpose it serves.

Technical Details

ReceiptToken.sol uses the magic number 100e8 in several places (1, 2, 3). RedeemDpxEth.sol uses 100e8 and 1e10 for the same purpose (1, 2, 3, etc.). The value 1e8 is also used in ReceiptToken.sol (1, 2). These values can be stored in a constant variable named `totalPercentage` to give the value more meaning and avoid typos in places where this value is used. This will not change gas consumption.

Impact

Informational.

Recommendation

Use constant variables instead of magic numbers.

7. Informational - Univ3 AMO incompatible with fee-on-transfer tokens

The transfer logic in ReceiptToken.sol will fail for fee-on-transfer tokens. There currently is no plan to use this code with fee-on-transfer tokens, but this limitation should be documented in case there are strategy changes in the future.

Technical Details

In Univ3LiquidityAmo.sol, `addLiquidity()` has a combination of `transferFrom()` and transferring that same amount, which will revert when transferring a fee-on-transfer token. This is because the initial `transferFrom()` action will result in less than the transferred amount arriving at the destination (because the token fee amount of value is not received), meaning that less than the originally transferred value can be transferred out of the contract.

Impact

Informational.

Recommendation

If there are plans to support other tokens in the future, modify the logic in Univ3LiquidityAmo.sol to call `transferFrom()` to directly send tokenA and tokenB to the Uniswap v3 pool, instead of using a two-step transfer process.

8. Informational - `_setupRole()` and `safeApprove()` are deprecated

Deprecated functions from OpenZeppelin library imports are used, but it is a best practice to replace these with functions that will remain in future versions of the library.

Technical Details

To promote future compatibility, we recommend using `_grantRole()` in lieu of `_setupRole()`. `_setupRole()` has been deprecated and starting from 5.X, `_setupRole()` no longer exists in the AccessControl library.

A similar suggestion applies to `safeApprove()`, which was deprecated in 2020. `safeApprove()` still exists in OZ 4.X versions but is removed in OZ 5.X.

Impact

Informational.

Recommendation

Avoid using deprecated functions when possible. Replace `_setupRole()` with `_grantRole()`. Replace `safeApprove()` with `safeIncreaseAllowance()`.

9. Informational - Upgrade dependencies

The OpenZeppelin contracts dependency is version 4.8.0, which is outdated. Consider updating to a newer version. The solmate version used is listed as version 6.7.0, but the main branch of the solmate repository is at 6.2.0, so it may be worth updating the solmate import to align with a more active or production-ready development branch of the repository.

Technical Details

OZ library v4.8.0 is not the latest version available. If there's any breaking changes in v4.9.3 that make it unusable, consider at least upgrading to v4.8.3 which fixes some minor issues in v4.8.0.

Impact

Informational.

Recommendation

Upgrade OZ dependency to a newer version. Determine which solmate library version is best to use.

10. Informational - Typo in imports

There are typos in some imports.

Technical Details

contracts/mocks/MockRdpxEthPriceOracle.sol has a typo in an import statement.

```
- import { IUniswapV2Pair } from "../uniswap_v2/IUniswapV2Pair.sol";
+ import { IUniswapV2Pair } from "../uniswap_V2/IUniswapV2Pair.sol";
```

tests/RdpxV2CoreTest.t.sol has three typos in import statements.

```
- import { UniV2LiquidityAMO } from "contracts/amo/UniV2LiquidityAMO.sol";
+ import { UniV2LiquidityAMO } from "contracts/amo/UniV2LiquidityAmo.sol";
- import { UniV3LiquidityAMO } from "contracts/amo/UniV3LiquidityAMO.sol";
+ import { UniV3LiquidityAMO } from "contracts/amo/UniV3LiquidityAmo.sol";
```

```
- import { ReLPContract } from "contracts/relp/ReLPContract.sol";
+ import { ReLPContract } from "contracts/reLP/ReLPContract.sol";
```

Impact

Informational.

Recommendation

Fix typos.

Final remarks

This audit focused on a very small amount of code compared to the overall size of the Dopex protocol. Many key elements, such as the rdpxV2Core contract and the custom rDPX price oracle, were not in scope of this audit. The off-chain logic of the Univ3 AMO was also out of scope, and because so many input arguments into the functions of that contract are set by the caller, it is difficult to point to any issues in Univ3 AMO usage until there are on-chain transactions.

With any algorithmic token stabilization mechanism comes the risk of a bank run if the stabilization mechanism causes a negative feedback loop. Preventing this scenario requires good design, but most of these design elements were out of scope of this audit. One possible risk is a scenario where rDPX price depegs from the price of WETH, which could cause

users to seek safe haven and exchange their rDPX token for WETH in the Univ3 pool, resulting in the depletion of WETH reserve assets. A similar issue could arise around dpxETH is the value of the underlying tokens backing dpxETH, specifically rDPX, act in unintended ways.