CANTINA

Napier v2

Competition

March 11, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

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

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

2 Security Review Summary

Napier is a suite of decentralized financial products that offers access to open financial services on the Ethereum Virtual Machine, with various entities and individuals contributing to its development and adoption.

From Jan 20th to Feb 10th Cantina hosted a competition based on napier-v2. The participants identified a total of **16** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 2
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 9

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Wrong reward distribution formula, YT holders can get more rewards for themselves by just collecting accrued YBT yield

Submitted by Catowice

Severity: High Risk

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

Summary: Wrong reward distribution allows a YT holder to get significantly more external rewards than other YT holders simply by claiming their YBT yield.

Finding description: YT holders, aside from yields in the form of YBT, are also entitled to the any external rewards that normal YBT holders are entitled to.

Whenever external rewards are collected by the PT, the formula for the rewards entitled per user is YT_{user}/YT, where:

- YT_{user} denotes the user's YT balance.
- YT denotes the total YT supply.

Indeed, the reward is distributed across the YT total supply in the form of `newIndex`.

- RewardMathLib.sol#L62:

```
function updateIndex(RewardIndex index, uint256 ytSupply, uint256 totalAccrued)
    internal
    pure
    returns (RewardIndex newIndex, uint256 lostReward)
{
    if (ytSupply == 0) return (index, totalAccrued);
    newIndex = index + RewardIndex.wrap(FixedPointMathLib.divWad(totalAccrued, ytSupply).toUint128());
    ↪  // @audit new index is updated according to yt supply
    lostReward = totalAccrued - FixedPointMathLib.mulWad(ytSupply, (newIndex - index).unwrap());
}
```

And the reward is accrued according to the user's owned YT balance.

- RewardMathLib.sol#L93:

```
function accrueUserReward(
    mapping(address user => Reward userReward) storage self,
    RewardIndex index, // rewardIndex(t_2)
    address account, // u
    uint256 ytBalance // ytBalance_u
) internal {
    RewardIndex prevIndex = self[account].userIndex;
    uint256 accrued = FixedPointMathLib.mulWad(ytBalance, RewardIndex.unwrap(index - prevIndex)); //
    ↪  @audit user receives reward according to their yt balance
    self[account].accrued += accrued.toUint128();
    self[account].userIndex = index;
}
```

However, this formula fails to take into account the fact that the claimed YBT yield itself is entitled to rewards when a user collects their accrued YBT yield (we call them "claimed users"). Because any yield made by the principal token is paid back YT holders in the form of YBT, claiming any accrued YBT yield will decrease the PT's YBT balance, hence decreasing its own entitled rewards amount.

Because the PT's YBT balance decreased, the reward speed for the PT has also decreased. Furthermore, the reduced reward speed goes 100% to the claimed user because they now hold the reward-entitled YBT.

The end effect is that the user is entitled to more rewards than others by simply claiming their YBT interest.

Likelihood explanation: This is an inherent flaw in the math, and the attack can be triggered by anyone (willingly or unwillingly) easily by doing nothing more than just collecting their yield.

Impact explanation:

- Users can get an unfair advantage in the reward distribution, by having the claimed YBT itself generates rewards all for themselves, and still having the same YT shares in relation to the total YT supply.
- All other YT holders will receive reduced rewards speed as a result due to the PT holding less YBTs.

Proof of concept (textual): We provide two scenarios for comparison. In each scenario we use the same setting:

- We start at exchange rate 1.
- We end at exchange rate 2.
- The YBT's interest and reward amount are constant: 50 reward tokens are distributed for every +1 exchange rate.
- Scenario 1: No one claims any yield:
  - Let the initial exchange rate be 1. The PT pool holds 100 YBT tokens.
  - Alice and Bob holds 50 YT tokens each, so each holds 50% of the YT supply.
  - Exchange rate goes to 3. The 100 YBT is entitled to 100 reward tokens.
  - The YTs are entitled to 66.6 YBT, and 100 reward tokens are distributed to YT holders.

  Since Alice and Bob hold 50% of the YT shares, they are entitled to 33.3 YBT and 50 reward tokens each.
- Scenario 2: Alice claims early:
  - Let the initial exchange rate be 1. The PT pool holds 100 YBT tokens.
  - Alice and Bob holds 50 YT tokens each, so each holds 50% of the YT supply.
  - Exchange rate goes to 2. The 100 YBT is entitled to 50 reward tokens.
  - The YTs are entitled to 50 YBT, and 50 reward tokens are distributed to YT holders.
    * This amount is distributed 50/50 to YT holders. Alice and Bob are entitled to 25 YBT and 25 rewards each.

  Alice claims at this point. The balances at this point are:
  - Alice: 50 YT, 25 YBT, 25 reward.
    * Unclaimed: 0 YBT, 0 reward.
  - Bob: 50 YT, 0 YBT, 0 reward.
    * Unclaimed: 25 YBT, 25 reward.
  - PT holdings: 75 YBT, 25 reward.

  We continue on reaching the exchange rate of 3:
  - Exchange rate goes to 3. The 75 YBT left is entitled to 37.5 reward tokens.
  - The YTs are entitled to 16.66 YBT, and 37.5 reward tokens are distributed to YT holders.
    * This amount is distributed 50/50 to YT holders. Alice and Bob are entitled to 8.33 YBT and 18.75 rewards each.

  At this point:
  - Alice: 50 YT, 25 YBT, 25 reward.
    * Unclaimed: 8.33 YBT, 18.75 reward.
  - Bob: 50 YT, 0 YBT, 0 reward.
    * Unclaimed: 33.33 YBT, 43.75 reward.
  - PT holdings: 75 YBT, 62.5 reward.

Note the difference against the first case. Bob is still entitled to 33.33 YBT, but only 43.75 rewards in total, instead of 50. Furthermore, the total claimable rewards are only $25 + 18.75 + 43.75 = 87.5$, contrary to the 100 rewards earlier.

This is because the 25 claimed YBT by Alice also yields rewards. By taking away some of the YBT in the form of shares, Alice still has the same 50% shares of accrued reward, but also the same. In other words, Alice has less unclaimed YBT compared to others, but are still subject to the same rewards share.

- In fact, the 25 YBT claimed by Alice earlier has yielded the missing 12.5 rewards. Alice is entitled to 43.75 rewards from holding the YT, 12.5 rewards for claiming the interest YBT early and taking the rewards for herself, for a total of $43.75 + 12.5 = 56.25$ rewards.

Conclusion: Despite holding the same amount of YT for the same duration of time:

- The two scenarios yield different reward results.
- Alice has managed to get more yield than Bob (56.25 compared to Bob's 43.75).

Proof of concept (coded): We provide a coded proof of concept that shows the exact scenario in the textual PoC. Paste this test into `PoC.t.sol`.

Run with `forge test --match-test test_POC1 -vv` and `forge test --match-test test_POC2 -vv`.

There are two tests:.

- `test_POC1` showcasing scenario 1.
- `test_POC2` showcasing scenario 2, where Alice collects midway.

The output log for each test will be shown following the coded proof of concept.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {Base, ZapBase} from "./Base.t.sol";

import "src/Types.sol";
import "src/Constants.sol";

import {MockFeeModule} from "./mocks/MockFeeModule.sol";

import {FeePctsLib, FeePcts} from "src/utils/FeePctsLib.sol";

// PoC imports
import {console2} from "forge-std/src/Test.sol";
import {MockERC20} from "./mocks/MockERC20.sol";

/// @dev In order to further configure the principal token or Zap, refer to the
↪  `test/shared/PrincipalToken.t.sol`,`test/shared/Zap.t.sol` and `test/shared/Fork.t.sol`
contract POCTest is ZapBase {
    function setUp() public override {
        super.setUp();

        _deployTwoCryptoDeployer();
        _setUpModules();

        // There is 1 principal token deployed with bare minimum configuration:
        // - PrincipalToken using MockERC4626 as the underlying
        // - PrincipalToken is configured with DepositCapVerifierModule, MockFeeModule and
        ↪  MockRewardProxyModule
        // The MockERC4626 uses MockERC20 as a base asset.
        // The PrincipalToken can accumulate 2 mock reward tokens.
        _deployInstance();
        _deployPeriphery();

        // Overwrite ConstantFeeModule withMockFeeModule
        FeePcts feePcts = FeePctsLib.pack(5_000, 0, 0, 0, BASIS_POINTS); // For setting up principal token,
        ↪  issuance fee should be 0
        deployCodeTo("MockFeeModule", address(feeModule));
        setMockFeePcts(address(feeModule), feePcts);

        _label();
    }
```

```
function _label() internal virtual override {
    super._label();
    // Label user defined addresses for easier debugging (See vm.label())
}

address supplier; // @poc this will be the YT supplier for alice and bob

function setupPoC() internal {
    supplier = makeAddr("supplier");
    // alice = makeAddr("alice"); // already set up in the base test
    // bob = makeAddr("bob"); // already set up in the base test

    uint256 amountWETH = 100 * 1e18;

    vm.startPrank(supplier);
    deal(address(base), supplier, amountWETH);
    base.approve(address(target), amountWETH);
    uint256 shares = target.deposit(amountWETH, supplier);

    console2.log("Initial 4626 vault balance: %d", base.balanceOf(address(target))/1e18); // 100e18
    console2.log("Resolver scale: %d", resolver.scale()/1e18); // 1
    console2.log("Shares received: %d", shares/(1e18)); // 100 shares

    target.approve(address(principalToken), type(uint256).max);
    uint256 amountPT = principalToken.supply(shares, supplier);

    //console2.log("%d", amountPT/(1e18)); // 100 PT
    //console2.log("%d", yt.balanceOf(supplier)/(1e18)); // 100 YT

    yt.transfer(alice, 50e18);
    yt.transfer(bob, 50e18);

    vm.stopPrank();

    //console2.log("%d", yt.balanceOf(alice)/(1e18)); // 50 YT
    //console2.log("%d", yt.balanceOf(bob)/(1e18)); // 50 YT
}

/// @dev updates the reward per second towards PT/YT only
/// assuming that the reward rate per second towards all YBT in existence is 1e18
/// and that the rewards is distributed pro-rata towards all YBT holders
function updateRewardPerSecForYTs() internal {
    uint256 rewardsPerSecForAllYbt = 1 * 1e18;
    uint256 rewardsPerSecForYt = (rewardsPerSecForAllYbt * target.balanceOf(address(principalToken)))
        / (target.totalSupply());

    console2.log("New rewards per second towards YT holders: %d", rewardsPerSecForYt);
    multiRewardDistributor.setRewardsPerSec(rewardTokens[0], rewardsPerSecForYt);
}

function test_POC1() external {
    setupPoC();

    /*
        Simulate reward accrual

        PoC notes: Base test sets the reward accrual speed directly on the proxy
        instead of on the YBT
           - In other words, `setRewardsPerSec` actually sets the speed of rewards that go
        straight into the PT, instead of to all YBT holders. In real life, rewards are always
        directed towards all YBT holders.

        For this reason, after each yield collection, we have to call `setRewardsPerSec` to be:
            (total reward speed for all YBT) * (YBT held by the PT) / (YBT total supply)

        Of course with multiply before divide
    */

    updateRewardPerSecForYTs(); // 1e18
    skip(100); // 1e18 * 100 seconds = 100e18 rewards

    /*
        Simulate yield

        PoC notes: To simulate yield, we mint the base token directly to the target ERC4626
        To up the exchange rate to 3, we mint 2 times as much as our starting amount (100e18)
```

```
        */
        uint256 amountYield = (100 * 1e18) * 2;
        deal(address(base), address(target), amountYield + base.balanceOf(address(target))); // @PoC deal
        ↪  actually adjusts the balance and not increasing it
        console2.log("4626 vault balance after yield: %d", base.balanceOf(address(target))/1e18); // 300
        console2.log("Resolver scale after yield: %d", resolver.scale()/1e18);

        /*
            Both Alice and Bob collect now, see how much they are entitled to
            For simplicity, we only use reward tokens[0]
        */
        vm.startPrank(alice);
        principalToken.collect(alice, alice);
        console2.log("Alice total reward: %d", MockERC20(rewardTokens[0]).balanceOf(alice)); // 50e18
        console2.log("Alice total YBT yield: %d", target.balanceOf(alice)); // 33.33
        vm.stopPrank();

        // updateRewardPerSecForYTs();
        // this is not needed because we want to see how much bob gets as well right away
        // --> reward speed is irrelevant because no time passes

        vm.startPrank(bob);
        principalToken.collect(bob, bob);
        console2.log("Bob total reward: %d", MockERC20(rewardTokens[0]).balanceOf(bob)); // 50e18
        console2.log("Bob total YBT yield: %d", target.balanceOf(bob)); // 33.33
        vm.stopPrank();
    }

    function test_POC2() external {
        setupPoC();

        /*
            Same as PoC 1, but at timestamp 50 (half the timespan), we distribute half the yield
            and let Alice collect
        */
        updateRewardPerSecForYTs(); // 1e18
        skip(50); // 1e18 * 50 seconds = 50e18 rewards

        // distribute yield round 1
        uint256 amountYield = 100 * 1e18;
        deal(address(base), address(target), amountYield + base.balanceOf(address(target)));
        console2.log("4626 vault balance after yield round 1: %d", base.balanceOf(address(target))/1e18); //
        ↪  100e18
        console2.log("Resolver scale after yield round 1: %d", resolver.scale()/1e18);

        vm.startPrank(alice);
        principalToken.collect(alice, alice);
        console2.log("Alice claimed reward at 500: %d", MockERC20(rewardTokens[0]).balanceOf(alice)); // 30
        ↪  million wei
        console2.log("Alice claimed YBT yield at 500: %d", target.balanceOf(alice)); // 25
        vm.stopPrank();

        //console2.log("Bob reward: %d", MockERC20(rewardTokens[0]).balanceOf(bob));
        //console2.log("Bob yield: %d", target.balanceOf(bob));

        /*
            We skip a further 50 seconds, and distribute another round of yield
        */

        console2.log("POC: Before 50 seconds timeskip, Alice's YBT balance is", target.balanceOf(alice)); // 25
        updateRewardPerSecForYTs(); // only 0.75e18 now
        skip(50); // 0.75e18 * 50 seconds = 37.5e18 rewards
        console2.log("POC: Timeskipped 50 seconds. Alice's YBT balance", target.balanceOf(alice)); // 25
        uint256 rewardsPerSecForAllYbt = 1 * 1e18;
        uint256 rewardsPerSecForAliceYbt = (rewardsPerSecForAllYbt * target.balanceOf(alice))
            / (target.totalSupply());
        uint256 aliceAccruedRewards = 50 * rewardsPerSecForAliceYbt;
        console2.log("POC: By holding 25 YBT, i.e. 25% of the total supply, Alice is entitled to 25% of the
        ↪  rewards per second, which is: %d", rewardsPerSecForAliceYbt);
        console2.log("POC: By holding YBT for herself for 50 seconds, Alice gained an additional %d",
        ↪  aliceAccruedRewards);

        //uint256 amountYield = 100 * 1e18;
        deal(address(base), address(target), amountYield + base.balanceOf(address(target)));
        console2.log("4626 vault balance after yield round 2: %d", base.balanceOf(address(target))/1e18); //
        ↪  100e18
```

```
        console2.log("Resolver scale after yield round 2: %d", resolver.scale()/1e18); // yield accrual: +1
        ↪  exchange rate

        /*
            Both Alice and Bob collect now, see how much they finally gets
        */
        vm.startPrank(alice);
        principalToken.collect(alice, alice);
        console2.log("Alice final reward from YT: %d", MockERC20(rewardTokens[0]).balanceOf(alice)); // 60
        ↪  million wei
        console2.log("Alice final yield from YT: %d", target.balanceOf(alice)); // 33.33
        vm.stopPrank();

        vm.startPrank(bob);
        principalToken.collect(bob, bob);
        console2.log("Bob final reward from YT: %d", MockERC20(rewardTokens[0]).balanceOf(bob)); // 60 million
        ↪  wei
        console2.log("Bob final yield from YT: %d", target.balanceOf(bob)); // 33.33
        vm.stopPrank();

        console2.log("POC: Final Alice rewards, including those from YT and from holding the accrued YBT: %d",
            MockERC20(rewardTokens[0]).balanceOf(alice) + aliceAccruedRewards);
    }
}
```

The output will be:

- For `forge test --match-test test_POC1 -vv`, everyone gets 50 rewards, 33.33 YBT yield.

```
Logs:
    Initial 4626 vault balance: 100
    Resolver scale: 1
    Shares received: 100
    New rewards per second towards YT holders: 1000000000000000000
    4626 vault balance after yield: 300
    Resolver scale after yield: 3
    Alice total reward: 50000000000000000000
    Alice total YBT yield: 33333333333333333300
    Bob total reward: 50000000000000000000
    Bob total YBT yield: 33333333333333333300
```

- For `forge test --match-test test_POC2 -vv`, everyone gets 33.33 YBT yield, but Alice gets 56.25 rewards in total, where Bob only gets 43.75.

```
Initial 4626 vault balance: 100
Resolver scale: 1
Shares received: 100
New rewards per second towards YT holders: 1000000000000000000
4626 vault balance after yield round 1: 200
Resolver scale after yield round 1: 2
Alice claimed reward at 500: 25000000000000000000
Alice claimed YBT yield at 500: 25000000000000000000
POC: Before 50 seconds timeskip, Alice's YBT balance is 25000000000000000000
New rewards per second towards YT holders: 750000000000000000
POC: Timeskipped 50 seconds. Alice's YBT balance 25000000000000000000
POC: By holding 25 YBT, i.e. 25% of the total supply, Alice is entitled to 25% of the rewards per
↪  second, which is: 250000000000000000
POC: By holding YBT for herself for 50 seconds, Alice gained an additional 12500000000000000000
4626 vault balance after yield round 2: 300
Resolver scale after yield round 2: 3
Alice final reward from YT: 43750000000000000000
Alice final yield from YT: 33333333333333333300
Bob final reward from YT: 43750000000000000000
Bob final yield from YT: 33333333333333333300
POC: Final Alice rewards, including those from YT and from holding the accrued YBT: 56250000000000000000
```

Recommendation: The amount of rewards the YT holder is entitled to should not be proportional to their YT balance divided by the total YT supply at the time of reward claiming. Instead, it should be proportional to their **unclaimed YBT yield plus (YT balance divided by exchange rate) versus the total YBT held by the PT**, or in other words, the YBT amount that the YT is worth.

In the above example, the reward distribution at the second claim should be:

- Bob's entitled reward shares should be $(33.33 + (50/3))/75 = 0.66$ shares. For 37.5 rewards, that is

$37.5 * 0.66 = 25$, correctly entitling Bob to 50 tokens in total.

- Bob's entitled reward shares should be $(8.33 + (50/3))/75 = 0.33$ shares. For 37.5 rewards, that is $37.5 * 0.33 = 12.5$, correctly entitling Alice to 37.5 tokens in total.
- Combined with Alice's 12.5 rewards generated by the claimed YBT, Alice correctly receives to 50 rewards in total.

See Pendle's whitepaper for proof.

Napier:

> We think if everyone accrues and collects yield very frequently, then there is a baseline reward. In the current implementation, the baseline reward is still guaranteed. The findings say the rewards also needs to be split fairly.
>
> We set up some realistic scenarios to check the difference in final rewards amounts. The scenario was modified based on the scenario described in the report. We found that it's not very critical.

Christoph Michel: I agree with the sponsor. The "baseline" reward is from the YBT backing the principal which is the majority of the YBT (until maturity). This issue describes an inaccuracy in the reward distribution compared to the theoretical ideal. Both the impact and likelihood are overstated. In addition, the recommendation can't even be efficiently implemented in a contract as it would require iterating and updating all YT holders on each interaction because the new "reward-generating balance" `user_unclaimed_yield + (ptSupply / scale) * ytBalance / ytSupply` is not constant between updates.

3.2 Medium Risk

3.2.1 Wrong initial TwoCurve liquidity computation

Submitted by Christoph Michel

Severity: Medium Risk

Context: ZapMathLib.sol#L27

Description: When adding initial liqudity to the `TwoCurveCryptoNG` pool, the conversion in `ZapMathLib.computeSharesToTwoCrypto` tries to split the total `shares` into `sharesToPool` and `sharesToPt` that are turned into PT according to the initial pool price. However, its computation does not take into account different PT and target (vault) decimals. The following decimals are involved:

- PT decimals = asset decimals.
- vault (target) decimals.
- `TwoCryptoNG initialPrice` decimals: always 18 decimals and price is measured as YBT per PT.
- resolver's `scale`: always in `18 + asset_decimals - vault_decimals`.

```
uint256 principal = 10 ** principalToken.decimals();
uint256 underlyingReserveInPT = principalToken.previewSupply(initialPrice * principal / WAD);

// previewSupply(shares) -> principal expects YBT = target decimals as input.
// But initialPrice * principal / WAD is in principal = asset decimals.
```

Example:

- Asset decimals = PT decimals: 6 (USDC).
- Target decimals = vault decimals = 18 (Morpho USDC Vault).
- `scale` is in `18 + asset_decimals - target_decimals = 6` decimals.
- If scale is therefore `1.0 = 1e6`, then it would compute `underlyingReserveInPT = previewSupply(0.7e18 * 1e6 / 1e18 = 0.7e6) = 0.7e6 * scale / 1e18 = 0.7e6 * 1e6 / 1e18 = 0`.
- Zero shares are sent to the AMM and the `two_crypto.add_liquidity` call reverts.

Impact Explanation: Medium - One cannot add initial liquidity to a TwoCurve pool using the `TwoCryptoZap`. The functionality is broken. The contract needs to be redeployed.

Likelihood Explanation: High - Adding initial liquidity to the pool must happen for every PT deployment. Vaults with different decimals than their underlying are very common. For example, Morpho vaults all have 18 decimals, regardless of underlying decimals.

Recommendation: The conversion must take into account the decimal difference of PT (inheriting asset decimals) and target (vault) decimals. `previewSupply( 10**target_decimals * initialPrice * principal / WAD / 10**asset_decimals )`. This can be simplified to `previewSupply(10**target_decimals * initialPrice / WAD)`.

Proof of Concept: The proof of concept creates a vault with 18 decimals and asset's decimals are 6. It tries to use the `createAddLiquidity` zap function but it reverts.

```
forge test --mc PoCCreateAndcreateAndAddLiquidityTest --mt test_Deposit --decode-internal -vv
```

Patch the `MockERC4626` so the decimals and decimals-offset are in sync:

```
// Fixes the issue of hardcoding decimal offset to 18 - asset_decimals regardless of vault's decimals
contract MockERC4626DecimalsFixed is MockERC4626Decimals {
    constructor(ERC20 _asset, bool useVirtualShares, uint8 _decimals) MockERC4626Decimals(_asset,
    ↪  useVirtualShares, _decimals) {
        // fix decimals offset
        if (useVirtualShares) {
            i_decimalsOffset = _decimals - i_underlyingDecimals;
        }
    }
}
```

Add this file as `repo/test/PoCCreateAndAddLiquidity.t.sol`:

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.10;

import "forge-std/src/Test.sol";

import {Base} from "./Base.t.sol";
import {TwoCryptoZapAMMTest} from "./shared/Zap.t.sol";

import {SafeTransferLib} from "solady/src/utils/SafeTransferLib.sol";

import {Factory} from "src/zap/TwoCryptoZap.sol";
import {TwoCryptoZap} from "src/zap/TwoCryptoZap.sol";
import {PrincipalToken} from "src/tokens/PrincipalToken.sol";
import {LibTwoCryptoNG} from "src/utils/LibTwoCryptoNG.sol";
import {FeePctsLib, FeePcts} from "src/utils/FeePctsLib.sol";
import {Errors} from "src/Errors.sol";
import "src/Types.sol";
import "src/Constants.sol";

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

import "forge-std/src/console.sol";

contract PoCCreateAndcreateAndAddLiquidityTest is TwoCryptoZapAMMTest {
    using LibTwoCryptoNG for TwoCrypto;

    function setUp() public override {
        Base.setUp();
        _deployTwoCryptoDeployer();
        _setUpModules();
        // No need to deploy instances
        _deployPeriphery();

        _label();
    }

    function getParams(uint256 shares) public view returns (TwoCryptoZap.CreateAndAddLiquidityParams memory
    ↪  params) {
        FeePcts feePcts = FeePctsLib.pack(5_000, 310, 100, 830, 2183);

        // @audit-info twocryptoParams always uses 0.7e18 price
```

```
        bytes memory poolArgs = abi.encode(twocryptoParams);
        bytes memory resolverArgs = abi.encode(address(target)); // Add appropriate resolver args if needed
        Factory.ModuleParam[] memory moduleParams = new Factory.ModuleParam[](1);
        moduleParams[0] = Factory.ModuleParam({
            moduleType: FEE_MODULE_INDEX,
            implementation: constantFeeModule_logic,
            immutableData: abi.encode(feePcts)
        });
        Factory.Suite memory suite = Factory.Suite({
            accessManagerImpl: address(accessManager_logic),
            resolverBlueprint: address(resolver_blueprint),
            ptBlueprint: address(pt_blueprint),
            poolDeployerImpl: address(twocryptoDeployer),
            poolArgs: poolArgs,
            resolverArgs: resolverArgs
        });

        params = TwoCryptoZap.CreateAndAddLiquidityParams({
            suite: suite,
            modules: moduleParams,
            expiry: expiry,
            curator: curator,
            shares: shares,
            minLiquidity: 0, // @audit-info should accept all swaps, so reverting is bug
            minYt: 0,
            deadline: block.timestamp
        });
    }

    function _setInstances(address twoCrypto) internal {
        twocrypto = TwoCrypto.wrap(twoCrypto);
        principalToken = PrincipalToken(twocrypto.coins(PT_INDEX));
        yt = principalToken.i_yt();
        resolver = principalToken.i_resolver();
        accessManager = principalToken.i_accessManager();
    }

    function test_Deposit() public {
        uint256 shares = 10 * 10 ** target.decimals();

        deal(address(target), alice, shares);

        _test_Deposit(alice, shares);
    }

    function _test_Deposit(address caller, uint256 shares) internal {
        shares = bound(shares, 0, SafeTransferLib.balanceOf(address(target), caller));
        _approve(target, caller, address(zap), shares);

        TwoCryptoZap.CreateAndAddLiquidityParams memory params = getParams(shares);

        vm.prank(caller);
        (bool s, bytes memory ret) = address(zap).call(abi.encodeCall(zap.createAndAddLiquidity, (params)));
        vm.assume(s);
        ( /* address pt */ , /* address yt */, address twoCrypto, uint256 liquidity, uint256 principal) =
            abi.decode(ret, (address, address, address, uint256, uint256));
        console.log("_test_Deposit: liquidity %18ee18", liquidity);

        _setInstances(twoCrypto);

        assertEq(twocrypto.balanceOf(alice), liquidity, "liquidity balance");
        assertEq(yt.balanceOf(alice), principal, "yt balance");
        assertNoFundLeft();
        assertEq(accessManager.owner(), curator, "Owner is curator");
    }

    // @audit-info custom overrides
    function _deployTokens() internal virtual override {
        randomToken = new MockERC20({_decimals: 6});
        base = new MockERC20({_decimals: 6});
        // @audit-info can go up from 6 to 9 decimals before reverting. the liquidity received will go down
        ↪  from 4e18 -> 0.75e18
        target = new MockERC4626DecimalsFixed({_asset: base, useVirtualShares: true, _decimals: 18 });
        console.log("target.decimals() %s", target.decimals());
    }
}
```

3.2.2 Collect front run permit

Submitted by TamayoNft, also found by typicalHuman, Christoph Michel and KupiaSec

Severity: Medium Risk

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

The principal token is exposing a function call `permitCollector` to allow an owner to approve a collector. This function is receiving a signature an setting the collector to the signer of the signature.

```
function permitCollector(address owner, address collector, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
    external
{
    assembly {
        // Revert if the block timestamp is greater than `deadline`.
        if gt(timestamp(), deadline) {
            mstore(0x00, 0x1a15a3cc) // `PermitExpired()`.
            revert(0x1c, 0x04)
        }
    }

    uint256 nonce = nonces(owner);
    bytes32 domainSeparator = DOMAIN_SEPARATOR();

    bytes32 digest;
    /// @solidity memory-safe-assembly
    assembly {
        // Dev: Forked from Solady's ERC20 permit and EIP712 hashTypedData implementation.
        ...
    }

    address signer = ECDSA.recover(digest, v, r, s);
    if (signer != owner) InvalidPermit.selector.revertWith();

    // WRITE
    assembly {  <----
        // Compute the nonce slot and increment the nonce without overflow check.
        mstore(0x0c, _NONCES_SLOT_SEED)
        mstore(0x00, owner)
        sstore(keccak256(0x0c, 0x20), add(nonce, 1))
    }
    _setApprovalCollector(owner, collector, true);
}
```

As you can see in the arrow above the nonce have being increased each time that the `permitCollector` is call this is a common practice to protect against replay attacks.

For other side we can see the `TwoCryptoZap` as a router to communicate with the pricipalToken. the `TwoCryptoZap` expose the `collectWithPermit` function to set up the `TwoCryptoZap` as a collector for a owner.

```
function collectWithPermit(CollectInput[] calldata inputs, address receiver) external nonReentrant {
    uint256 length = inputs.length;
    for (uint256 i = 0; i != length;) {
        CollectInput calldata input = inputs[i];
        ContractValidation.checkPrincipalToken(i_factory, input.principalToken);

        _permitCollector(input, msg.sender); <----
        PrincipalToken(input.principalToken).collect(receiver, msg.sender);
        unchecked {
            ++i;
        }
    }
}
```

```
function _permitCollector(CollectInput calldata input, address owner) internal {
    if (input.permit.deadline > 0) {
        PrincipalToken(input.principalToken).permitCollector(
            owner, address(this), input.permit.deadline, input.permit.v, input.permit.r, input.permit.s
        );
    }
}
```

The problem is that if a user or contract call `collectWithPermit` in the `TwoCryptoZap` an attacker can just take the signature and call directly `permitCollector` in the principalToken making revert the `collectWithPermit` function because the nonce already increase. This will temporary DoS the `collectWithPermit` meanwhile the user figurate why the call is reverting. Note that this specially problematic for contracts that interact directly with the `TwoCryptoZap`.

Impact Explanation: `collectWithPermit` can be DoS temporally because a front run permit issue. This is specially problematic for contracts that interact with the `TwoCryptoZap`.

Proof of Concept: Run the next proof of concep in `file:/test/zap/BatchCollect.t.sol`.

```
function test_Front_run_permit() public {
    //@audit
    PrincipalToken[] memory pts = new PrincipalToken[](1);
    pts[0] = principalToken;
    TwoCryptoZap.CollectInput[] memory inputs = new TwoCryptoZap.CollectInput[](pts.length);

    // Permit directly
    bool[] memory skips = new bool[](pts.length);
    skips[0] = true;

    _Temp memory t = _Temp({
        privateKey: wallet.privateKey,
        owner: wallet.addr,
        nonce: principalToken.nonces(wallet.addr),
        deadline: block.timestamp,
        v: 0,
        r: bytes32(0),
        s: bytes32(0)
    });
    _signPermit(t);

    TwoCryptoZap.PermitCollectInput memory permit =
        TwoCryptoZap.PermitCollectInput({deadline: t.deadline, v: t.v, r: t.r, s: t.s});
    inputs[0] = TwoCryptoZap.CollectInput({principalToken: address(pts[0]), permit: permit});

    principalToken.permitCollector(t.owner, address(zap), t.deadline, t.v, t.r, t.s);// front running the
    ↪  permit call

    vm.expectRevert();
    zap.collectWithPermit(inputs, address(this));//this call will revert because  InvalidPermit()
}
```

As you can see the zap call is reverting for InvalidPermit() error.

This is a well know vulnerability (see the permission denied post py trust security).

Recommendation: The solution is try an catch the call made to the principal token (as shown in the aforementioned post).