Strata PreDeposit Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Dacian

Giovanni Di Siena

June 11, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Strata is a perpetual yield tranching protocol built on Converge, designed to offer structured yield exposure on USDe, Ethena's delta-neutral synthetic stablecoin. This audit covers the "PreDeposit" functionality primarily related to these two contracts:

- `pUSDeVault` : a two-phase, multi-asset ERC4626 vault
- `pUSDeDepositor` : a "front-end" or "helper" contract implementing easy deposit options into `pUSDeVault`

The 3 other contracts in the audit scope `MetaVault`, `PreDepositVault`, `PreDepositPhaser` are simply part of the inheritance chain of `pUSDeVault`. The protocol interacts with the following tokens:

- USDe : ERC20, Ethena's synthetic USD token
- sUSDe : ERC4626, yield-bearing equivalent of `USDe`, staked into an ERC4626 vault

pUSDeVault

`pUSDeVault` is an upgradeable multi-asset ERC4626 vault which can operate in one of two phases:

- Points Phase - accepts and holds `USDe` as the ERC4626 underlying asset, and potentially also other assets allowed by the contract owner
- Yield Phase - initially when activated by the owner via `startYieldPhase`, redeems all additionally supported assets then stakes the entire `USDe` balance by depositing it into the `sUSDe` ERC4626 vault. Once activated the Yield Phase is permanent, `sUSDe` becomes a supported asset and any future `USDe` deposits are automatically staked in the same way.

pUSDeDepositor

`pUSDeDepositor` is an upgradeable utility contract which allows users to deposit into `pUSDeVault` using:

- `USDe` - the simplest option to deposit the primary underlying ERC20 asset of `pUSDeVault`
- `sUSDe` - shares from `sUSDe` which is used as the staking vault for `pUSDeVault`'s `USDe`, only during the yield phase

- `autoSwap to USDe` - other stablecoin tokens allowed by the owner are swapped for `USDe`, then swap output `USDe` is deposited into `pUSDeVault` in one transaction
- `supported vaults` - shares from other supported ERC4626 vaults

Centralization

The contracts have an owner and are upgradeable; users interacting with the protocol must have complete trust in the protocol team. The owner has the ability to call the following special functions:

- `updateYUSDeVault` - updates yUSDe vault address (note contracts associated with yUSDe were not part of this audit)
- `startYieldPhase` - begins the yield phase converting all supporting vault deposits into `USDe` and staking all `USDe` by depositing into `sUSDe`
- `updateSwapInfo` - swap routing information but the `autoSwap to USDe` deposit method
- `setDepositsEnabled`, `setWithdrawalsEnabled` - enable or disable deposits and withdrawals; owner can prevent users from withdrawing
- `addVault`, `removeVault` - add or remove additionally supported vaults

5 Audit Scope

The scope of this audit is limited to:

```
strata-money-contracts/contracts/predeposit/MetaVault.sol
strata-money-contracts/contracts/predeposit/PreDepositPhaser.sol
strata-money-contracts/contracts/predeposit/PreDepositVault.sol
strata-money-contracts/contracts/predeposit/pUSDeDepositor.sol
strata-money-contracts/contracts/predeposit/pUSDeVault.sol
```

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the Strata PreDeposit smart contracts provided by Strata. In this period, a total of 38 issues were found.

The findings consist of 1 Critical, 1 High, 3 Medium and 16 Low severity issues with the remainder being gas optimizations and informational.

- 1 Critical allowed an attacker to drain `sUSDe` protocol balance during the yield phase
- 1 High was a niche edge case where during the yield phase, when supporting vaults were enabled and being used, a state could arise where users couldn't withdraw vault assets they were entitled to
- 2 Mediums concerned the `MetaVault::redeemRequiredBaseAssets` function which did not work as intended and could result in several incorrect edge-case behaviors with negative consequences. 1 Medium was a rounding issue that would leak value from `yUSDe` depositors to `pUSDe` redeemers
- 16 Lows were a wide variety of incorrectly handled edge cases and ERC4626 specification violations but with low probability and impact

The Critical finding was related to the interaction between in-scope and out-of-scope components during the yield phase, and was found outside of the allotted time for the audit. Prior to using the yield phase in production deployment we recommend another audit with all files in scope.

Code & Test Suite Analysis

The code quality was generally good though at times due to the inheritance heirarchy it can be confusing to trace through execution flows, for example in the different overrides of functions related to deposits and withdrawals and conditional execution paths within the overrides.

The protocol did have a typescript-based hardhat test suite and added a Foundry test harness at our request for the audit. We extended the Foundry test harness to:

- add a number of targeted test cases including some stateless fuzz tests and PoCs for our findings
- wrote an invariant fuzz testing suite

The protocol committed our tests to their repository. We encourage the protocol to continue adding tests to our Foundry test suite as this supports advanced features such as fuzz and invariant testing.

Summary

Project Name	Strata PreDeposit
Repository	contracts
Commit	e053c804f538...
Audit Timeline	May 26th - May 28th, 2025
Methods	Manual Review, Fuzz/Invariant Testing

Issues Found

Critical Risk	1
High Risk	1
Medium Risk	3
Low Risk	16
Informational	9
Gas Optimizations	8
Total Issues	38

Summary of Findings

[C-1] An attacker can drain the entire protocol balance of sUSDe during the yield phase due to incorrect redemption accounting logic in `pUSDeVault::_withdraw`	Resolved
[H-1] During the yield phase, when using supported vaults, users can't withdraw vault assets they are entitled to	Resolved
[M-1] `MetaVault::redeemRequiredBaseAssets` should be able to redeem small amounts from each vault to fill requested amount and avoid redeeming more than requested	Resolved
[M-2] DoS of meta vault withdrawals during points phase if one vault is paused or attempted redemption exceeds the maximum	Resolved
[M-3] Value leakage due to pUSDe redemptions rounding against the protocol/yUSDe depositors	Resolved

[L-01] Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision	Resolved
[L-02] In `pUSDeDepositor::deposit_viaSwap`, using `block.timestamp` in swap deadline is not very effective	Resolved
[L-03] Hard-coded slippage in `pUSDeDepositor::deposit_viaSwap` can lead to denial of service	Resolved
[L-04] Use `SafeERC20::forceApprove` instead of standard `IERC20::approve`	Resolved
[L-05] `MetaVault::redeem` erroneously calls `ERC4626Upgradeable::withdraw` when attempting to redeem `USDe` from `pUSDeVault`	Resolved
[L-06] Duplicate vaults can be pushed to `assetsArr`	Resolved
[L-07] `MetaVault::addVault` should enforce identical underlying base asset	Resolved
[L-08] `pUSDeVault::startYieldPhase` should not remove supported vaults from being supported or should prevent new supported vaults once in the yield phase	Resolved
[L-09] No way to compound deposited supported vault assets into `sUSDe` stake during yield phase	Resolved
[L-10] `pUSDeVault::maxWithdraw` doesn't account for withdrawal pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`	Resolved
[L-11] `pUSDeVault::maxDeposit` doesn't account for deposit pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`	Resolved
[L-12] `pUSDeVault::maxMint` doesn't account for mint pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`	Resolved
[L-13] `pUSDeVault::maxRedeem` doesn't account for redemption pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`	Resolved
[L-14] `yUSDeVault` inherits from `PreDepositVault` but doesn't call `onAfterDepositChecks` or `onAfterWithdrawalChecks`	Resolved
[L-15] Inability to remove and redeem from vaults with withdrawal issues could result in a bank-run	Resolved
[L-16] `yUSDeVault` edge cases should be explicitly handled to prevent view functions from reverting	Resolved
[I-1] Use named mappings to explicitly denote the purpose of keys and values	Resolved
[I-2] Disable initializers on upgradeable contracts	Resolved
[I-3] Don't initialize to default values	Resolved
[I-4] Use explicit sizes instead of `uint`	Resolved
[I-5] Prefix internal and private function names with _ character	Resolved
[I-6] Use unchained initializers instead	Resolved
[I-7] Missing zero deposit amount validation	Resolved
[I-8] `PreDepositVault::initialize` should not be exposed as public	Resolved
[I-9] Inconsistency in `currentPhase` between `pUSDeVault` and `yUSDeVault`	Resolved
[G-1] Cache identical storage reads	Resolved

[G-2] Using `calldata` is more efficient to `memory` for read-only external function inputs	Acknowledged
[G-3] Use named returns where this can eliminate in-function variable declaration	Resolved
[G-4] Inline small internal functions only used once	Resolved
[G-5] `PreDepositVault` checks should fail early	Acknowledged
[G-6] Superfluous vault support validation can be removed from `pUSDeDepositor::deposit`	Resolved
[G-7] Remove unused return value from `pUSDeVault::stakeUSDe` and explicitly revert if `USDeAssets == 0`	Resolved
[G-8] Unnecessarily complex iteration logic in `MetaVault::redeemMetaVaults` can be simplified	Resolved

7 Findings

7.1 Critical Risk

7.1.1 An attacker can drain the entire protocol balance of sUSDe during the yield phase due to incorrect redemption accounting logic in `pUSDeVault::_withdraw`

Description: After transitioning to the yield phase, the entire protocol balance of USDe is deposited into sUSDe and pUSDe can be deposited into the yUSDe vault to earn additional yield from the sUSDe. When initiating a redemption, `yUSDeVault::_withdraw` is called which in turn invokes `pUSDeVault::redeem`:

```
    function _withdraw(address caller, address receiver, address owner, uint256 pUSDeAssets, uint256
    ↪   shares) internal override {
        if (!withdrawalsEnabled) {
            revert WithdrawalsDisabled();
        }

        if (caller != owner) {
            _spendAllowance(owner, caller, shares);
        }


        _burn(owner, shares);
@>      pUSDeVault.redeem(pUSDeAssets, receiver, address(this));
        emit Withdraw(caller, receiver, owner, pUSDeAssets, shares);
    }
```

This is intended to have the overall effect of atomically redeeming yUSDe -> pUSDe -> sUSDe by previewing and applying any necessary yield from sUSDe:

```
    function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
    ↪   internal override {

            if (PreDepositPhase.YieldPhase == currentPhase) {
                // sUSDeAssets = sUSDeAssets + user_yield_sUSDe
@>              assets += previewYield(caller, shares);

@>              uint sUSDeAssets = sUSDe.previewWithdraw(assets); // @audit - this rounds up because
↪   sUSDe requires the amount of sUSDe burned to receive assets amount of USDe to round up, but below
↪   we are transferring this rounded value out to the receiver which actually rounds against the
↪   protocol/yUSDe depositors!

                _withdraw(
                    address(sUSDe),
                    caller,
                    receiver,
                    owner,
                    assets, // @audit - this should not include the yield, since it is decremented from
                    ↪   depositedBase
                    sUSDeAssets,
                    shares
                );
                return;
            }
        ...
    }
```

However, by incrementing `assets` in the case where this is a yUSDe redemption and there has been yield accrued by sUSDe, this will attempt to decrement the `depositedBase` state by more than intended:

```
    function _withdraw(
            address token,
            address caller,
            address receiver,
            address owner,
            uint256 baseAssets,
            uint256 tokenAssets,
            uint256 shares
        ) internal virtual {
            if (caller != owner) {
                _spendAllowance(owner, caller, shares);
            }
@>          depositedBase -= baseAssets; // @audit - this can underflow when redeeming yUSDe because
↪   previewYield() increments assets based on sUSDe preview but this decrement should be equivalent to
↪   the base asset amount that is actually withdrawn from the vault (without yield)

            _burn(owner, shares);
            SafeERC20.safeTransfer(IERC20(token), receiver, tokenAssets);
            onAfterWithdrawalChecks();

            emit Withdraw(caller, receiver, owner, baseAssets, shares);
            emit OnMetaWithdraw(receiver, token, tokenAssets, shares);
        }
```

If the incorrect state update results in an unexpected underflow then yUSDe depositors may be unable to redeem their shares (principal + yield). However, if a faulty yUSDe redemption is processed successfully (i.e. if the relative amount of USDe underlying pUSDe is sufficiently large compared to the total supply of yUSDe and the corresponding sUSDe yield) then pUSDe depositors will erroneously and unexpectedly redeem their shares for significantly less USDe than they originally deposited. This effect will be magnified by subsequent yUSDe redemptions as the `total_yield_USDe` will be computed as larger than it is in reality due to `depositedBase` being much smaller than it should be:

```
    function previewYield(address caller, uint256 shares) public view virtual returns (uint256) {
        if (PreDepositPhase.YieldPhase == currentPhase && caller == address(yUSDe)) {
            uint total_sUSDe = sUSDe.balanceOf(address(this));
            uint total_USDe = sUSDe.previewRedeem(total_sUSDe);

@>          uint total_yield_USDe = total_USDe - Math.min(total_USDe, depositedBase);
            uint y_pUSDeShares = balanceOf(caller);

            uint caller_yield_USDe = total_yield_USDe.mulDiv(shares, y_pUSDeShares,
            ↪  Math.Rounding.Floor);

            return caller_yield_USDe;
        }
        return 0;
    }
```

This in turn causes `depositedBase` to be further decremented until it is eventually tends to zero, impacting all functionality that relies of the overridden `totalAssets()`. Given that it is possible to inflate the sUSDe yield by either transferring USDe directly or waiting to sandwich a legitimate yield accrual (since `sUSDe::previewRedeem` does not account for the vesting schedule) this allows an attacker to completely devastate the pUSDe/yUSDe accounting, redeeming their yUSDe for close to the entire protocol sUSDe balance at the expense of all other depositors.

Impact: Significant loss of user funds.

Proof of Concept:

```
pragma solidity 0.8.28;

import {Test} from "forge-std/Test.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {MockUSDe} from "../contracts/test/MockUSDe.sol";
import {MockStakedUSDe} from "../contracts/test/MockStakedUSDe.sol";
import {MockERC4626} from "../contracts/test/MockERC4626.sol";

import {pUSDeVault} from "../contracts/predeposit/pUSDeVault.sol";
import {yUSDeVault} from "../contracts/predeposit/yUSDeVault.sol";

import {console2} from "forge-std/console2.sol";

contract CritTest is Test {
    uint256 constant MIN_SHARES = 0.1 ether;

    MockUSDe public USDe;
    MockStakedUSDe public sUSDe;
    pUSDeVault public pUSDe;
    yUSDeVault public yUSDe;

    address account;

    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    function setUp() public {
        address owner = msg.sender;

        // Prepare Ethena and Ethreal contracts
        USDe = new MockUSDe();
        sUSDe = new MockStakedUSDe(USDe, owner, owner);

        // Prepare pUSDe and Depositor contracts
        pUSDe = pUSDeVault(
            address(
                new ERC1967Proxy(
                    address(new pUSDeVault()),
                    abi.encodeWithSelector(pUSDeVault.initialize.selector, owner, USDe, sUSDe)
                )
            )
        );

        yUSDe = yUSDeVault(
            address(
                new ERC1967Proxy(
                    address(new yUSDeVault()),
                    abi.encodeWithSelector(yUSDeVault.initialize.selector, owner, USDe, sUSDe, pUSDe)
                )
            )
        );

        vm.startPrank(owner);
        pUSDe.setDepositsEnabled(true);
        pUSDe.setWithdrawalsEnabled(true);
        pUSDe.updateYUSDeVault(address(yUSDe));

        // deposit USDe and burn minimum shares to avoid reverting on redemption
        uint256 initialUSDeAmount = pUSDe.previewMint(MIN_SHARES);
        USDe.mint(owner, initialUSDeAmount);
        USDe.approve(address(pUSDe), initialUSDeAmount);
        pUSDe.mint(MIN_SHARES, address(0xdead));
        vm.stopPrank();

        if (pUSDe.balanceOf(address(0xdead)) != MIN_SHARES) {
            revert("address(0xdead) should have MIN_SHARES shares of pUSDe");
        }
    }

    function test_crit() public {
        uint256 aliceDeposit = 100 ether;
        uint256 bobDeposit = 2 * aliceDeposit;

        // fund users
        USDe.mint(alice, aliceDeposit);
        USDe.mint(bob, bobDeposit);

        // alice deposits into pUSDe
        vm.startPrank(alice);
        USDe.approve(address(pUSDe), aliceDeposit);
        uint256 aliceShares_pUSDe = pUSDe.deposit(aliceDeposit, alice);
        vm.stopPrank();

        // bob deposits into pUSDe
        vm.startPrank(bob);
        USDe.approve(address(pUSDe), bobDeposit);
        uint256 bobShares_pUSDe = pUSDe.deposit(bobDeposit, bob);
        vm.stopPrank();

        // setup assertions
        assertEq(pUSDe.balanceOf(alice), aliceShares_pUSDe, "Alice should have shares equal to her
        ↪  deposit");
        assertEq(pUSDe.balanceOf(bob), bobShares_pUSDe, "Bob should have shares equal to his deposit");

        {
            // phase change
            account = msg.sender;
            uint256 initialAdminTransferAmount = 1e6;
            vm.startPrank(account);
            USDe.mint(account, initialAdminTransferAmount);
            USDe.approve(address(pUSDe), initialAdminTransferAmount);
            pUSDe.deposit(initialAdminTransferAmount, address(yUSDe));
            pUSDe.startYieldPhase();
            yUSDe.setDepositsEnabled(true);
            yUSDe.setWithdrawalsEnabled(true);
            vm.stopPrank();
        }

        // bob deposits into yUSDe
        vm.startPrank(bob);
        pUSDe.approve(address(yUSDe), bobShares_pUSDe);
        uint256 bobShares_yUSDe = yUSDe.deposit(bobShares_pUSDe, bob);
        vm.stopPrank();

        // simulate sUSDe yield transfer
        uint256 sUSDeYieldAmount = 100 ether;
        USDe.mint(address(sUSDe), sUSDeYieldAmount);

        {
            // bob redeems from yUSDe
            uint256 bobBalanceBefore_sUSDe = sUSDe.balanceOf(bob);
            vm.prank(bob);
            yUSDe.redeem(bobShares_yUSDe/2, bob, bob);
            uint256 bobRedeemed_sUSDe = sUSDe.balanceOf(bob) - bobBalanceBefore_sUSDe;
            uint256 bobRedeemed_USDe = sUSDe.previewRedeem(bobRedeemed_sUSDe);

            console2.log("Bob redeemed sUSDe (1): %s", bobRedeemed_sUSDe);
            console2.log("Bob} redeemed USDe (1): %s", bobRedeemed_USDe);

            // bob can redeem again
            bobBalanceBefore_sUSDe = sUSDe.balanceOf(bob);
            vm.prank(bob);
            yUSDe.redeem(bobShares_yUSDe/5, bob, bob);
            uint256 bobRedeemed_sUSDe_2 = sUSDe.balanceOf(bob) - bobBalanceBefore_sUSDe;
            uint256 bobRedeemed_USDe_2 = sUSDe.previewRedeem(bobRedeemed_sUSDe);

            console2.log("Bob redeemed sUSDe (2): %s", bobRedeemed_sUSDe_2);
            console2.log("Bob redeemed USDe (2): %s", bobRedeemed_USDe_2);

            // bob redeems once more
            bobBalanceBefore_sUSDe = sUSDe.balanceOf(bob);
            vm.prank(bob);
            yUSDe.redeem(bobShares_yUSDe/6, bob, bob);
            uint256 bobRedeemed_sUSDe_3 = sUSDe.balanceOf(bob) - bobBalanceBefore_sUSDe;
            uint256 bobRedeemed_USDe_3 = sUSDe.previewRedeem(bobRedeemed_sUSDe);

            console2.log("Bob redeemed sUSDe (3): %s", bobRedeemed_sUSDe_3);
            console2.log("Bob redeemed USDe (3): %s", bobRedeemed_USDe_3);
        }

        console2.log("pUSDe balance of sUSDe after bob's redemptions: %s",
        ↪   sUSDe.balanceOf(address(pUSDe)));
        console2.log("pUSDe depositedBase after bob's redemptions: %s", pUSDe.depositedBase());

        // alice redeems from pUSDe
        uint256 aliceBalanceBefore_sUSDe = sUSDe.balanceOf(alice);
        vm.prank(alice);
        uint256 aliceRedeemed_USDe_reported = pUSDe.redeem(aliceShares_pUSDe, alice, alice);
        uint256 aliceRedeemed_sUSDe = sUSDe.balanceOf(alice) - aliceBalanceBefore_sUSDe;
        uint256 aliceRedeemed_USDe = sUSDe.previewRedeem(aliceRedeemed_sUSDe);

        console2.log("Alice redeemed sUSDe: %s", aliceRedeemed_sUSDe);
        console2.log("Alice redeemed USDe: %s", aliceRedeemed_USDe);
        console2.log("Alice lost %s USDe", aliceDeposit - aliceRedeemed_USDe);

        // uncomment to observe the assertion fail
        // assertApproxEqAbs(aliceRedeemed_USDe, aliceDeposit, 10, "Alice should redeem approximately
        ↪   her deposit in USDe");
    }
}
```

Recommended Mitigation: While the assets corresponding to the accrued yield should be included when previewing the sUSDe withdrawal, only the base assets should be passed to the subsequent call to `_withdraw()`:

```
function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
↪   internal override {

        if (PreDepositPhase.YieldPhase == currentPhase) {
            // sUSDeAssets = sUSDeAssets + user_yield_sUSDe
--          assets += previewYield(caller, shares);
++          uint256 assetsPlusYield = assets + previewYield(caller, shares);

--          uint sUSDeAssets = sUSDe.previewWithdraw(assets);
++          uint sUSDeAssets = sUSDe.previewWithdraw(assetsPlusYield);

```

```
            _withdraw(
                address(sUSDe),
                caller,
                receiver,
                owner,
                assets
                sUSDeAssets,
                shares
            );
            return;
        }
    ...
}
```

Strata: Fixed in commit 903d052.

Cyfrin: Verified. Yield is no longer included within the decremented assets amount and the test now passes with the assertion included.

7.2 High Risk

7.2.1 During the yield phase, when using supported vaults, users can't withdraw vault assets they are entitled to

Description: During the yield phase, when using supported vaults, users can't withdraw vault assets they are entitled to.

Proof of Concept:

```
function test_yieldPhase_supportedVaults_userCantWithdrawVaultAssets() external {
    // user1 deposits $1000 USDe into the main vault
    uint256 user1AmountInMainVault = 1000e18;
    USDe.mint(user1, user1AmountInMainVault);

    vm.startPrank(user1);
    USDe.approve(address(pUSDe), user1AmountInMainVault);
    uint256 user1MainVaultShares = pUSDe.deposit(user1AmountInMainVault, user1);
    vm.stopPrank();

    assertEq(pUSDe.totalAssets(), user1AmountInMainVault);
    assertEq(pUSDe.balanceOf(user1), user1MainVaultShares);

    // admin triggers yield phase on main vault which stakes all vault's USDe
    pUSDe.startYieldPhase();
    // totalAssets() still returns same amount as it is overridden in pUSDeVault
    assertEq(pUSDe.totalAssets(), user1AmountInMainVault);
    // balanceOf shows pUSDeVault has deposited its USDe in sUSDe
    assertEq(USDe.balanceOf(address(pUSDe)), 0);
    assertEq(USDe.balanceOf(address(sUSDe)), user1AmountInMainVault);

    // create an additional supported ERC4626 vault
    MockERC4626 newSupportedVault = new MockERC4626(USDe);
    pUSDe.addVault(address(newSupportedVault));
    // add eUSDe again since `startYieldPhase` removes it
    pUSDe.addVault(address(eUSDe));

    // verify two additional vaults now suppported
    assertTrue(pUSDe.isAssetSupported(address(eUSDe)));
    assertTrue(pUSDe.isAssetSupported(address(newSupportedVault)));

    // user2 deposits $600 into each vault
    uint256 user2AmountInEachSubVault = 600e18;
    USDe.mint(user2, user2AmountInEachSubVault*2);

    vm.startPrank(user2);
    USDe.approve(address(eUSDe), user2AmountInEachSubVault);
    uint256 user2SubVaultSharesInEach = eUSDe.deposit(user2AmountInEachSubVault, user2);
    USDe.approve(address(newSupportedVault), user2AmountInEachSubVault);
    newSupportedVault.deposit(user2AmountInEachSubVault, user2);
    vm.stopPrank();

    // verify balances correct
    assertEq(eUSDe.totalAssets(), user2AmountInEachSubVault);
    assertEq(newSupportedVault.totalAssets(), user2AmountInEachSubVault);

    // user2 deposits using their shares via MetaVault::deposit
    vm.startPrank(user2);
    eUSDe.approve(address(pUSDe), user2SubVaultSharesInEach);
    pUSDe.deposit(address(eUSDe), user2SubVaultSharesInEach, user2);
    newSupportedVault.approve(address(pUSDe), user2SubVaultSharesInEach);
    pUSDe.deposit(address(newSupportedVault), user2SubVaultSharesInEach, user2);
    vm.stopPrank();

    // verify main vault total assets includes everything
    assertEq(pUSDe.totalAssets(), user1AmountInMainVault + user2AmountInEachSubVault*2);
    // main vault not carrying any USDe balance
    assertEq(USDe.balanceOf(address(pUSDe)), 0);
    // user2 lost their subvault shares
    assertEq(eUSDe.balanceOf(user2), 0);
    assertEq(newSupportedVault.balanceOf(user2), 0);
    // main vault gained the subvault shares
    assertEq(eUSDe.balanceOf(address(pUSDe)), user2SubVaultSharesInEach);
    assertEq(newSupportedVault.balanceOf(address(pUSDe)), user2SubVaultSharesInEach);

    // verify user2 entitled to withdraw their total token amount
    assertEq(pUSDe.maxWithdraw(user2), user2AmountInEachSubVault*2);

    // try and do it, reverts due to insufficient balance
    vm.startPrank(user2);
    vm.expectRevert(); // ERC20InsufficientBalance
    pUSDe.withdraw(user2AmountInEachSubVault*2, user2, user2);

    // try 1 wei more than largest deposit from user 1, fails for same reason
    vm.expectRevert(); // ERC20InsufficientBalance
    pUSDe.withdraw(user1AmountInMainVault+1, user2, user2);

    // can withdraw up to max deposit amount $1000
    pUSDe.withdraw(user1AmountInMainVault, user2, user2);

    // user2 still has $200 left to withdraw
    assertEq(pUSDe.maxWithdraw(user2), 200e18);

    // trying to withdraw it reverts
    vm.expectRevert(); // ERC20InsufficientBalance
    pUSDe.withdraw(200e18, user2, user2);

    // can't withdraw anymore, even trying 1 wei will revert
    vm.expectRevert();
    pUSDe.withdraw(1e18, user2, user2);
}
```

Recommended Mitigation: In `pUSDeVault::_withdraw`, inside the yield-phase `if` condition, there should be a call to `redeemRequiredBaseAssets` if there is insufficient `USDe` balance to fulfill the withdrawal.

Alternatively another potential fix is to not allow supported vaults to be added during the yield phase (apart from `sUSDe` which is added when the yield phase is enabled).

Strata: Fixed in commit 076d23e by no longer allowing adding new supporting vaults during the yield phase.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 `MetaVault::redeemRequiredBaseAssets` should be able to redeem small amounts from each vault to fill requested amount and avoid redeeming more than requested

Description: `MetaVault::redeemRequiredBaseAssets` is supposed to iterate through the supported vaults, redeeming assets until the required amount of base assets is obtained:

```
/// @notice Iterates through supported vaults and redeems assets until the required amount of base
↪  tokens is obtained
```

Its implementation however only retrieves from a supported vault if that one withdrawal can satisfy the desired amount:

```
function redeemRequiredBaseAssets (uint baseTokens) internal {
    for (uint i = 0; i < assetsArr.length; i++) {
        IERC4626 vault = IERC4626(assetsArr[i].asset);
        uint totalBaseTokens = vault.previewRedeem(vault.balanceOf(address(this)));
        // @audit only withdraw if a single withdraw can satisfy desired amount
        if (totalBaseTokens >= baseTokens) {
            vault.withdraw(baseTokens, address(this), address(this));
            break;
        }
    }
}
```

Impact: This has a number of potential problems:

1) if no single withdraw can satisfy the desired amount, then the calling function will revert due to insufficient funds even if the desired amount could be satisfied by multiple smaller withdrawals from different supported vaults
2) a single withdraw may be greater than the desired amount, leaving `USDe` tokens inside the vault contract. This is suboptimal as then they would not be earning yield by being staked in `sUSDe`, and there appears to be no way for the contract owner to trigger the staking once the yield phase has started, since supporting vaults can be added and deposits for them work during the yield phase

Recommended Mitigation: `MetaVault::redeemRequiredBaseAssets` should:

- keep track of the total currently redeemed amount
- calculate the remaining requested amount as the requested amount minus the total currently redeemed amount
- if the current vault is not able to redeem the remaining requested amount, redeem as much as possible and increase the total currently redeemed amount by the amount redeemed
- if the current vault could redeem more than the remaining requested amount, redeem only enough to satisfy the remaining requested amount

The above strategy ensures that:

- small amounts from multiple vaults can be used to fulfill the requested amount
- greater amounts than requested are not withdrawn, so no `USDe` tokens remain inside the vault unable to be staked and not earning yield

Strata: Fixed in commits 4efba0c, 7e6e859.

Cyfrin: Verified.

7.3.2 DoS of meta vault withdrawals during points phase if one vault is paused or attempted redemption exceeds the maximum

Description: `pUSDeVault::_withdraw` assumes any `USDe` shortfall is covered by the multi-vaults; however, `redeemRequiredBaseAssets()` does not guarantee that the required assets are available or actually withdrawn, so the subsequent ERC-20 token transfer could fail and DoS withdrawals if the ERC-4626 withdrawal does not already revert. Usage of `ERC4626Upgradeable::previewRedeem` in `redeemRequiredBaseAssets()` is problematic as this could attempt to withdraw more assets than the vault will allow. Per the ERC-4626 specification, `previewRedeem()`:

- MUST NOT account for redemption limits like those returned from maxRedeem and should always act as though the redemption would be accepted, regardless if the user has enough shares, etc.
- MUST NOT revert due to vault specific user/global limits. MAY revert due to other conditions that would also cause redeem to revert.

So an availability-aware check such as `maxWithdraw()` which considers pause states and any other limits should be used instead to prevent one vault reverting when it may be possible to process the withdrawal by redeeming from another.

Impact: If one of the supported meta vaults is paused or experiences a hack of the underlying `USDe` which results in a decrease in share price during the points phase then this will prevent withdrawals from being processed even if it is possible to do so by redeeming from another.

Proof of Concept: First modify the `MockERC4626` to simulate a vault that pauses deposits/withdrawals and could return fewer assets when querying `maxWithdraw()` when compared with `previewRedeem()`:

```
contract MockERC4626 is ERC4626 {
    bool public depositsEnabled;
    bool public withdrawalsEnabled;
    bool public hacked;

    error DepositsDisabled();
    error WithdrawalsDisabled();

    event DepositsEnabled(bool enabled);
    event WithdrawalsEnabled(bool enabled);

    constructor(IERC20 token) ERC20("MockERC4626", "M4626") ERC4626(token)  {}

    function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal
    ↪   override {
        if (!depositsEnabled) {
            revert DepositsDisabled();
        }

        super._deposit(caller, receiver, assets, shares);
    }

    function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
        internal
        override
    {
        if (!withdrawalsEnabled) {
            revert WithdrawalsDisabled();
        }

        super._withdraw(caller, receiver, owner, assets, shares);
    }

    function maxWithdraw(address owner) public view override returns (uint256) {
        if (!withdrawalsEnabled) {
            revert WithdrawalsDisabled();
        }
        if (hacked) {
            return super.maxWithdraw(owner) / 2; // Reduce max withdraw by half to simulate some limit
        }
        return super.maxWithdraw(owner);
    }

    function totalAssets() public view override returns (uint256) {
        if (hacked) {
            return super.totalAssets() * 3/4; // Reduce total assets by 25% to simulate some loss
        }
        return super.totalAssets();
    }

    function setDepositsEnabled(bool depositsEnabled_) external {
        depositsEnabled = depositsEnabled_;
        emit DepositsEnabled(depositsEnabled_);
    }

    function setWithdrawalsEnabled(bool withdrawalsEnabled_) external {
        withdrawalsEnabled = withdrawalsEnabled_;
        emit WithdrawalsEnabled(withdrawalsEnabled_);
    }

    function hack() external {
        hacked = true;
    }
}
```

The following test can then be run in `pUSDeVault.t.sol`:

```
error WithdrawalsDisabled();
error ERC4626ExceededMaxWithdraw(address owner, uint256 assets, uint256 max);
error ERC20InsufficientBalance(address from, uint256 balance, uint256 amount);

function test_redeemRequiredBaseAssetsDoS() public {
    assert(address(USDe) != address(0));

    account = msg.sender;

    // deposit USDe
    USDe.mint(account, 10 ether);
    deposit(USDe, 10 ether);
    assertBalance(pUSDe, account, 10 ether, "Initial deposit");

    // deposit eUSDe
    USDe.mint(account, 10 ether);
    USDe.approve(address(eUSDe), 10 ether);
    eUSDe.setDepositsEnabled(true);
    eUSDe.deposit(10 ether, account);
    assertBalance(eUSDe, account, 10 ether, "Deposit to eUSDe");
    eUSDe.approve(address(pUSDeDepositor), 10 ether);
    pUSDeDepositor.deposit(eUSDe, 10 ether, account);

    // simulate trying to withdraw from the eUSDe vault when it is paused
    uint256 withdrawAmount = 20 ether;
    eUSDe.setWithdrawalsEnabled(false);
    vm.expectRevert(abi.encodeWithSelector(WithdrawalsDisabled.selector));
    pUSDe.withdraw(address(USDe), withdrawAmount, account, account);
    eUSDe.setWithdrawalsEnabled(true);
    // deposit USDe from another account
    account = address(0x1234);
    vm.startPrank(account);
    USDe.mint(account, 10 ether);
    USDe.approve(address(eUSDe), 10 ether);
    eUSDe.deposit(10 ether, account);
    assertBalance(eUSDe, account, 10 ether, "Deposit to eUSDe");
    eUSDe.approve(address(pUSDeDepositor), 10 ether);
    pUSDeDepositor.deposit(eUSDe, 10 ether, account);
    vm.stopPrank();
    account = msg.sender;
    vm.startPrank(account);

    // deposit eUSDe2
    USDe.mint(account, 5 ether);
    USDe.approve(address(eUSDe2), 5 ether);
    eUSDe2.setDepositsEnabled(true);
    eUSDe2.deposit(5 ether, account);
    assertBalance(eUSDe2, account, 5 ether, "Deposit to eUSDe2");
    eUSDe2.approve(address(pUSDeDepositor), 5 ether);
    pUSDeDepositor.deposit(eUSDe2, 5 ether, account);


    // simulate when previewRedeem() in redeemRequiredBaseAssets() returns more than maxWithdraw()
    ↪  during withdrawal
    // as a result of a hack and imposition of a limit
    eUSDe.hack();
    uint256 maxWithdraw = eUSDe.maxWithdraw(address(pUSDe));
    vm.expectRevert(abi.encodeWithSelector(ERC4626ExceededMaxWithdraw.selector, address(pUSDe),
    ↪  withdrawAmount/2, maxWithdraw));
    pUSDe.withdraw(address(USDe), withdrawAmount, account, account);

    // attempt to withdraw from eUSDe2 vault, but redeemRequiredBaseAssets() skips withdrawal attempt
    // so there are insufficient assets to cover the subsequent transfer even though there is enough in
    ↪  the vaults
    eUSDe2.setWithdrawalsEnabled(true);
    vm.expectRevert(abi.encodeWithSelector(ERC20InsufficientBalance.selector, address(pUSDe),
    ↪  eUSDe2.balanceOf(address(pUSDe)), withdrawAmount));
    pUSDe.withdraw(address(eUSDe2), withdrawAmount, account, account);
}
```

Recommended Mitigation:

```
    function redeemRequiredBaseAssets (uint baseTokens) internal {
        for (uint i = 0; i < assetsArr.length; i++) {
            IERC4626 vault = IERC4626(assetsArr[i].asset);
--          uint totalBaseTokens = vault.previewRedeem(vault.balanceOf(address(this)));
++          uint256 totalBaseTokens = vault.maxWithdraw(address(this));
            if (totalBaseTokens >= baseTokens) {
                vault.withdraw(baseTokens, address(this), address(this));
                break;
            }
        }
    }
```

Strata: Fixed in commit 4efba0c.

Cyfrin: Verified.

7.3.3 Value leakage due to pUSDe redemptions rounding against the protocol/yUSDe depositors

Description: After transitioning to the yield phase, redemptions of both pUSDe and yUSDe are processed by `pUSDeVault::_withdraw` such that they are both paid out in sUSDe. This is achieved by computing the sUSDe balance corresponding to the required USDe amount by calling its `previewWithdraw()` function:

```
    function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)
    ↪  internal override {

            if (PreDepositPhase.YieldPhase == currentPhase) {
                // sUSDeAssets = sUSDeAssets + user_yield_sUSDe
@>              assets += previewYield(caller, shares);

@>              uint sUSDeAssets = sUSDe.previewWithdraw(assets); // @audit - this rounds up because
↪   sUSDe requires the amount of sUSDe burned to receive assets amount of USDe to round up, but below
↪   we are transferring this rounded value out to the receiver which actually rounds against the
↪   protocol/yUSDe depositors!

                _withdraw(
                    address(sUSDe),
                    caller,
                    receiver,
                    owner,
                    assets, // @audit - this should not include the yield, since it is decremented from
                    ↪  depositedBase
                    sUSDeAssets,
                    shares
                );
                return;
            }
        ...
    }
```

The issue with this is that `previewWithdraw()` returns the required sUSDe balance that must be burned to receive the specified USDe amount and so rounds up accordingly; however, here this rounded sUSDe amount is being transferred out of the protocol. This means that the redemption actually rounds in favour of the receiver and against the protocol/yUSDe depositors.

Impact: Value can leak from the system in favour of pUSDe redemptions at the expense of other yUSDe depositors.

Proof of Concept: Note that the following test will revert due to underflow when attempting to determine the fully redeemed amounts unless the mitigation from C-01 is applied:

```
pragma solidity 0.8.28;

import {Test} from "forge-std/Test.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {MockUSDe} from "../contracts/test/MockUSDe.sol";
import {MockStakedUSDe} from "../contracts/test/MockStakedUSDe.sol";
import {MockERC4626} from "../contracts/test/MockERC4626.sol";

import {pUSDeVault} from "../contracts/predeposit/pUSDeVault.sol";
import {yUSDeVault} from "../contracts/predeposit/yUSDeVault.sol";

import {console2} from "forge-std/console2.sol";

contract RoundingTest is Test {
    uint256 constant MIN_SHARES = 0.1 ether;
```

```
MockUSDe public USDe;
MockStakedUSDe public sUSDe;
pUSDeVault public pUSDe;
yUSDeVault public yUSDe;

address account;

address alice = makeAddr("alice");
address bob = makeAddr("bob");

function setUp() public {
    address owner = msg.sender;

    USDe = new MockUSDe();
    sUSDe = new MockStakedUSDe(USDe, owner, owner);

    pUSDe = pUSDeVault(
        address(
            new ERC1967Proxy(
                address(new pUSDeVault()),
                abi.encodeWithSelector(pUSDeVault.initialize.selector, owner, USDe, sUSDe)
            )
        )
    );

    yUSDe = yUSDeVault(
        address(
            new ERC1967Proxy(
                address(new yUSDeVault()),
                abi.encodeWithSelector(yUSDeVault.initialize.selector, owner, USDe, sUSDe, pUSDe)
            )
        )
    );

    vm.startPrank(owner);
    pUSDe.setDepositsEnabled(true);
    pUSDe.setWithdrawalsEnabled(true);
    pUSDe.updateYUSDeVault(address(yUSDe));

    // deposit USDe and burn minimum shares to avoid reverting on redemption
    uint256 initialUSDeAmount = pUSDe.previewMint(MIN_SHARES);
    USDe.mint(owner, initialUSDeAmount);
    USDe.approve(address(pUSDe), initialUSDeAmount);
    pUSDe.mint(MIN_SHARES, address(0xdead));
    vm.stopPrank();

    if (pUSDe.balanceOf(address(0xdead)) != MIN_SHARES) {
        revert("address(0xdead) should have MIN_SHARES shares of pUSDe");
    }
}

function test_rounding() public {
    uint256 userDeposit = 100 ether;

    // fund users
    USDe.mint(alice, userDeposit);
    USDe.mint(bob, userDeposit);

    // alice deposits into pUSDe
    vm.startPrank(alice);
    USDe.approve(address(pUSDe), userDeposit);
```

```
uint256 aliceShares_pUSDe = pUSDe.deposit(userDeposit, alice);
vm.stopPrank();

// bob deposits into pUSDe
vm.startPrank(bob);
USDe.approve(address(pUSDe), userDeposit);
uint256 bobShares_pUSDe = pUSDe.deposit(userDeposit, bob);
vm.stopPrank();

// setup assertions
assertEq(pUSDe.balanceOf(alice), aliceShares_pUSDe, "Alice should have shares equal to her
↪  deposit");
assertEq(pUSDe.balanceOf(bob), bobShares_pUSDe, "Bob should have shares equal to his deposit");

{
    // phase change
    account = msg.sender;
    uint256 initialAdminTransferAmount = 1e6;
    vm.startPrank(account);
    USDe.mint(account, initialAdminTransferAmount);
    USDe.approve(address(pUSDe), initialAdminTransferAmount);
    pUSDe.deposit(initialAdminTransferAmount, address(yUSDe));
    pUSDe.startYieldPhase();
    yUSDe.setDepositsEnabled(true);
    yUSDe.setWithdrawalsEnabled(true);
    vm.stopPrank();
}

// bob deposits into yUSDe
vm.startPrank(bob);
pUSDe.approve(address(yUSDe), bobShares_pUSDe);
uint256 bobShares_yUSDe = yUSDe.deposit(bobShares_pUSDe, bob);
vm.stopPrank();

// simulate sUSDe yield transfer
uint256 sUSDeYieldAmount = 1_000 ether;
USDe.mint(address(sUSDe), sUSDeYieldAmount);

// alice redeems from pUSDe
uint256 aliceBalanceBefore_sUSDe = sUSDe.balanceOf(alice);
vm.prank(alice);
uint256 aliceRedeemed_USDe_reported = pUSDe.redeem(aliceShares_pUSDe, alice, alice);
uint256 aliceRedeemed_sUSDe = sUSDe.balanceOf(alice) - aliceBalanceBefore_sUSDe;
uint256 aliceRedeemed_USDe_actual = sUSDe.previewRedeem(aliceRedeemed_sUSDe);

// bob redeems from yUSDe
uint256 bobBalanceBefore_sUSDe = sUSDe.balanceOf(bob);
vm.prank(bob);
uint256 bobRedeemed_pUSDe_reported = yUSDe.redeem(bobShares_yUSDe, bob, bob);
uint256 bobRedeemed_sUSDe = sUSDe.balanceOf(bob) - bobBalanceBefore_sUSDe;
uint256 bobRedeemed_USDe = sUSDe.previewRedeem(bobRedeemed_sUSDe);

console2.log("Alice redeemed sUSDe: %s", aliceRedeemed_sUSDe);
console2.log("Alice redeemed USDe (reported): %s", aliceRedeemed_USDe_reported);
console2.log("Alice redeemed USDe (actual): %s", aliceRedeemed_USDe_actual);

console2.log("Bob redeemed pUSDe (reported): %s", bobRedeemed_pUSDe_reported);
console2.log("Bob redeemed pUSDe (actual): %s", bobShares_pUSDe);
console2.log("Bob redeemed sUSDe: %s", bobRedeemed_sUSDe);
console2.log("Bob redeemed USDe: %s", bobRedeemed_USDe);

// post-redemption assertions
```

```
        assertEq(
            aliceRedeemed_USDe_reported,
            aliceRedeemed_USDe_actual,
            "Alice's reported and actual USDe redemption amounts should match"
        );

        assertGe(
            bobRedeemed_pUSDe_reported,
            bobShares_pUSDe,
            "Bob should redeem at least the same amount of pUSDe as his original deposit"
        );

        assertGe(
            bobRedeemed_USDe, userDeposit, "Bob should redeem at least the same amount of USDe as his
            ↪  initial deposit"
        );

        assertLe(
            aliceRedeemed_USDe_actual,
            userDeposit,
            "Alice should redeem no more than the same amount of USDe as her initial deposit"
        );
    }
}
```

The following Echidna optimization test can also be run to maximise this discrepancy:

```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {BaseSetup} from "@chimera/BaseSetup.sol";
import {CryticAsserts} from "@chimera/CryticAsserts.sol";
import {vm} from "@chimera/Hevm.sol";

import {pUSDeVault} from "contracts/predeposit/pUSDeVault.sol";
import {yUSDeVault} from "contracts/predeposit/yUSDeVault.sol";
import {MockUSDe} from "contracts/test/MockUSDe.sol";
import {MockStakedUSDe} from "contracts/test/MockStakedUSDe.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

// echidna . --contract CryticRoundingTester --config echidna_rounding.yaml --format text --workers 16
↪  --test-limit 1000000
contract CryticRoundingTester is BaseSetup, CryticAsserts {
    uint256 constant MIN_SHARES = 0.1 ether;

    MockUSDe USDe;
    MockStakedUSDe sUSDe;
    pUSDeVault pUSDe;
    yUSDeVault yUSDe;

    address owner;
    address alice = address(uint160(uint256(keccak256(abi.encodePacked("alice")))));
    address bob = address(uint160(uint256(keccak256(abi.encodePacked("bob")))));
    uint256 severity;

    constructor() payable {
        setup();
    }

    function setup() internal virtual override {
        owner = msg.sender;
        USDe = new MockUSDe();
        sUSDe = new MockStakedUSDe(USDe, owner, owner);

        pUSDe = pUSDeVault(
            address(
                new ERC1967Proxy(
                    address(new pUSDeVault()),
                    abi.encodeWithSelector(pUSDeVault.initialize.selector, owner, USDe, sUSDe)
                )
            )
        );

        yUSDe = yUSDeVault(
            address(
                new ERC1967Proxy(
                    address(new yUSDeVault()),
                    abi.encodeWithSelector(yUSDeVault.initialize.selector, owner, USDe, sUSDe, pUSDe)
                )
            )
        );

        vm.startPrank(owner);
        pUSDe.setDepositsEnabled(true);
        pUSDe.setWithdrawalsEnabled(true);
        pUSDe.updateYUSDeVault(address(yUSDe));

        // deposit USDe and burn minimum shares to avoid reverting on redemption
        uint256 initialUSDeAmount = pUSDe.previewMint(MIN_SHARES);
        USDe.mint(owner, initialUSDeAmount);
        USDe.approve(address(pUSDe), initialUSDeAmount);
        pUSDe.mint(MIN_SHARES, address(0xdead));
        vm.stopPrank();

        if (pUSDe.balanceOf(address(0xdead)) != MIN_SHARES) {
            revert("address(0xdead) should have MIN_SHARES shares of pUSDe");
        }
    }

    function target(uint256 aliceDeposit, uint256 bobDeposit, uint256 sUSDeYieldAmount) public {
        aliceDeposit = between(aliceDeposit, 1, 100_000 ether);
        bobDeposit = between(bobDeposit, 1, 100_000 ether);
        sUSDeYieldAmount = between(sUSDeYieldAmount, 1, 500_000 ether);
        precondition(aliceDeposit <= 100_000 ether);
        precondition(bobDeposit <= 100_000 ether);
        precondition(sUSDeYieldAmount <= 500_000 ether);

        // fund users
        USDe.mint(alice, aliceDeposit);
        USDe.mint(bob, bobDeposit);

        // alice deposits into pUSDe
        vm.startPrank(alice);
        USDe.approve(address(pUSDe), aliceDeposit);
        uint256 aliceShares_pUSDe = pUSDe.deposit(aliceDeposit, alice);
        vm.stopPrank();

        // bob deposits into pUSDe
        vm.startPrank(bob);
        USDe.approve(address(pUSDe), bobDeposit);
        uint256 bobShares_pUSDe = pUSDe.deposit(bobDeposit, bob);
        vm.stopPrank();
        // setup assertions
        eq(pUSDe.balanceOf(alice), aliceShares_pUSDe, "Alice should have shares equal to her deposit");
        eq(pUSDe.balanceOf(bob), bobShares_pUSDe, "Bob should have shares equal to his deposit");

        {
            // phase change
            uint256 initialAdminTransferAmount = 1e6;
            vm.startPrank(owner);
            USDe.mint(owner, initialAdminTransferAmount);
            USDe.approve(address(pUSDe), initialAdminTransferAmount);
            pUSDe.deposit(initialAdminTransferAmount, address(yUSDe));
            pUSDe.startYieldPhase();
            yUSDe.setDepositsEnabled(true);
            yUSDe.setWithdrawalsEnabled(true);
            vm.stopPrank();
        }

        // bob deposits into yUSDe
        vm.startPrank(bob);
        pUSDe.approve(address(yUSDe), bobShares_pUSDe);
        uint256 bobShares_yUSDe = yUSDe.deposit(bobShares_pUSDe, bob);
        vm.stopPrank();

        // simulate sUSDe yield transfer
        USDe.mint(address(sUSDe), sUSDeYieldAmount);

        // alice redeems from pUSDe
        uint256 aliceBalanceBefore_sUSDe = sUSDe.balanceOf(alice);
        vm.prank(alice);
        uint256 aliceRedeemed_USDe_reported = pUSDe.redeem(aliceShares_pUSDe, alice, alice);
        uint256 aliceRedeemed_sUSDe = sUSDe.balanceOf(alice) - aliceBalanceBefore_sUSDe;
        uint256 aliceRedeemed_USDe_actual = sUSDe.previewRedeem(aliceRedeemed_sUSDe);

        // bob redeems from yUSDe
        uint256 bobBalanceBefore_sUSDe = sUSDe.balanceOf(bob);
        vm.prank(bob);
        uint256 bobRedeemed_pUSDe_reported = yUSDe.redeem(bobShares_yUSDe, bob, bob);
        uint256 bobRedeemed_sUSDe = sUSDe.balanceOf(bob) - bobBalanceBefore_sUSDe;
        uint256 bobRedeemed_USDe = sUSDe.previewRedeem(bobRedeemed_sUSDe);

        // optimize
        if (aliceRedeemed_USDe_actual > aliceDeposit) {
            uint256 diff = aliceRedeemed_USDe_actual - aliceDeposit;
            if (diff > severity) {
                severity = diff;
            }
        }
    }

    function echidna_opt_severity() public view returns (uint256) {
        return severity;
    }
}
```

Config:

```
testMode: "optimization"
prefix: "echidna_"
coverage: true
corpusDir: "echidna_rounding"
balanceAddr: 0x1043561a8829300000
balanceContract: 0x1043561a8829300000
```

```
filterFunctions: []
cryticArgs: ["--foundry-compile-all"]
deployer: "0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496"
contractAddr: "0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496"
shrinkLimit: 100000
```

Output:

```
echidna_opt_severity: max value: 444330
```

Recommended Mitigation: Rather than calling `previewWithdraw()` which rounds up, call `convertToShares()` which rounds down:

```
function previewWithdraw(uint256 assets) public view virtual override returns (uint256) {
    return _convertToShares(assets, Math.Rounding.Up);
}

function convertToShares(uint256 assets) public view virtual override returns (uint256) {
    return _convertToShares(assets, Math.Rounding.Down);
}
```

Strata: Fixed in commit 59fcf23.

Cyfrin: Verified. The sUSDe to transfer out to the receiver is now calculated using `convertToShares()` which rounds down.

7.4 Low Risk

7.4.1 Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision

Description: The protocol has upgradeable contracts which other contracts inherit from. These contracts should either use:

- ERC7201 namespaced storage layouts - example
- storage gaps (though this is an older and no longer preferred method)

The ideal mitigation is that all upgradeable contracts use ERC7201 namespaced storage layouts.

Without using one of the above two techniques storage collision can occur during upgrades.

Strata: Fixed in commit 98068bd.

Cyfrin: Verified.

7.4.2 In `pUSDeDepositor::deposit_viaSwap`, using `block.timestamp` in swap deadline is not very effective

Description: Using `block.timestamp` in a swap deadline is not very effective since `block.timestamp` will be the block which the transaction gets put in, so the swap will never be able to expire in this way.

Instead the current `block.timestamp` should be retrieved off-chain and passed as input to the swap transaction.

Strata: Fixed in commit 2c43c07.

Cyfrin: Verified. Callers can now override the default swap deadline.

7.4.3 Hard-coded slippage in `pUSDeDepositor::deposit_viaSwap` can lead to denial of service

Description: Hard-coded slippage in `pUSDeDepositor::deposit_viaSwap` can lead to denial of service and in dramatic cases even lock user funds.

Recommended Mitigation: Slippage parameters should be calculated off-chain and supplied as input to swaps.

Strata: Fixed in commit 2c43c07.

Cyfrin: Verified. Callers can now override the default slippage.

7.4.4 Use `SafeERC20::forceApprove` instead of standard `IERC20::approve`

Description: Use `SafeERC20::forceApprove` when dealing with a range of potential tokens instead of standard `IERC20::approve`:

```
predeposit/yUSDeDepositor.sol
58:        pUSDe.approve(address(yUSDe), amount);

predeposit/pUSDeVault.sol
178:        USDe.approve(address(sUSDe), USDeAssets);

predeposit/pUSDeDepositor.sol
86:            asset.approve(address(vault), amount);
98:        sUSDe.approve(address(pUSDe), amount);
110:        USDe.approve(address(pUSDe), amount);
122:        token.approve(swapInfo.router, amount);
```

Strata: Fixed in commit f258bdc.

Cyfrin: Verified.

7.4.5 `MetaVault::redeem` erroneously calls `ERC4626Upgradeable::withdraw` when attempting to redeem `USDe` from `pUSDeVault`

Description: Unlike `MetaVault::deposit`, `MetaVault::mint`, and `MetaVault::withdraw` which all invoke the corresponding `IERC4626` function, `MetaVault::redeem` erroneously calls `ERC4626Upgradeable::withdraw` when attempting to redeem `USDe` from `pUSDeVault`:

```
function redeem(address token, uint256 shares, address receiver, address owner) public virtual returns
↪  (uint256) {
    if (token == asset()) {
        return withdraw(shares, receiver, owner);
    }
    ...
}
```

Impact: The behavior of `MetaVault::redeem` differs from that which is expected depending on whether `token` is specified as `USDe` or one of the other supported vault tokens.

Recommended Mitigation:

```
    function redeem(address token, uint256 shares, address receiver, address owner) public virtual
    ↪  returns (uint256) {
        if (token == asset()) {
--          return withdraw(shares, receiver, owner);
++          return redeem(shares, receiver, owner);
        }
        ...
    }
```

Strata: Fixed in commit 7665e7f.

Cyfrin: Verified.

7.4.6 Duplicate vaults can be pushed to `assetsArr`

Description: While `MetaVault::addVault` is protected by the `onlyOwner` modifier, there is no restriction on the number of times this function can be called with a given `vaultAddress` as argument:

```
    function addVault(address vaultAddress) external onlyOwner {
        addVaultInner(vaultAddress);
    }

    function addVaultInner (address vaultAddress) internal {
        TAsset memory vault = TAsset(vaultAddress, EAssetType.ERC4626);
        assetsMap[vaultAddress] = vault;
@>      assetsArr.push(vault);

        emit OnVaultAdded(vaultAddress);
    }
```

In such a scenario, the vault will become duplicated within the `assetsArr` array. When called in `pUSDeVault::startYieldPhase`, the core redemption logic of `MetaVault::redeemMetaVaults` continues to function as expected. During the second iteration for the given vault address, the contract balance will simply be zero, so the redemption will be skipped, the `assetsMap` entry will again be re-written to default values, and the duplicate element will be removed from the array:

```
    function removeVaultAndRedeemInner (address vaultAddress) internal {
        // Redeem
        uint balance = IERC20(vaultAddress).balanceOf(address(this));
@>      if (balance > 0) {
@>          IERC4626(vaultAddress).redeem(balance, address(this), address(this));
```

```
        }

        // Clean
        TAsset memory emptyAsset;
@>      assetsMap[vaultAddress] = emptyAsset;
        uint length = assetsArr.length;
        for (uint i = 0; i < length; i++) {
            if (assetsArr[i].asset == vaultAddress) {
                assetsArr[i] = assetsArr[length - 1];
@>              assetsArr.pop();
                break;
            }
        }
    }

    /// @dev Internal method to redeem all assets from supported vaults
    /// @notice Iterates through all supported vaults and redeems their assets for the base token
    function redeemMetaVaults () internal {
        while (assetsArr.length > 0) {
@>          removeVaultAndRedeemInner(assetsArr[0].asset);
        }
    }
```

However, if the given vault is removed from the list of supported vaults, `MetaVault::removeVault` will not allow the duplicate entry to be removed since the `requireSupportedVault()` invocation would fail on any subsequent attempt given that the mapping state is already overwritten to `address(0)` in the `removeVaultAndRedeemInner()` invocation:

```
    function requireSupportedVault(address token) internal view {
@>      address vaultAddress = assetsMap[token].asset;
        if (vaultAddress == address(0)) {
            revert UnsupportedAsset(token);
        }
    }

    function removeVault(address vaultAddress) external onlyOwner {
@>      requireSupportedVault(vaultAddress);
        removeVaultAndRedeemInner(vaultAddress);

        emit OnVaultRemoved(vaultAddress);
    }
```

The consequence of this depends on the intentions of the owner:

- If they intend to keep the vault supported, all `MetaVault` functionality relying on the specified asset being a supported vault will revert if it has been attempted by the owner to remove a duplicated vault.
- If they intend to completely remove the vault, this will not be possible; however, it will also not be possible to make any subsequent deposits, so impact is limited to redeeming during the transition to the yield phase rather than instantaneously.

Impact: Vault assets could be redeemed later than intended and users could be temporarily prevented from withdrawing their funds.

Proof of Concept: The following test should be included in `pUSDeVault.t.sol`:

```
function test_duplicateVaults() public {
    pUSDe.addVault(address(eUSDe));
    pUSDe.removeVault(address(eUSDe));
    assertFalse(pUSDe.isAssetSupported(address(eUSDe)));
    vm.expectRevert();
    pUSDe.removeVault(address(eUSDe));
}
```

Recommended Mitigation: Revert if the given vault has already been added.

Strata: Fixed in commit 787d1c7.

Cyfrin: Verified.

7.4.7 `MetaVault::addVault` should enforce identical underlying base asset

Description: When supporting additional vaults, `MetaVault::addVault` should enforce that the new vault being supported has an identical underlying base asset as itself. Otherwise:

- `redeemRequiredBaseAssets` won't work as expected since the newly supported vault doesn't have the same base asset
- `MetaVault::depositedBase` will become corrupt, especially if the underlying asset tokens use different decimal precision

Proof of Concept:

```
function test_vaultSupportedWithDifferentUnderlyingAsset() external {
    // create ERC4626 vault with different underlying ERC20 asset
    MockUSDe differentERC20 = new MockUSDe();
    MockERC4626 newSupportedVault = new MockERC4626(differentERC20);

    // verify pUSDe doesn't have same underlying asset as new vault
    assertNotEq(pUSDe.asset(), newSupportedVault.asset());

    // but still allows it to be added
    pUSDe.addVault(address(newSupportedVault));

    // this breaks `MetaVault::redeemRequiredBaseAssets` since
    // the newly supported vault doesn't have the same base asset
}
```

Recommended Mitigation: Change `MetaVault::addVaultInner`:

```
    function addVaultInner (address vaultAddress) internal {
+       IERC4626 newVault = IERC4626(vaultAddress);
+       require(newVault.asset() == asset(), "Vault asset mismatch");
```

Strata: Fixed in commits 9e64f09, 706c2df.

Cyfrin: Verified.

7.4.8 `pUSDeVault::startYieldPhase` should not remove supported vaults from being supported or should prevent new supported vaults once in the yield phase

Description: The intention of `pUSDeVault::startYieldPhase` is to convert assets from existing supported vaults into `USDe` in order to then stake the vault's total `USDe` into the `sUSDe` vault.

However because this ends up calling `MetaVault::removeVaultAndRedeemInner`, all the supported vaults are also removed after their assets are converted.

But new vaults can continue to be added during the yield phase, so it makes no sense to remove all supported vaults at this time.

Impact: The contract owner will need to re-add all the previously enabled supported vaults causing all user deposits to revert until this is done.

Proof Of Concept:

```
function test_supportedVaultsRemovedWhenYieldPhaseEnabled() external {
    // supported vault prior to yield phase
    assertTrue(pUSDe.isAssetSupported(address(eUSDe)));

    // user1 deposits $1000 USDe into the main vault
    uint256 user1AmountInMainVault = 1000e18;
    USDe.mint(user1, user1AmountInMainVault);

    vm.startPrank(user1);
    USDe.approve(address(pUSDe), user1AmountInMainVault);
    uint256 user1MainVaultShares = pUSDe.deposit(user1AmountInMainVault, user1);
    vm.stopPrank();

    // admin triggers yield phase on main vault
    pUSDe.startYieldPhase();

    // supported vault was removed when initiating yield phase
    assertFalse(pUSDe.isAssetSupported(address(eUSDe)));

    // but can be added back in?
    pUSDe.addVault(address(eUSDe));
    assertTrue(pUSDe.isAssetSupported(address(eUSDe)));

    // what was the point of removing it if it can be re-added
    // and used again during the yield phase?
}
```

Recommended Mitigation: Don't remove all supported vaults when calling `pUSDeVault::startYieldPhase`; just convert their assets to `USDe` but continue to allow the vaults themselves to be supported and accept future deposits.

Alternatively don't allow supported vaults to be added during the yield phase (apart from sUSDe which is added when the yield phase is enabled). In this case removing them when enabled the yield phase is fine, but add code to disallow adding them once the yield phase is enabled.

Strata: Fixed in commit 076d23e by no longer allowing adding new supporting vaults during the yield phase.

Cyfrin: Verified.

7.4.9 No way to compound deposited supported vault assets into `sUSDe` stake during yield phase

Description: Once the yield phase has been enabled, `pUSDeVault` still allows new supported vaults to be added and deposits via supported vaults.

However for supported vaults which are not `sUSDe`, there is no way to withdraw their base token `USDe` and compound into the `sUSDe` vault stake used by the `pUSDeVault` vault.

Recommended Mitigation: Either don't allow supported vaults to be added apart from `sUSDe` once yield phase has been enabled, or implement a function to withdraw their base token and compound it into the main stake.

Strata: Fixed in commit 076d23e by no longer allowing adding new supporting vaults during the yield phase.

Cyfrin: Verified.

7.4.10 `pUSDeVault::maxWithdraw` doesn't account for withdrawal pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`

Description: EIP-4626 states on `maxWithdraw`:

> MUST factor in both global and user-specific limits, like if withdrawals are entirely disabled (even temporarily) it MUST return 0.

pUSDeVault::maxWithdraw doesn't account for withdrawal pausing, in violation of EIP-4626 which can break protocols integrating with pUSDeVault.

Proof of Concept:

```
function test_maxWithdraw_WhenWithdrawalsPaused() external {
    // user1 deposits $1000 USDe into the main vault
    uint256 user1AmountInMainVault = 1000e18;
    USDe.mint(user1, user1AmountInMainVault);

    vm.startPrank(user1);
    USDe.approve(address(pUSDe), user1AmountInMainVault);
    uint256 user1MainVaultShares = pUSDe.deposit(user1AmountInMainVault, user1);
    vm.stopPrank();

    // admin pauses withdrawals
    pUSDe.setWithdrawalsEnabled(false);

    // reverts as maxWithdraw returns user1AmountInMainVault even though
    // attempting to withdraw would revert
    assertEq(pUSDe.maxWithdraw(user1), 0);

    // https://eips.ethereum.org/EIPS/eip-4626 maxWithdraw says:
    // MUST factor in both global and user-specific limits,
    // like if withdrawals are entirely disabled (even temporarily) it MUST return 0
}
```

Recommended Mitigation: When withdrawals are paused, maxWithdraw should return 0. The override of maxWithdraw should likely be done in PreDepositVault because there is where the pausing is implemented.

Strata: Fixed in commit 8021069.

Cyfrin: Verified.

7.4.11 pUSDeVault::maxDeposit doesn't account for deposit pausing, in violation of EIP-4626 which can break protocols integrating with pUSDeVault

Description: EIP-4626 states on maxDeposit:

> MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.

pUSDeVault::maxDeposit doesn't account for deposit pausing, in violation of EIP-4626 which can break protocols integrating with pUSDeVault.

Proof of Concept:

```
function test_maxDeposit_WhenDepositsPaused() external {
    // admin pauses deposits
    pUSDe.setDepositsEnabled(false);

    // reverts as maxDeposit returns uint256.max even though
    // attempting to deposit would revert
    assertEq(pUSDe.maxDeposit(user1), 0);

    // https://eips.ethereum.org/EIPS/eip-4626 maxDeposit says:
    // MUST factor in both global and user-specific limits,
    // like if deposits are entirely disabled (even temporarily) it MUST return 0.
}
```

Recommended Mitigation: When deposits are paused, maxDeposit should return 0. The override of maxDeposit should likely be done in PreDepositVault because there is where the pausing is implemented.

Strata: Fixed in commit 8021069.

Cyfrin: Verified.

7.4.12 `pUSDeVault::maxMint` doesn't account for mint pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`

Description: EIP-4626 states on `maxMint`:

> MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0.

`pUSDeVault::maxMint` doesn't account for mint pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`. Since `MetaVault::mint` uses `_deposit`, mints will be paused when deposits are paused.

Proof of Concept:

```
function test_maxMint_WhenDepositsPaused() external {
    // admin pauses deposits
    pUSDe.setDepositsEnabled(false);

    // should revert here as maxMint should return 0
    // since deposits are paused and `MetaVault::mint` uses `_deposit`
    assertEq(pUSDe.maxMint(user1), type(uint256).max);

    // attempt to mint to show the error
    uint256 user1AmountInMainVault = 1000e18;
    USDe.mint(user1, user1AmountInMainVault);

    vm.startPrank(user1);
    USDe.approve(address(pUSDe), user1AmountInMainVault);
    // reverts with DepositsDisabled since `MetaVault::mint` uses `_deposit`
    uint256 user1MainVaultShares = pUSDe.mint(user1AmountInMainVault, user1);
    vm.stopPrank();

    // https://eips.ethereum.org/EIPS/eip-4626 maxMint says:
    // MUST factor in both global and user-specific limits,
    // like if mints are entirely disabled (even temporarily) it MUST return 0.
}
```

Recommended Mitigation: When deposits are paused, `maxMint` should return 0. The override of `maxMint` should likely be done in `PreDepositVault` because there is where the pausing is implemented.

Strata: Fixed in commit 8021069.

Cyfrin: Verified.

7.4.13 `pUSDeVault::maxRedeem` doesn't account for redemption pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`

Description: EIP-4626 states on `maxRedeem`:

> MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

`pUSDeVault::maxRedeem` doesn't account for redemption pausing, in violation of EIP-4626 which can break protocols integrating with `pUSDeVault`. `MetaVault::redeem` uses `_withdraw` so redemptions will be paused when withdrawals are paused.

Proof of Concept:

```
function test_maxRedeem_WhenWithdrawalsPaused() external {
    // user1 deposits $1000 USDe into the main vault
    uint256 user1AmountInMainVault = 1000e18;
    USDe.mint(user1, user1AmountInMainVault);

    vm.startPrank(user1);
    USDe.approve(address(pUSDe), user1AmountInMainVault);
    uint256 user1MainVaultShares = pUSDe.deposit(user1AmountInMainVault, user1);
    vm.stopPrank();

    // admin pauses withdrawals
    pUSDe.setWithdrawalsEnabled(false);

    // doesn't revert but it should since `MetaVault::redeem` uses `_withdraw`
    // and withdraws are paused, so `maxRedeem` should return 0
    assertEq(pUSDe.maxRedeem(user1), user1AmountInMainVault);

    // reverts with WithdrawalsDisabled
    vm.prank(user1);
    pUSDe.redeem(user1MainVaultShares, user1, user1);

    // https://eips.ethereum.org/EIPS/eip-4626 maxRedeem says:
    // MUST factor in both global and user-specific limits,
    // like if redemption are entirely disabled (even temporarily) it MUST return 0
}
```

Recommended Mitigation: When withdrawals are paused, `maxRedeem` should return 0. The override of `maxRedeem` should likely be done in `PreDepositVault` because there is where the pausing is implemented.

Strata: Fixed in commit 8021069.

Cyfrin: Verified.

7.4.14 `yUSDeVault` inherits from `PreDepositVault` but doesn't call `onAfterDepositChecks` or `onAfterWithdrawalChecks`

Description: `pUSDeVault` and `yUSDeVault` both inherit from `PreDepositVault`.

`pUSDeVault` uses `PreDepositVault::onAfterDepositChecks` and `onAfterWithdrawalChecks` inside its overriden `_deposit` and `_withdraw` functions.

However `yUSDeVault` doesn't do this; instead it attempts to re-implement the same code as these functions inside its `_deposit` and `_withdraw`, but omits this code from `onAfterWithdrawalChecks`:

```
if (totalSupply() < MIN_SHARES) {
    revert MinSharesViolation();
}
```

Impact: The `MIN_SHARES` check won't be enforced in `yUSDeVault`.

Recommended Mitigation: Use `PreDepositVault::onAfterDepositChecks` and `onAfterWithdrawalChecks` inside `yUSDeVault::_deposit` and `_withdraw`.

Alternatively if the omission of the `MIN_SHARES` check is intentional, then add a boolean parameter to `onAfterWithdrawalChecks` whether to perform the check or not so that `yUSDeVault` can use the two functions it inherits to reduce code duplication.

Strata: Fixed in commits 3f02ce5, 0812d57.

Cyfrin: Verified.

7.4.15 Inability to remove and redeem from vaults with withdrawal issues could result in a bank-run

Description: When deposits are made to the `pUSDeVault`, `depositedBase` is incremented based on the previewed quote amount of USDe underlying the external ERC-4626 vaults; however, these instantaneous preview quotes are not necessarily accurate when compared to the maximum amount that is actually withdrawable. For example, `MetaVault::deposit` implements calculation of the base USDe assets as:

```
uint baseAssets = IERC4626(token).previewRedeem(tokenAssets);
```

But if the vault has overridden the max withdraw/redeem functions with custom logic that apply some limits then this previewed value could be larger than the actual maximum withdrawable USDe amount. This is possible because the ERC-4626 specification states that preview functions must not account for withdrawal/redemption limits like those returned from maxWithdraw/maxRedeem and should always act as though the redemption would be accepted.

Therefore, given that there is not actually a withdrawal that is executed during the deposit, the `depositedBase` state is incremented assuming the underlying USDe if fully redeemable, but it is not until removing and redeeming the vault that a revert could arise if the third-party vault malfunctions or restricts withdrawals. Currently, the only way to pause new deposits for a given vault is by removing the asset from the supported list; however, doing so also triggers a withdrawal of USDe which can fail for the reasons stated above, preventing the asset from being removed.

While none of the externally-supported vault tokens intend to function with a decrease in share price, it is of course not possible except in very simplistic implementations to rule out the possibility of a smart contract hack in which the underlying USDe is stolen from one of the supported vaults. Combined with the issue above, given that users are free to withdraw into a any supported vault token regardless of those that they supplied, full withdraw by other users into unaffected vault tokens (or even if the required USDe is pulled from these vaults by `MetaVault::redeemRequiredBaseAssets` to process their withdrawals), this could result in a subset of users being left with the bad debt rather than it being amortized.

It is understood that the protocol team has strict criteria for supporting new third-party vaults, including the need for instant withdrawals, no limits, no cooldowns, and not pausable, though exceptions may be made for partners that maintain robust communication channels regarding development plans and updates.

Impact: The inability to remove and redeem from vaults with withdrawal issues could result in a bank-run that leaves a subset of users with un-redeemable tokens.

Recommended Mitigation: Implement some mechanism to disable new deposits to a vault without having to remove it and (attempt to) fully-redeem the underlying tokens. To amortize any losses a potential faulty vault, it may be necessary to track the individual vault contributions to `depositedBase` and so that they can be negated from redemption calculations.

Strata: Fixed in commit ae71893.

Cyfrin: Verified.

7.4.16 `yUSDeVault` edge cases should be explicitly handled to prevent view functions from reverting

Description: Per the ERC-4626 specification, the preview functions "MUST NOT revert due to vault specific user/global limits. MAY revert due to other conditions that would also cause mint/deposit/redeem/withdraw to revert".

```
    function totalAccruedUSDe() public view returns (uint256) {
@>      uint pUSDeAssets = super.totalAssets();  // @audit - should return early if pUSDeAssets is zero
↪   to avoid reverting in the call below

@>      uint USDeAssets = _convertAssetsToUSDe(pUSDeAssets, true);
        return USDeAssets;
    }

    function _convertAssetsToUSDe (uint pUSDeAssets, bool withYield) internal view returns (uint256) {
@>      uint sUSDeAssets = pUSDeVault.previewRedeem(withYield ? address(this) : address(0),
↪   pUSDeAssets); // @audit - this can revert if passing yUSDe as the caller when it has no pUSDe
↪   balance
        uint USDeAssets = sUSDe.previewRedeem(sUSDeAssets);
        return USDeAssets;
    }

    function previewDeposit(uint256 pUSDeAssets) public view override returns (uint256) {
        uint underlyingUSDe = _convertAssetsToUSDe(pUSDeAssets, false);

@>      uint yUSDeShares = _valueMulDiv(underlyingUSDe, totalAssets(), totalAccruedUSDe(),
↪   Math.Rounding.Floor); // @audit - should explicitly handle the case where totalAccruedUSDe()
↪   returns zero rather than relying on _valueMulDiv() behaviour
        return yUSDeShares;
    }

    function previewMint(uint256 yUSDeShares) public view override returns (uint256) {
@>      uint underlyingUSDe = _valueMulDiv(yUSDeShares, totalAccruedUSDe(), totalAssets(),
↪   Math.Rounding.Ceil); // @audit - should explicitly handle the case where totalAccruedUSDe() and/or
↪   totalAssets() returns zero rather than relying on _valueMulDiv() behaviour
        uint pUSDeAssets = pUSDeVault.previewDeposit(underlyingUSDe);
        return pUSDeAssets;
    }

    function _valueMulDiv(uint256 value, uint256 mulValue, uint256 divValue, Math.Rounding rounding)
    ↪   internal view virtual returns (uint256) {
        return value.mulDiv(mulValue + 1, divValue + 1, rounding);
    }
```

As noted using `// @audit` tags in the code snippets above, `yUSDeVault::previewMint` and `yUSDeVault::previewDeposit` can revert for multiple reasons, including:

- when the pUSDe balance of the yUSDe vault is zero.
- when `pUSDeVault::previewRedeem` reverts due to division by zero in `pUSDeVault::previewYield`, invoked from `_convertAssetsToUSDe()` within `totalAccruedUSDe()`.

```
     function previewYield(address caller, uint256 shares) public view virtual returns (uint256) {
        if (PreDepositPhase.YieldPhase == currentPhase && caller == address(yUSDe)) {

            uint total_sUSDe = sUSDe.balanceOf(address(this));
            uint total_USDe = sUSDe.previewRedeem(total_sUSDe);

            uint total_yield_USDe = total_USDe - Math.min(total_USDe, depositedBase);

@>          uint y_pUSDeShares = balanceOf(caller); // @audit - should return early if this is zero to
↪   avoid reverting below
@>          uint caller_yield_USDe = total_yield_USDe.mulDiv(shares, y_pUSDeShares,
↪   Math.Rounding.Floor);

            return caller_yield_USDe;
        }
        return 0;
    }

    function previewRedeem(address caller, uint256 shares) public view virtual returns (uint256) {
        return previewRedeem(shares) + previewYield(caller, shares);
    }
```

While a subset of these reverts could be considered "due to other conditions that would also cause deposit to revert", such as due to overflow, it would be better to explicitly handle these other edge cases. Additionally, even when called in isolation `yUSDeVault::totalAccruedUSDe` will revert if the pUSDe balance of the yUSDeVault is zero. Instead, this should simply return zero.

Strata: Fixed in commit 0f366e1.

Cyfrin: Verified. The zero assets/shares edge cases are now explicitly handled in `yUSDeVault::_convertAsset-`

`sToUSDe` and pUSDeVault::previewYield, `including when the` yUSDe‘ state is not initialized as so will be equal to the zero address.

7.5 Informational

7.5.1 Use named mappings to explicitly denote the purpose of keys and values

Description: Use named mappings to explicitly denote the purpose of keys and values:

```
predeposit/MetaVault.sol
23:    // Track the assets in the mapping for easier access
24:    mapping(address => TAsset) public assetsMap;

predeposit/pUSDeDepositor.sol
35:    mapping (address => TAutoSwap) autoSwaps;

test/MockStakedUSDe.sol
20:  mapping(address => UserCooldown) public cooldowns;
```

Strata: Fixed in commit ab231d9.

Cyfrin: Verified.

7.5.2 Disable initializers on upgradeable contracts

Description: Disable initializers on upgradeable contracts:

- yUSDeVault
- yUSDeDepositor
- pUSDeVault
- pUSDeDepositor

```
+    /// @custom:oz-upgrades-unsafe-allow constructor
+    constructor() {
+        _disableInitializers();
+    }
```

Strata: Fixed in commit 49060b2.

Cyfrin: Verified.

7.5.3 Don't initialize to default values

Description: Don't initialize to default values as Solidity already does this:

```
predeposit/MetaVault.sol
220:        for (uint i = 0; i < length; i++) {
241:        for (uint i = 0; i < assetsArr.length; i++) {
```

Strata: Fixed in commit 07b471f.

Cyfrin: Verified.

7.5.4 Use explicit sizes instead of `uint`

Description: While `uint` defaults to `uint256`, it is considered good practice to use the explicit types including the size and to avoid using `uint`:

```
predeposit/yUSDeDepositor.sol
65:        uint beforeAmount = asset.balanceOf(address(this));
73:        uint pUSDeShares = pUSDeDepositor.deposit(asset, amount, address(this));

predeposit/MetaVault.sol
```

```
53:        uint baseAssets = IERC4626(token).previewRedeem(tokenAssets);
54:        uint shares = previewDeposit(baseAssets);
70:        uint baseAssets = previewMint(shares);
71:        uint tokenAssets = IERC4626(token).previewWithdraw(baseAssets);
211:        uint balance = IERC20(vaultAddress).balanceOf(address(this));
219:        uint length = assetsArr.length;
220:        for (uint i = 0; i < length; i++) {
240:    function redeemRequiredBaseAssets (uint baseTokens) internal {
241:        for (uint i = 0; i < assetsArr.length; i++) {
243:            uint totalBaseTokens = vault.previewRedeem(vault.balanceOf(address(this)));

predeposit/pUSDeVault.sol
62:            uint total_sUSDe = sUSDe.balanceOf(address(this));
63:            uint total_USDe = sUSDe.previewRedeem(total_sUSDe);
65:            uint total_yield_USDe = total_USDe - Math.min(total_USDe, depositedBase);
67:            uint y_pUSDeShares = balanceOf(caller);
68:            uint caller_yield_USDe = total_yield_USDe.mulDiv(shares, y_pUSDeShares,
↪   Math.Rounding.Floor);
121:            uint sUSDeAssets = sUSDe.previewWithdraw(assets);
138:        uint USDeBalance = USDe.balanceOf(address(this));
171:        uint USDeBalance = USDe.balanceOf(address(this));

predeposit/yUSDeVault.sol
38:        uint pUSDeAssets = super.totalAssets();
39:        uint USDeAssets = _convertAssetsToUSDe(pUSDeAssets, true);
43:    function _convertAssetsToUSDe (uint pUSDeAssets, bool withYield) internal view returns (uint256)
↪   {
44:        uint sUSDeAssets = pUSDeVault.previewRedeem(withYield ? address(this) : address(0),
↪   pUSDeAssets);
45:        uint USDeAssets = sUSDe.previewRedeem(sUSDeAssets);
59:        uint underlyingUSDe = _convertAssetsToUSDe(pUSDeAssets, false);
60:        uint yUSDeShares = _valueMulDiv(underlyingUSDe, totalAssets(), totalAccruedUSDe(),
↪   Math.Rounding.Floor);
74:        uint underlyingUSDe = _valueMulDiv(yUSDeShares, totalAccruedUSDe(), totalAssets(),
↪   Math.Rounding.Ceil);
75:        uint pUSDeAssets = pUSDeVault.previewDeposit(underlyingUSDe);
```

Strata: Fixed in commit 61f5910.

Cyfrin: Verified.

7.5.5 Prefix internal and private function names with _ character

Description: It is considered good practice in Solidity to prefix internal and private function names with _ character. This is done sometimes but not other times; ideally apply this consistently:

```
predeposit/PreDepositPhaser.sol
15:    function setYieldPhaseInner () internal {

predeposit/yUSDeDepositor.sol
54:    function deposit_pUSDe (address from, uint256 amount, address receiver) internal returns
↪   (uint256) {
62:    function deposit_pUSDeDepositor (address from, IERC20 asset, uint256 amount, address receiver)
↪   internal returns (uint256) {

predeposit/PreDepositVault.sol
59:    function onAfterDepositChecks () internal view {
64:    function onAfterWithdrawalChecks () internal view {

predeposit/pUSDeVault.sol
```

```
93:    function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal
↪  override {
115:    function _withdraw(address caller, address receiver, address owner, uint256 assets, uint256
↪  shares) internal override {
177:    function stakeUSDe(uint256 USDeAssets) internal returns (uint256) {

predeposit/yUSDeVault.sol
43:    function _convertAssetsToUSDe (uint pUSDeAssets, bool withYield) internal view returns (uint256)
↪  {
79:    function _deposit(address caller, address receiver, uint256 pUSDeAssets, uint256 shares)
↪  internal override {
86:    function _withdraw(address caller, address receiver, address owner, uint256 pUSDeAssets, uint256
↪  shares) internal override {
101:    function _valueMulDiv(uint256 value, uint256 mulValue, uint256 divValue, Math.Rounding
↪  rounding) internal view virtual returns (uint256) {

predeposit/MetaVault.sol
84:    function _deposit(address token, address caller, address receiver, uint256 baseAssets, uint256
↪  tokenAssets, uint256 shares) internal virtual {
160:    ) internal virtual {
175:    function requireSupportedVault(address token) internal view {
191:    function addVaultInner (address vaultAddress) internal {
209:    function removeVaultAndRedeemInner (address vaultAddress) internal {
231:    function redeemMetaVaults () internal {
240:    function redeemRequiredBaseAssets (uint baseTokens) internal {

predeposit/pUSDeDepositor.sol
92:    function deposit_sUSDe (address from, uint256 amount, address receiver) internal returns
↪  (uint256) {
102:    function deposit_USDe (address from, uint256 amount, address receiver) internal returns
↪  (uint256) {
114:    function deposit_viaSwap (address from, IERC20 token, uint256 amount, address receiver)
↪  internal returns (uint256) {
146:    function getPhase () internal view returns (PreDepositPhase phase) {

test/ethena/StakedUSDe.sol
190:  function _checkMinShares() internal view {
203:    internal
225:    internal
239:  function _updateVestingAmount(uint256 newVestingAmount) internal {
251:  function _beforeTokenTransfer(address from, address to, uint256) internal virtual {

test/ethena/SingleAdminAccessControl.sol
72:  function _grantRole(bytes32 role, address account) internal override returns (bool) {
```

Strata: Fixed in commit b154fec.

Cyfrin: Verified.

7.5.6 Use unchained initializers instead

Description: The direct use of initializer functions rather than their unchained equivalents should be avoided to prevent potential duplicate initialization.

Strate: Fixed in commit def7d36.

Cyfrin: Verified.

7.5.7 Missing zero deposit amount validation

Description: Unlike `pUSDeDepositor::deposit_USDe`, `pUSDeDepositor::deposit_sUSDe` does not enforce that the deposited amount is non zero:

```
require(amount > 0, "Deposit is zero");
```

A similar case is present when comparing `yUSDeDepositor::deposit_pUSDeDepositor` and `yUSDeDepositor::deposit_pUSDe`.

Strata: Fixed in commit 1378b6a.

Cyfrin: Verified.

7.5.8 `PreDepositVault::initialize` should not be exposed as public

Description: `PreDepositVault::initialize` is currently exposed as public. Based on the `pUSDeVault` and `yUSDeVault` implementations that invoke this super function, it is not intended. While this does not appear to be exploitable or cause any issues that prevent initialization, it would be better to mark this base implementation as internal and use the `onlyInitializing` modifier instead.

```
    function initialize(
        address owner_
        , string memory name
        , string memory symbol
        , IERC20 USDe_
        , IERC4626 sUSDe_
        , IERC20 stakedAsset
--  ) public virtual initializer {
++  ) internal virtual onlyInitializing {
        __ERC20_init(name, symbol);
        __ERC4626_init(stakedAsset);
        __Ownable_init(owner_);

        USDe = USDe_;
        sUSDe = sUSDe_;
    }
```

Strata: Fixed in commits 6ac05c2 and def7d36.

Cyfrin: Verified. `PreDepositVault::initialize` is now marked as internal and uses the `onlyInitializing` modifier.

7.5.9 Inconsistency in `currentPhase` between `pUSDeVault` and `yUSDeVault`

Description: Both `pUSDeVault` and `yUSDeVault` inherit the `PreDepositVault` which in turn inherits the `PreDepositPhaser`; however, there is an inconsistency between the state of `pUSDe::currentPhase`, which is updated when the phase changes, and `yUSDe::currentPhase`, which is never updated and is thus always the default `PointsPhase` variant. This is assumedly not an issue given that this state is never needed for the yUSDe vault, though a view function is exposed by virtue of the state variable being public which could cause confusion.

Recommended Mitigation: The simplest solution would be modifying this state to be internal by default and only expose the corresponding view function within `pUSDeVault`.

Strata: Fixed in commit aac3b61.

Cyfrin: Verified. The `yUSDeVault` now returns the `pUSDeVault` phase state.

7.6 Gas Optimization

7.6.1 Cache identical storage reads

Description: As reading from storage is expensive, it is more gas-efficient to cache values and read them from the cache if the storage has not changed. Cache identical storage reads:

`PreDepositPhaser.sol`:

```
// use PreDepositPhase.YieldPhase instead
19:         emit PhaseStarted(currentPhase);
```

`pUSDeDepositor.sol`:

```
// cache sUSDe and pUSDe to save 3 storage reads
// also change `deposit` to cache `sUSDe` and pass it as input to `deposit_sUSDe` saves 1 more storage
↪  read
96:             SafeERC20.safeTransferFrom(sUSDe, from, address(this), amount);
98:         sUSDe.approve(address(pUSDe), amount);
99:         return IMetaVault(address(pUSDe)).deposit(address(sUSDe), amount, receiver);

// cache USDe and pUSDe to save 2 storage reads
// also change `deposit` to cache `USDe` and pass it as input to `deposit_USDe` saves 1 more storage
↪  read
107:             SafeERC20.safeTransferFrom(USDe, from, address(this), amount);
110:         USDe.approve(address(pUSDe), amount);
111:         return pUSDe.deposit(amount, receiver);

// cache USDe to save 2 storage reads
// also change `deposit` to cache `USDe` and `autoSwaps[address(asset)]` then pass them as inputs to
↪  `deposit_viaSwap` saves 2 more storage reads
127:         uint256 USDeBalance = USDe.balanceOf(address(this));
130:             tokenOut: address(USDe),
140:         uint256 amountOut = USDe.balanceOf(address(this)) - USDeBalance;
```

`yUSDeDepositor.sol`:

```
// cache pUSDe and yUSDe to save 2 storage reads
56:             SafeERC20.safeTransferFrom(pUSDe, from, address(this), amount);
58:         pUSDe.approve(address(yUSDe), amount);
59:         return yUSDe.deposit(amount, receiver);
```

`MetaVault.sol`:

```
// cache assetsArr.length
241:         for (uint i = 0; i < assetsArr.length; i++) {
```

Strata: Fixed in commit 9a19939.

Cyfrin: Verified.

7.6.2 Using `calldata` is more efficient to `memory` for read-only external function inputs

Description: Using `calldata` is more efficient to `memory` for read-only external function inputs:

`PreDepositVault`:

```
35:         , string memory name
36:         , string memory symbol
```

Strata Money: "initialize" (__init_Vault) is now internal, so the calldata can't be used with the parameters.

Cyfrin: Acknowledged.

7.6.3 Use named returns where this can eliminate in-function variable declaration

Description: Use named returns where this can eliminate in-function variable declaration:

- `yUSDeVault` : functions `totalAccruedUSDe`, `_convertAssetsToUSDe`, `previewDeposit`, `previewMint`
- `pUSDeVault` : function `previewYield`
- `MetaVault` : functions `deposit`, `mint`, `withdraw`, `redeem`

Strata: Fixed in commits 3241635 and c68a705.

Cyfrin: Verified.

7.6.4 Inline small internal functions only used once

Description: It is more gas efficient to inline small internal functions only used once.

For example `pUSDeDepositor::getPhase` is only called by `deposit_sUSDe`. Changing `deposit_sUSDe` to cache `pUSDe` then use the cached copy in the call to `PreDepositPhaser::currentPhase` saves 1 storage read in addition to saving the function call overhead.

Strata: Fixed in commit 9398379.

Cyfrin: Verified.

7.6.5 `PreDepositVault` checks should fail early

Description: `PreDepositVault` implements after deposit/withdrawal checks to enforce several invariants; however, it is only necessary to check the minimum shares violation after execution of the calling functions. To consume less gas, it is better to split these checks into separate before/after functions and revert early if either deposits or withdrawals are disabled.

```
function onAfterDepositChecks () internal view {
    if (!depositsEnabled) {
        revert DepositsDisabled();
    }
}
function onAfterWithdrawalChecks () internal view {
    if (!withdrawalsEnabled) {
        revert WithdrawalsDisabled();
    }
    if (totalSupply() < MIN_SHARES) {
        revert MinSharesViolation();
    }
}
```

Strata: Acknowledged, as the pause state is considered an edge case, so in normal use users would instead benefit from a single method call for all the required checks.

Cyfrin: Acknowledged.

7.6.6 Superfluous vault support validation can be removed from `pUSDeDepositor::deposit`

Description: If the caller to `pUSDeDepositor::deposit` attempts to deposit a vault token that is not `USDe` or one of those preconfigured with an auto swap path, it will first query `MetaVault::isAssetSupported`:

```
    function deposit(IERC20 asset, uint256 amount, address receiver) external returns (uint256) {
        address user = _msgSender();
        ...
        IMetaVault vault = IMetaVault(address(pUSDe));
@>      if (vault.isAssetSupported(address(asset))) {
            SafeERC20.safeTransferFrom(asset, user, address(this), amount);
            asset.approve(address(vault), amount);
            return vault.deposit(address(asset), amount, receiver);
        }
@>      revert InvalidAsset(address(asset));
    }
```

If the specified vault token fails all validation then it falls through to the `InvalidAsset` custom error; however, this is not strictly necessary as `MetaVault::deposit` already performs the same validation within `MetaVault::requireSupportedVault`:

```
    function deposit(address token, uint256 tokenAssets, address receiver) public virtual returns
    ↪   (uint256) {
        if (token == asset()) {
            return deposit(tokenAssets, receiver);
        }
@>      requireSupportedVault(token);
        ...
    }

    function requireSupportedVault(address token) internal view {
        address vaultAddress = assetsMap[token].asset;
        if (vaultAddress == address(0)) {
@>          revert UnsupportedAsset(token);
        }
    }
```

Recommended Mitigation: If it is not intentionally desired to fail early, consider removing the superfluous validation to save gas in the happy path case:

```
function deposit(IERC20 asset, uint256 amount, address receiver) external returns (uint256) {
        address user = _msgSender();
        ...
        IMetaVault vault = IMetaVault(address(pUSDe));
--      if (vault.isAssetSupported(address(asset))) {
            SafeERC20.safeTransferFrom(asset, user, address(this), amount);
            asset.approve(address(vault), amount);
            return vault.deposit(address(asset), amount, receiver);
--      }
--      revert InvalidAsset(address(asset));
    }
```

Strata: Fixed in commit 7f0c5dc.

Cyfrin: Verified.

7.6.7 Remove unused return value from `pUSDeVault::stakeUSDe` and explicitly revert if `USDeAssets == 0`

Description: Remove unused return value from `pUSDeVault::stakeUSDe` and explicitly revert if `USDeAssets == 0`.

Strata: Fixed in commit 513d589.

Cyfrin: Verified.

7.6.8 Unnecessarily complex iteration logic in `MetaVault::redeemMetaVaults` can be simplified

Description: `MetaVault::redeemMetaVaults` is currently implemented as a while loop, indexing the first array element and calling `MetaVault::removeVaultAndRedeemInner` which implements a "replace-and-pop" solution for removing elements from the `assetsArr` array:

```
    function removeVaultAndRedeemInner (address vaultAddress) internal {
        // Redeem
        uint balance = IERC20(vaultAddress).balanceOf(address(this));
        if (balance > 0) {
            IERC4626(vaultAddress).redeem(balance, address(this), address(this));
        }

        // Clean
        TAsset memory emptyAsset;
        assetsMap[vaultAddress] = emptyAsset;
        uint length = assetsArr.length;
        for (uint i = 0; i < length; i++) {
            if (assetsArr[i].asset == vaultAddress) {
@>              assetsArr[i] = assetsArr[length - 1];
@>              assetsArr.pop();
                break;
            }
        }
    }

    function redeemMetaVaults () internal {
        while (assetsArr.length > 0) {
@>          removeVaultAndRedeemInner(assetsArr[0].asset);
        }
    }
```

While this logic is still required for use in `MetaVault::removeVault`, where the contract admin can manually remove a single underlying vault, it would be preferable to avoid re-using this functionality for `MetaVault::redeemMetaVaults`. Instead, starting at the final element and walking backwards would preserve the ordering of the array and avoid unnecessary storage writes.

Strata: Fixed in commit fbb6818 and 98bd92d.

Cyfrin: Verified. The logic has been simplified by iterating over the asset addresses, deleting the individual mapping entries, and finally deleting the array.