CANTINA

Stability Contracts

Competition

June 10, 2025

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Stability acts as permissionless, non-custodial and automatic asset management solution based on AI.

From May 23rd to May 29th Cantina hosted a competition based on stability-contracts. The participants identified a total of **15** issues in the following risk categories:

- High Risk: 5
- Medium Risk: 10
- Low Risk: 0
- Gas Optimizations: 0
- Informational: 0

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

Stability DAO has provided fixes for all **high** severity findings and acknowledged all **medium** severity findings as they are planning to fix them in the near future.

3 Findings

3.1 High Risk

3.1.1 Anyone can easily disable fuse mode for `ERC4626Strategies` and DoS Vault withdraw process

Submitted by BengalCatBalu

Severity: High Risk

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

Summary: Fuse mode is when Strategy abruptly stops investing. Here are two functions that work in Fuse Mode for strategy:

```
/// @inheritdoc IStrategy
function emergencyStopInvesting() external onlyGovernanceOrMultisig {
    // slither-disable-next-line unused-return
    _withdrawAssets(total(), address(this));
}


/// @inheritdoc IStrategy
function transferAssets(
    uint amount,
    uint total_,
    address receiver
) external onlyVault returns (uint[] memory amountsOut) {
    _beforeTransferAssets();
    //slither-disable-next-line unused-return
    return StrategyLib.transferAssets(_getStrategyBaseStorage(), amount, total_, receiver);
}
```

During Fuse Mode, the withdrawal process changes slightly - now instead of withdrawing assets from the investment and transferring them to the user, the assets are directly transferred from the strategy balance. Here is the snippet from the `Vault::withdraw` that shows it:

```
uint localTotalSupply = totalSupply();
uint totalValue = _strategy.total();

uint[] memory amountsOut;

{
    address underlying = _strategy.underlying();
    // nosemgrep
    bool isUnderlyingWithdrawal = assets_.length == 1 && underlying != address(0) && underlying == assets_[0];

    // fuse is not triggered
    if (totalValue > 0) {
        uint value = amountShares * totalValue / localTotalSupply;
        if (isUnderlyingWithdrawal) {
            amountsOut = new uint[](1);
            amountsOut[0] = value;
            _strategy.withdrawUnderlying(amountsOut[0], receiver);
        } else {
            amountsOut = _strategy.withdrawAssets(assets_, value, receiver);
        }
    } else {
        if (isUnderlyingWithdrawal) {
            amountsOut = new uint[](1);
            amountsOut[0] = amountShares * IERC20(underlying).balanceOf(address(_strategy)) / localTotalSupply;
            _strategy.withdrawUnderlying(amountsOut[0], receiver);
        } else {
            amountsOut = _strategy.transferAssets(amountShares, localTotalSupply, receiver);
        }
    }

    uint len = amountsOut.length;
    // nosemgrep
    for (uint i; i < len; ++i) {
        if (amountsOut[i] < minAssetAmountsOut[i]) {
            revert ExceedSlippageExactAsset(assets_[i], amountsOut[i], minAssetAmountsOut[i]);
        }
    }
}
```

However, let's look at how we determine whether Strategy fuse mode has arrived on a contract? This is done by calling `uint totalValue = _strategy.total();`. In the `StrategyBase` implementation, the total function is implemented using an internal counter:

```
/// @inheritdoc IStrategy
function total() public view virtual override returns (uint) {
    return _getStrategyBaseStorage().total;
}
```

However, in `4626Strategy` this function is overridden and, as we see, uses balance, a veiled call to `balanceOf`:

```
/// @inheritdoc IStrategy
function total() public view override returns (uint) {
    StrategyBaseStorage storage __$__ = _getStrategyBaseStorage();
    return StrategyLib.balance(__$__._underlying);
}
```

Thus, if you define FuseMode by calling total when interacting with `StrategyBase`, it is safe. But when interacting with `4626Strategy` - this does not work, because any dust donation of underlaying shares to a strategy contract will result in fuse mode not being enabled and withdraw will work on usual scenario.

Impact Explanation: Lets imagine that Strategy enter the Fuse Mode, And here is 100 USDC on the strategy contract. Underlaying 4626 of this strategy is with USDC assets:

- User wants to withdraw from vault his 10 USDC. In fuse mode 10 USDCs would simply be sent to Strategy directly to the user using the `_strategy.transferAssets` call.
- But, right before the withdrawal transaction, someone donates 1 wei of underlaying shares to strategy - so the withdraw process no longer identifies that this Strategy is in fuse mode.

Now the output for the user will be done in the usual scenario - and will obviously fail the slippage protection set by the user, or worse - return 0 USDC to the user. This check can be easily broken too.

Likelihood Explanation: High, because it takes absolutely nothing to carry out an attack. Just 1 wei underlaying shares is enough for DoS output in Fuse Mode.

Recommendation: Fuse mode detection should be done not only on total value.

Stability DAO: Fixed in PR 311.

3.1.2 Missing Slippage Control in `WrappedMetaVault`

Submitted by 0xprincexyz, also found by sergei2340, JeremiahNoah, Daniel526 and Daniel526

Severity: High Risk

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

Summary: The `WrappedMetaVault` contract lacks slippage control in its deposit and withdraw functions, allowing users to receive fewer shares or assets than expected due to share price manipulation or front-running. This violates best practices for ERC4626 vaults, exposing users to financial losses and breaking expected security guarantees.

Finding Description: The `WrappedMetaVault` contract, which implements the ERC4626 standard, does not enforce minimum output checks (slippage control) for deposits and withdrawals. Slippage control ensures that users receive at least a specified minimum number of shares (minShares) during deposits or assets (minAssets) during withdrawals, protecting against share price changes due to front-running, market volatility, or malicious manipulation (e.g., share inflation attacks).

Users expect to receive shares or assets proportional to the vault's current share price, as estimated by `previewDeposit` or `previewWithdraw`. Without slippage control, users can receive significantly less due to external manipulation, violating fairness. Protection Against Manipulation: The absence of slippage checks allows attackers to manipulate the vault's share price (e.g., by donating assets to the underlying metaVault), causing financial losses for users. Reliability: External contracts relying on predictable deposit/withdrawal outcomes may fail if the share price changes unexpectedly, breaking integration reliability.

```
function _deposit(address caller, address receiver, uint assets, uint shares) internal override {
    WrappedMetaVaultStorage storage $ = _getWrappedMetaVaultStorage();
    if ($.isMulti) {
        address _metaVault = $.metaVault;
        address[] memory _assets = new address[](1);
        _assets[0] = asset();
        uint[] memory amountsMax = new uint[](1);
        amountsMax[0] = assets;
        IERC20(_assets[0]).safeTransferFrom(caller, address(this), assets);
        _mint(receiver, shares); // No minShares check
        IERC20(_assets[0]).forceApprove(_metaVault, assets);
        IStabilityVault(_metaVault).depositAssets(_assets, amountsMax, 0, address(this));
        emit Deposit(caller, receiver, assets, shares);
    } else {
        super._deposit(caller, receiver, assets, shares);
    }
}
```

The function mints shares without checking if they meet a minimum threshold, leaving users vulnerable to receiving fewer shares if the metaVault's share price is manipulated.

```
function _withdraw(address caller, address receiver, address owner, uint assets, uint shares) internal
↪  override {
    WrappedMetaVaultStorage storage $ = _getWrappedMetaVaultStorage();
    if ($.isMulti) {
        if (caller != owner) {
            _spendAllowance(owner, caller, shares);
        }
        address[] memory _assets = new address[](1);
        _assets[0] = asset();
        IStabilityVault($.metaVault).withdrawAssets(
            _assets,
            (assets + 1) * 10 ** (18 - IERC20Metadata(asset()).decimals()),
            new uint[](1),
            receiver,
            address(this)
        ); // No minAssets check
        _burn(owner, shares);
        emit Withdraw(caller, receiver, owner, assets, shares);
    } else {
        super._withdraw(caller, receiver, owner, assets, shares);
    }
}
```

The function withdraws assets without ensuring the actual assets received meet a minimum threshold, exposing users to losses if the share price drops.

Impact Explanation: Financial Loss: Users can lose significant value due to receiving fewer shares or assets than expected. For example, in the proof of concept, a user depositing 100 tokens received ~90.91% fewer shares, equivalent to a 1,000-token loss at the inflated share price.

Likelihood Explanation: The attack requires an attacker to monitor the mempool and execute a front-running transaction. This is feasible in Ethereum's public mempool, especially with automated MEV bots.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {Test, console} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {ERC4626} from "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

import {WrappedMetaVault} from "../../src/core/vaults/WrappedMetaVault.sol";

/**
 * @notice Mock ERC20 token for testing
 */
contract MockERC20 is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

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

/**
 * @notice Mock ERC4626 vault that demonstrates the vulnerability
 */
contract MockERC4626Vault is ERC4626 {

    constructor(IERC20 asset, string memory name, string memory symbol)
        ERC20(name, symbol)
        ERC4626(asset)
    {}

    function totalAssets() public view override returns (uint256) {
        // Return the actual balance of the underlying asset
        return IERC20(asset()).balanceOf(address(this));
    }

    function mint(address to, uint256 amount) external {
```

```
        _mint(to, amount);
    }

    // Simulate asset donation for manipulation attacks
    function simulateAssetDonation(uint256 amount) external {
        // This simulates someone directly transferring assets to the vault
        // which inflates the share price without minting new shares
        IERC20(asset()).transferFrom(msg.sender, address(this), amount);
    }

    // Function with slippage protection for demonstration
    function depositWithSlippage(uint256 assets, address receiver, uint256 minSharesOut) external returns
    ↪  (uint256) {
        uint256 shares = previewDeposit(assets);
        require(shares >= minSharesOut, "Slippage tolerance exceeded");
        return deposit(assets, receiver);
    }

    // Function with slippage protection for demonstration
    function withdrawWithSlippage(uint256 assets, address receiver, address owner, uint256 maxSharesBurn)
    ↪  external returns (uint256) {
        uint256 shares = previewWithdraw(assets);
        require(shares <= maxSharesBurn, "Slippage tolerance exceeded");
        return withdraw(assets, receiver, owner);
    }
}

/**
 * @title WrappedMetaVault Missing Slippage Control PoC
 * @notice Demonstrates that WrappedMetaVault lacks slippage protection for deposits and withdrawals,
 *         making users vulnerable to front-running and MEV attacks.
 */
contract WrappedMetaVaultSlippageControlPoCTest is Test {

    MockERC4626Vault public vault;
    MockERC20 public asset;

    address public user = makeAddr("user");
    address public attacker = makeAddr("attacker");

    uint256 public constant INITIAL_SUPPLY = 1000000e18;
    uint256 public constant USER_DEPOSIT = 100e18; // 100 tokens
    uint256 public constant ATTACKER_AMOUNT = 10000e18; // 10,000 tokens

    function setUp() public {
        // Deploy mock asset and vault to demonstrate the vulnerability
        asset = new MockERC20("Test Token", "TEST");
        vault = new MockERC4626Vault(asset, "Test Vault", "TVAULT");

        // Give tokens to participants
        asset.mint(user, USER_DEPOSIT * 10);
        asset.mint(attacker, ATTACKER_AMOUNT * 10);

        // Initial vault setup with some liquidity to establish 1:1 ratio
        // First mint assets to the vault
        asset.mint(address(this), 1000e18);
        asset.approve(address(vault), 1000e18);

        // Deposit to establish initial 1:1 ratio (1000 assets = 1000 shares)
        vault.deposit(1000e18, address(this));

        vm.label(user, "User");
        vm.label(attacker, "Attacker");
        vm.label(address(vault), "MockVault");
        vm.label(address(asset), "TestToken");
    }

    /**
     * @notice Basic test to verify vault setup works correctly
     */
    function test_basicVaultFunctionality() public {
        console.log("=== Basic Vault Functionality Test ===");

        console.log("Initial vault state:");
        console.log("   Total assets:", vault.totalAssets() / 1e18);
        console.log("   Total supply:", vault.totalSupply() / 1e18);
```

```
        console.log("   Share price:", vault.convertToAssets(1e18) / 1e18);

        // Test basic deposit
        vm.startPrank(user);
        asset.approve(address(vault), USER_DEPOSIT);
        uint256 shares = vault.deposit(USER_DEPOSIT, user);
        vm.stopPrank();

        console.log("\nAfter user deposit:");
        console.log("   User deposited:", USER_DEPOSIT / 1e18, "tokens");
        console.log("   User received:", shares / 1e18, "shares");
        console.log("   New total assets:", vault.totalAssets() / 1e18);
        console.log("   New total supply:", vault.totalSupply() / 1e18);

        assertTrue(shares > 0, "User should receive shares");
        assertTrue(vault.balanceOf(user) == shares, "User should own the shares");
    }

    /**
     * @notice PRACTICAL DEMO: Shows how user loses money due to share inflation attack
     * Attacker donates assets to inflate share price, then user gets fewer shares
     */
    function test_depositFrontRunningAttack() public {
        console.log("=== PRACTICAL DEMO: Share Inflation Attack on Deposit ===");

        // Step 1: Show initial state
        console.log("\n1. Initial State:");
        console.log("   Vault total assets:", vault.totalAssets() / 1e18);
        console.log("   Vault total supply:", vault.totalSupply() / 1e18);
        console.log("   Share price (assets per share):", vault.convertToAssets(1e18) / 1e18);

        // Step 2: User calculates expected shares at current rate
        uint256 expectedShares = vault.previewDeposit(USER_DEPOSIT);
        console.log("\n2. User's Expectation (calculated off-chain):");
        console.log("   User wants to deposit:", USER_DEPOSIT / 1e18, "tokens");
        console.log("   Expected shares at current rate:", expectedShares / 1e18);

        // Step 3: Attacker front-runs by donating assets to inflate share price
        console.log("\n3. Attacker Front-Runs with Share Inflation Attack:");
        vm.startPrank(attacker);

        // Attacker donates assets directly to vault to inflate share price
        // This increases totalAssets() without increasing totalSupply()
        asset.approve(address(vault), ATTACKER_AMOUNT);
        vault.simulateAssetDonation(ATTACKER_AMOUNT); // Donate 10,000 tokens

        console.log("   Attacker donates:", ATTACKER_AMOUNT / 1e18, "tokens to vault");
        console.log("   NEW total assets:", vault.totalAssets() / 1e18);
        console.log("   Total supply unchanged:", vault.totalSupply() / 1e18);
        console.log("   NEW share price after attack:", vault.convertToAssets(1e18) / 1e18);
        vm.stopPrank();

        // Step 4: User's transaction executes at inflated price
        console.log("\n4. User's Transaction Executes (at inflated price):");
        vm.startPrank(user);
        asset.approve(address(vault), USER_DEPOSIT);
        uint256 actualShares = vault.deposit(USER_DEPOSIT, user);
        console.log("   User deposits:", USER_DEPOSIT / 1e18, "tokens");
        console.log("   User gets shares:", actualShares / 1e18);
        vm.stopPrank();

        // Step 5: Calculate user's loss
        console.log("\n5. USER'S FINANCIAL LOSS:");
        if (actualShares < expectedShares) {
            uint256 shareLoss = expectedShares - actualShares;
            uint256 dollarLoss = vault.convertToAssets(shareLoss);
            console.log("   Expected shares:", expectedShares / 1e18);
            console.log("   Actual shares received:", actualShares / 1e18);
            console.log("   Shares lost:", shareLoss / 1e18);
            console.log("   Dollar value lost:", dollarLoss / 1e18, "tokens");
            console.log("   Loss percentage:", (shareLoss * 100) / expectedShares, "%");

            console.log("\n[CRITICAL] User lost money due to NO SLIPPAGE PROTECTION!");
            console.log("If ERC4626 had minSharesOut parameter, user could have protected themselves");

            // Verify the user actually lost money
```

```
            assertTrue(actualShares < expectedShares, "User should have received fewer shares");
            assertTrue(dollarLoss > 0, "User should have lost money");
        } else {
            console.log("   Expected shares:", expectedShares / 1e18);
            console.log("   Actual shares received:", actualShares / 1e18);

            // Calculate the percentage difference
            uint256 percentageDiff = actualShares > expectedShares ?
                ((actualShares - expectedShares) * 100) / expectedShares :
                ((expectedShares - actualShares) * 100) / expectedShares;

            console.log("   Difference:", percentageDiff, "%");

            if (actualShares != expectedShares) {
                console.log("\n[VULNERABILITY DEMONSTRATED] Share inflation attack changed user's outcome!");
                console.log("Even small differences show the attack vector works");
                console.log("In a real scenario with larger amounts, losses would be significant");
            }

            // Still demonstrate the vulnerability exists
            console.log("\n[VULNERABILITY] ERC4626 functions lack slippage protection!");
            console.log("Users cannot protect themselves from share price manipulation");
        }
    }
}
```

Output:

```
=== PRACTICAL DEMO: Share Inflation Attack on Deposit ===

1. Initial State:
Vault total assets: 1000.
Vault total supply: 1000.
Share price (assets per share): 1.

2. User's Expectation (calculated off-chain):
User wants to deposit: 100 tokens.
Expected shares at current rate: 100.

3. Attacker Front-Runs with Share Inflation Attack:
Attacker donates: 10000 tokens to vault.
NEW total assets: 11000.
Total supply unchanged: 1000.
NEW share price after attack: 10.

4. User's Transaction Executes (at inflated price):
User deposits: 100 tokens.
User gets shares: 9.

5. USER'S FINANCIAL LOSS:
Expected shares: 100.
Actual shares received: 9.
Shares lost: 90.
Dollar value lost: 1000 tokens.
Loss percentage: 90 %.

[CRITICAL] User lost money due to NO SLIPPAGE PROTECTION!
```

Recommendation: To fix the missing slippage control, modify the deposit and withdraw functions to include `minShares` and `minAssets` parameters, ensuring users receive at least the expected outputs.

Stability DAO: Fixed in PR 309.

3.1.3 Incorrect slippage check in meta vault deposit

Submitted by BengalCatBalu, also found by willycode20, 0xgh0st, mussucal, sabanaku, YanecaB, chainsentry, Z-Bra and Nexarion

Severity: High Risk

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

Summary: Lets consider this part from deposit process in meta vault:

```
(uint targetVaultPrice,) = IStabilityVault(v.targetVault).price();
uint targetVaultSharesAfter = IERC20(v.targetVault).balanceOf(address(this));
uint depositedTvl = (targetVaultSharesAfter - targetVaultSharesBefore) * targetVaultPrice / 1e18;
uint balanceOut = _usdAmountToMetaVaultBalance(depositedTvl); // in peg asset
uint sharesToCreate;
if (v.totalSharesBefore == 0) {
    sharesToCreate = balanceOut;
} else {
    sharesToCreate = _amountToShares(balanceOut, v.totalSharesBefore, v.totalSupplyBefore);
}

_mint($, receiver, sharesToCreate, balanceOut);

if (balanceOut < minSharesOut) {
    revert ExceedSlippage(balanceOut, minSharesOut);
}
```

The comparison of `balanceOut` and `minSharesOut` is used as a slippage check. However, these are variables of quite different indicators. `balanceOut` measures the amount in pegAsset that came on the contract, while `minSharesOut` indicates how many meta vault shares were issued. As we can see in the second case - `sharesToCreate` may not be equal to `balanceOut`.

Impact Explanation: Slippage check is broken.

Likelihood Explanation: High. It occurs in most cases.

Recommendation: Change `balanceOut` to `sharesToCreate`.

Stability DAO: Fixed in PR 309.

3.1.4 `MetaVault` not resetting allowance of refunded tokens allows hackers to steal tokens with malicious parameters in deposit call

Submitted by rscodes

Severity: High Risk

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

Summary: In `depositAssets` of `MetaVault.sol`:

```
for (uint i; i < v.len; ++i) {
    IERC20(assets_[i]).safeTransferFrom(msg.sender, address(this), amountsMax[i]);
    v.balanceBefore[i] = IERC20(assets_[i]).balanceOf(address(this));
    IERC20(assets_[i]).forceApprove(v.targetVault, amountsMax[i]);
}
uint targetVaultSharesBefore = IERC20(v.targetVault).balanceOf(address(this));
IStabilityVault(v.targetVault).depositAssets(assets_, amountsMax, 0, address(this));
for (uint i; i < v.len; ++i) {
    v.amountsConsumed[i] = v.balanceBefore[i] - IERC20(assets_[i]).balanceOf(address(this));
    uint refund = amountsMax[i] - v.amountsConsumed[i];
    if (refund != 0) {
        IERC20(assets_[i]).safeTransfer(msg.sender, refund);
    }
}
```

We can see that assets not used by the vault deposits are refunded to the user, however the approval of the excess unused allowance given to the target vault by the metaVault isn't decremented. The thing is `MetaVault.sol` does not check that `asset_` (which is provided by the user) is indeed the asset required. Hence, consider the attack vector where the Hacker provides an `asset_[0]` as a worthless fake token. And when `MetaVault` calls the `targetVault`'s `depositAssets`, the target vault uses the previous dangling approval to take asset tokens lying in MetaVault while treating the Hacker as the one that deposited it, so MetaVault will award shares to the Hacker as per normal at the end of the fuction. This works because if we look at target vault's `depositAssets`, going to `VaultBase.sol`:

```
// In VaultBase.sol, when assets_[0] != underlying, this part is ran
(v.amountsConsumed, v.value) = v.strategy.previewDepositAssetsWrite(assets_, amountsMax);
// nosemgrep
for (uint i; i < v.len; ++i) {
    IERC20(v.assets[i]).safeTransferFrom(msg.sender, address(v.strategy), v.amountsConsumed[i]);
}
```

`previewDepositAssetsWrite` is in `StrategyBase.sol` and if you trace there you can see that it shaves away the first parameter `assets_` and just uses `amountsMax` to calculate the return value. Then `IERC20(v.assets[i]).safeTransferFrom...` runs. Note that `v.assets[i]` is not the user provided value and hence it is the real asset, which means target vault will take meta vault's token using the dangling approval and the transaction will run smoothly. Back to the original `depositAssets` of `MetaVault.sol` the code then awards the shares to the hacker.

Note: Note that `MetaVault.sol` is designed to hold asset tokens. You can look at `tvl()` where the asset balance is added:

```
function tvl() public view returns (uint tvl_, bool trusted_) {
    // ...

    // get TVL of assets on contract balance
    address[] memory _assets = $.assets.values();
    len = _assets.length;
    uint[] memory assetsOnBalance = new uint[](len);
    for (uint i; i < len; ++i) {
        assetsOnBalance[i] = IERC20(_assets[i]).balanceOf(address(this));
    }
    (uint assetsTvlUsd,,, bool trustedAssetsPrices) = priceReader.getAssetsPrice(_assets, assetsOnBalance);
    tvl_ += assetsTvlUsd;
    // ...
}
```

And it makes sense as the assets invested into the strategy vaults are **under the metaVault**'s address so any asset rewards claimed would be going to the metaVault address' balance.

Impact: Asset token balance in Meta Vault can be stolen by Attackers. By providing fake tokens, meta vault is tricked into depositing the current real asset tokens in its balance and giving share ownership to the Attacker.

Recommendation: Check that the `assets_` parameter matches the real intended asset token since it is user provided.

Stability DAO: Fixed in PR 309.

3.1.5 Calculation formula for revenue is wrong

Submitted by rscodes, also found by ngochungp295, Aamirusmani1552, wellbyt3, kalyanSingh, 0xMSF14, 0xG0P1, korok, 0xDeoGratias, sergei2340, 0xerenyeager, Agontuk1, chainsentry, evmninja and HeckerTrieuTien

Severity: High Risk

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

Summary: During a `doHardWork()` call in `StrategyBase.sol`, `_claimRevenue()` a function in `ERC4626StrategyBase.sol` is called. `_claimRevenue` uses `_getRevenue` to calculate the revenue.

```
function _getRevenue(
    uint newSharePrice,
    address u
) internal view returns (address[] memory __assets, uint[] memory amounts) {
    ERC4626StrategyBaseStorage storage $ = _getERC4626StrategyBaseStorage();
    StrategyBaseStorage storage __$__ = _getStrategyBaseStorage();
    __assets = __$__._assets;
    amounts = new uint[](1);
    uint oldSharePrice = $.lastSharePrice;
    // nosemgrep
    if (newSharePrice > oldSharePrice && oldSharePrice != 0) {
->      amounts[0] = StrategyLib.balance(u) * newSharePrice * (newSharePrice - oldSharePrice) / oldSharePrice
↪   / 1e18;
    }
}
```

Revenue is being calculated as the price increase per share, however the current calculation is wrong. `(newSharePrice - oldSharePrice) / oldSharePrice` is the increase in % from the old price, but it is multiplied by `newSharePrice`.

Impact: Wrong amount of revenue is being used for the compounding work in `doHardWork()`.

Recommendation: Fix the formula, the increase in % should be applied on the original value `oldSharePrice`.

Stability DAO: Fixed in PR 309.

3.2 Medium Risk

3.2.1 Potential Precision Loss and Unfair Share Allocation for Early Depositors

Submitted by evmninja, also found by rscodes, bl4ck4non, CoheeYang, space image and skippyBrussels

Severity: Medium Risk

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

Summary: The initialisation of share calculation parameters by the first depositor can lead to significant precision loss for subsequent depositors if the first deposit is minimal. This could result in later depositors receiving fewer shares than merited or even zero shares for small deposits. Related Asset: MetaVault.sol. Related Code:

```
// ...existing code...
function depositAssets(
    address[] memory assets_,
    uint[] memory amountsMax,
    uint minSharesOut,
    address receiver
) external nonReentrant {
// ...existing code...
    uint sharesToCreate;
    if (v.totalSharesBefore == 0) {
        sharesToCreate = balanceOut; // Vulnerable point: if balanceOut is small
    } else {
        sharesToCreate = _amountToShares(balanceOut, v.totalSharesBefore, v.totalSupplyBefore);
    }

    _mint($, receiver, sharesToCreate, balanceOut);
// ...existing code...
}

function _amountToShares(uint amount, uint totalShares_, uint totalSupply_) internal pure returns (uint) {
    if (totalSupply_ == 0) { // This check prevents division by zero if called when totalSupply is 0
        return 0;
    }
    return amount * totalShares_ / totalSupply_; // Precision loss if totalShares_ is small
}
// ...existing code...
```

Finding Description: The MetaVault contract calculates user balances and mints shares based on a ratio of the total value locked (TVL) to an internal accounting unit called `totalShares`. When the first user deposits assets, the `totalShares` is initialised based on the value of this first deposit. Specifically, in the `depositAssets` function (and similarly in `previewDepositAssets`), if `totalSharesBefore` is zero, the internal `sharesToCreate` is set directly to `balanceOut` (the external token amount derived from the deposited value).

If this initial `balanceOut` is an extremely small number (e.g., 1 wei of the MetaVault token), `totalShares` will also be set to this small number. Subsequent share calculations, such as `_amountToShares(amount, totalShares, totalSupply_)`, which effectively calculates `(amount * totalShares_) / totalSupply_`, will suffer from precision loss due to integer division. If `totalShares_` is 1, this becomes `amount / totalSupply_`. If amount is less than `totalSupply_`, the result will be zero internal shares, meaning the depositor loses their deposited assets (in terms of receiving shares) or receives a disproportionately small number of shares.

Step-by-step analysis:

1. The `depositAssets` function is called when the MetaVault is empty (`$.totalShares == 0`).
2. The value of the deposit is determined as `depositedTvl`, then converted to `balanceOut` (MetaVault's external token amount).
3. Because `v.totalSharesBefore == 0`, `sharesToCreate` is set to `balanceOut` (line 261 in MetaVault.sol).
4. `_mint` is called, and `$.totalShares` becomes `balanceOut`.

5. Consider `balanceOut` (and thus `$.totalShares`) is 1 (e.g., first deposit was 1 wei).
6. Later, `totalSupply()` (derived from TVL) might grow significantly due to appreciation or further large deposits that manage to get some shares. Let's say `totalSupply()` becomes 1,000,000,000.
7. A new user attempts to deposit an amount of, say, 500,000.
8. `_amountToShares(500000, 1, 1000000000)` calculates `(500000 * 1) / 1000000000`, which results in 0 due to integer division.
9. The new user would be minted 0 internal shares and thus their `balanceOf` would be 0, effectively losing their deposit to the vault or providing free exit liquidity for others.

Impact Explanation: Impact: High. Users could receive zero shares for their deposits or a significantly unfair amount, potentially leading to a loss of deposited funds from their perspective as they hold no claim.

Likelihood Explanation: Likelihood: Low to Medium. Requires a very small initial deposit. While tests show large initial deposits, the contract does not enforce a minimum first deposit or a robust initial share issuance strategy. Automated systems or uninformed users could trigger this.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import {Test, console} from "forge-std/Test.sol";
import {IERC20, IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {IMetaVault, IStabilityVault} from "../../src/interfaces/IMetaVault.sol";
import "../core/MetaVault.Sonic.t.sol";  // Importing the existing test setup

contract MetaVaultPrecisionLossTest is MetaVaultSonicTest {
    function setUp() public override {
        super.setUp();
    }

    function test_precision_loss_attack() public {
        address attacker = address(0xBAD);
        address victim = address(0xDEAD);

        // Select a simple MetaVault (the first one in our array)
        address metaVault = metaVaults[0];
        address[] memory assets = IMetaVault(metaVault).assetsForDeposit();

        // Step 1: Attacker makes a minimal deposit to establish very low totalShares
        // Calculate a deposit amount that would result in just 1 wei of shares
        uint[] memory tinyDeposit = new uint[](assets.length);

        // For USDC (6 decimals), this would be a very small amount
        // This needs adjustment based on the asset's price and decimals
        tinyDeposit[0] = 1;  // 1 wei of USDC or whatever the asset is

        _dealAndApprove(attacker, metaVault, assets, tinyDeposit);

        vm.startPrank(attacker);
        // Make initial deposit - this sets totalShares = 1
        IStabilityVault(metaVault).depositAssets(assets, tinyDeposit, 0, attacker);
        vm.stopPrank();

        // Verify attacker received minimal shares
        uint attackerBalance = IERC20(metaVault).balanceOf(attacker);
        console.log("Attacker's balance:", attackerBalance);
        // Should be extremely small (potentially 1 wei)

        // Step 2: Record the vault's state after first deposit
        (uint sharePrice,,,) = IMetaVault(metaVault).internalSharePrice();
        console.log("Share price after attack:", sharePrice);

        // Step 3: Now victim makes a small but legitimate deposit
        uint[] memory smallDeposit = new uint[](assets.length);
        // Small deposit but not tiny - just a fraction of a USDC (or whatever asset)
        smallDeposit[0] = 1000; // 0.001 USDC if 6 decimals

        _dealAndApprove(victim, metaVault, assets, smallDeposit);
```

```
        vm.startPrank(victim);
        IStabilityVault(metaVault).depositAssets(assets, smallDeposit, 0, victim);
        vm.stopPrank();

        // Check victim's received shares
        uint victimBalance = IERC20(metaVault).balanceOf(victim);
        console.log("Victim's balance:", victimBalance);

        // The victim might receive zero shares due to precision loss:
        // calculation is: amount * totalShares / totalSupply
        // With totalShares = 1 and totalSupply >> 1, this rounds to 0

        // To make this even more clear, let's attempt to withdraw and show that
        // the victim has essentially lost their deposit
        vm.roll(block.number + 6); // Move forward blocks to pass flash loan protection

        vm.startPrank(victim);
        if (victimBalance > 0) {
            console.log("Victim can withdraw assets");
            IStabilityVault(metaVault).withdrawAssets(assets, victimBalance, new uint[](assets.length));
        } else {
            console.log("Victim CANNOT withdraw anything - deposit is lost");
            // Would fail if we tried to withdraw with balance = 0
        }
        vm.stopPrank();

        // Show the attacker still has their shares and can withdraw
        vm.roll(block.number + 6);
        vm.startPrank(attacker);
        if (attackerBalance > 0) {
            // Attacker can withdraw, potentially getting both their deposit AND the victim's
            IStabilityVault(metaVault).withdrawAssets(assets, attackerBalance, new uint[](assets.length));
            console.log("Attacker successfully withdrew funds");
        }
        vm.stopPrank();
    }
}
```

Recommendation: The most standard fix is to mint an initial amount of shares (e.g., 1000 * 10^18) to the zero address during the initialize function. This ensures totalShares is never critically low.

```
// In initialize function:
// ...existing code...
function initialize(
    // ... params ...
) public initializer {
    __Controllable_init(platform_);
    __ReentrancyGuard_init();
    MetaVaultStorage storage $ = _getMetaVaultStorage();
    // ... other initializations ...
    $.pegAsset = pegAsset_;
    $.name = name_;
    $.symbol = symbol_;

    // Mint initial shares to prevent precision issues with the first deposit
    uint initialDeadShares = 1000 * (10**decimals()); // e.g., 1000 full tokens
    if (initialDeadShares > 0) { // Ensure decimals() doesn't make it zero if it could be < 18
        $.totalShares += initialDeadShares;
        $.shareBalance[address(0)] += initialDeadShares; // Mint to dead address
        emit Transfer(address(0), address(0), 0); // Emit for share minting, value is 0 as it's initial setup
    }

    emit TargetProportions(proportions_);
}
// ...existing code...
```

By pre-initialising `totalShares` with a reasonably large number, the denominator in `_amountToShares` (which uses `totalSupply_`, which is related to `totalShares_` via share price) will not be disproportionately small compared to the numerator `amount * totalShares_`. This maintains precision and ensures fair share allocation even for small subsequent deposits. The `if (v.totalSharesBefore == 0)` branch in `depositAssets` would then likely not be hit by the first actual user depositor if initial shares are minted in initialize.

3.2.2 Anyone can inflate vault shares value in fuse mode

Submitted by BengalCatBalu, also found by Arno

Severity: Medium Risk

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

Summary: When dust donation to 4626Strategy can change the total and disable checking for fuse mode, allowing stealing all assets during fuse mode. Lets consider deposit process on vault. First and most important, protocol detect Fuse mode with total() call, which can be easily manipulated on 4626Strategies.

```
v._totalSupply = totalSupply();
v.totalValue = v.strategy.total();

// nosemgrep
if (v._totalSupply != 0 && v.totalValue == 0) {
    revert FuseTrigger();
}
```

```
function total() public view override returns (uint) {
    StrategyBaseStorage storage __$__ = _getStrategyBaseStorage();
    return StrategyLib.balance(__$__._underlying);
}
```

That is, during Fuse Mode - total supply != 0, since there were already vault deposits before that. But the attacker can make the total Value whatever he wants with the help of donation. Obviously we will made it minimum - 1 wei. Next, let's look at the mint shares process.

In _calcMintShares totalSupply - is vault shares total supply, value - underlaying shares value received during deposit process, `totalValue = v.totalValue = v.strategy.total();` - That is, the same value we manipulated and it is 1 wei. Obviously - the ouput amount of shares will be (totalSupply * value / 1).

```
/// @dev Calculating amount of new shares for given deposited value and totals
function _calcMintShares(
    uint totalSupply_,
    uint value_,
    uint totalValue_,
    uint[] memory amountsConsumed,
    address[] memory assets_
) internal view returns (uint mintAmount, uint initialShares) {
    if (totalSupply_ > 0) {
        mintAmount = value_ * totalSupply_ / totalValue_;
        initialShares = 0; // hide warning
    } else {
        // calc mintAmount for USD amount of value
        // its setting sharePrice to 1e18
        IPriceReader priceReader = IPriceReader(IPlatform(platform()).priceReader());
        //slither-disable-next-line unused-return
        (mintAmount,,,) = priceReader.getAssetsPrice(assets_, amountsConsumed);

        // initialShares for saving share price after full withdraw
        initialShares = _INITIAL_SHARES;
        if (mintAmount < initialShares * 1000) {
            revert NotEnoughAmountToInitSupply(mintAmount, initialShares * 1000);
        }
        mintAmount -= initialShares;
    }
}
```

And finally let's consider the mint function.

```
function _mintShares(
    VaultBaseStorage storage $,
    uint totalSupply_,
    uint value_,
    uint totalValue_,
    uint[] memory amountsConsumed,
    uint minSharesOut,
    address[] memory assets_,
    address receiver
) internal returns (uint mintAmount) {
    uint initialShares;
    (mintAmount, initialShares) = _calcMintShares(totalSupply_, value_, totalValue_, amountsConsumed, assets_);
    uint _maxSupply = $.maxSupply;
    // nosemgrep
    if (_maxSupply != 0 && mintAmount + totalSupply_ > _maxSupply) {
        revert ExceedMaxSupply(_maxSupply);
    }
    if (mintAmount < minSharesOut) {
        revert ExceedSlippage(mintAmount, minSharesOut);
    }
    if (initialShares > 0) {
        _mint(ConstantsLib.DEAD_ADDRESS, initialShares);
    }
    if (receiver == address(0)) {
        receiver = msg.sender;
    }
    _mint(receiver, mintAmount);
}
```

We see that the only thing that can prevent the attacker from getting 99% of the supply is the max supply constraint. Obviously, when we get `totalSupply * value` - we can easily receive maxSupply - previous_-total_supply of vault shares.

Impact: So, we saw that during fuse mode you can easily get most of the vault shares absolutely for free. Having obtained this value of shares we obviously now claim the Majority of funds of all depositors.

Likelihood Explanation: In order for an attack to be possible, Strategy must go into Fuse Mode - this happens when emergencyStopInvesting is called.

```
/// @notice Emergency stop investing by strategy, withdraw liquidity without rewards.
/// This action triggers FUSE mode.
/// Only governance or multisig can call this.
/// @inheritdoc IStrategy
function emergencyStopInvesting() external onlyGovernanceOrMultisig {
    // slither-disable-next-line unused-return
    _withdrawAssets(total(), address(this));
}
```

After that, the cost of the attack is very very low. 1 wei of underlaying vault shares donation + necessary amount of assets (converted to vault shares) to not exceed maxSupply / totalSupply.

Recommendation: First of all - in deposit function you need to implement better detection of fuse mode.

Stability DAO: Fixed in PR 111.

3.2.3 `CVault.sol` - Inconsistent Initialization Parameter Declaration Causes Deployment Failures

Submitted by BaiMaStryke

Severity: Medium Risk

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

Summary: `CVault` contract's declared initialization parameter requirements are inconsistent with its actual implementation, causing any automated factory-based deployment or integration to inevitably fail. This impacts protocol factories, scaling, and all forms of automated integration.

Finding Description: The `CVault` contract advertises its initialization requirements via the `getUniqueInitParamLength()` function, which claims the vault requires 1 address parameter (`vaultInitAddresses.length == 1`) and 0 numeric parameters (`vaultInitNums.length == 0`). However,

the actual `initialize()` implementation strictly demands **both arrays must be empty**, otherwise it reverts with `IncorrectInitParams()`.

- `getUniqueInitParamLength()` returns `(1, 0)`, advertising that one address is needed.
- `initialize()` requires `vaultInitAddresses.length == 0 && vaultInitNums.length == 0`, or it reverts.
- Any factory or automated deployer that relies on the interface will fill parameters as declared and be rejected by the implementation logic, causing an immediate revert.

This logical inconsistency breaks all interface-based automation, bulk upgrades, and seamless protocol integration, making `CVault` incompatible with common DeFi deployment and integration patterns.

Impact Explanation: This issue falls under Breaks Non-Core Functionality and Temporary DoS:

- Breaks the protocol's ability to scale, upgrade, or be deployed in bulk by any factory or automation system.
- Integrations, testing frameworks, and third-party tools rely on `getUniqueInitParamLength()` and will universally fail.
- While core protocol security is not directly impacted, usability and extensibility are severely compromised.

According to the Cantina Severity Matrix:

> "Breaks Non-Core Functionality" and "Temporary Disruption or DoS" are categorized as Medium severity.

Likelihood Explanation:

- Extremely high: Any factory, upgrade system, or DeFi integration relying on the declared parameters will hit this bug and fail.
- Practically guaranteed: This issue will always occur for any attempt at protocol extension or automated deployment; it is not an edge case.
- This is not a theoretical low-likelihood bug; it directly affects all standard integration and deployment workflows.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {Test} from "forge-std/Test.sol";
import {CVault, IVault} from "../../src/core/vaults/CVault.sol";
import {IControllable} from "../../src/interfaces/IControllable.sol";

contract CVaultInitParamMismatchPoC is Test {
    function test_CVault_getUniqueInitParamLength_returns_wrong_values() public {
        CVault impl = new CVault();
        (uint addrCount, uint numCount) = impl.getUniqueInitParamLength();
        assertEq(addrCount, 1, "getUniqueInitParamLength declares 1 address needed");
        assertEq(numCount, 0, "getUniqueInitParamLength declares 0 nums needed");
    }

    function test_CVault_mock_implementation_direct_call() public {
        // Create a mock CVault that bypasses initializer for testing
        MockCVaultTestable mock = new MockCVaultTestable();
        // Test 1: Wrong params (should revert)
        address[] memory wrongAddrs = new address[](1);
        wrongAddrs[0] = address(0x1234);
        uint[] memory wrongNums = new uint[](0);
        IVault.VaultInitializationData memory wrongData = IVault.VaultInitializationData({
            platform: address(0x1),
            strategy: address(0x2),
            name: "Wrong Vault",
            symbol: "WRONG",
            tokenId: 1,
            vaultInitAddresses: wrongAddrs,
            vaultInitNums: wrongNums
```

```
        });
        vm.expectRevert(IControllable.IncorrectInitParams.selector);
        mock.testInitializeParams(wrongData);
        // Test 2: Correct params (should pass)
        address[] memory correctAddrs = new address[](0);
        uint[] memory correctNums = new uint[](0);
        IVault.VaultInitializationData memory correctData = IVault.VaultInitializationData({
            platform: address(0x1),
            strategy: address(0x2),
            name: "Correct Vault",
            symbol: "CORRECT",
            tokenId: 1,
            vaultInitAddresses: correctAddrs,
            vaultInitNums: correctNums
        });
        mock.testInitializeParams(correctData);
    }
}

contract MockCVaultTestable {
    function testInitializeParams(IVault.VaultInitializationData memory vaultInitializationData) external pure {
        if (vaultInitializationData.vaultInitAddresses.length != 0 ||
        ↪  vaultInitializationData.vaultInitNums.length != 0) {
            revert IControllable.IncorrectInitParams();
        }
    }
}
```

Expected/Observed Output:

- Test 1 (one address param): always reverts with `IncorrectInitParams`, inconsistent with interface declaration.
- Test 2 (zero address param): succeeds, no revert.
- This proves the interface and implementation are fundamentally at odds.

Recommendation: Fix suggestion: Ensure the interface and implementation match.

- If `initialize()` should not require any parameters, set `_UNIQUE_INIT_ADDRESSES` to 0.

```
uint internal constant _UNIQUE_INIT_ADDRESSES = 0;
```

- If one parameter is needed in the future, update `initialize()` to accept and require it (length 1). Regardless, the declaration must match the logic to restore compatibility with factory and automated deployment systems.

3.2.4 New vault addition could freeze all operations

Submitted by 0x15, also found by CoheeYang

Severity: Medium Risk

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

Description: When a new child vault is added via `addVault`, its `totalSupply` remains zero. The `MetaVault.currentProportions()` function divides by each vault's `totalSupply` to compute USD values:

```
vaultUsdValue[i] = vaultSharesBalance * vaultTvl / vaultTotalSupply;
```

Because `vaultTotalSupply == 0` for the fresh vault, every call to `currentProportions()` reverts with a divide-by-zero error. As a result, all higher-level actions (deposits, withdrawals, rebalances) immediately fail, trapping user funds and halting the protocol until someone manually "seeds" the new vault outside the normal UI paths.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {IMetaVault} from "../../src/interfaces/IMetaVault.sol";
import {SonicConstantsLib} from "../../chains/sonic/SonicConstantsLib.sol";
```

```
contract MetaVaultDivideByZeroTest is Test {
    IMetaVault public metaVault;

    function setUp() public {
        vm.selectFork(vm.createFork(vm.envString("SONIC_RPC_URL"), 27965000));
        metaVault = IMetaVault(SonicConstantsLib.METAVAULT_metaUSDC);
    }

    function test_divideByZeroVulnerability() public {
        console.log("=== DIVIDE BY ZERO VULNERABILITY ===");

        // Show normal operation
        try metaVault.currentProportions() returns (uint[] memory proportions) {
            console.log("Normal: currentProportions() works, returned", proportions.length, "proportions");
        } catch {
            console.log("currentProportions() already failing");
        }

        // Demonstrate the problematic calculation from currentProportions():
        // vaultUsdValue[i] = vaultSharesBalance * vaultTvl / vaultTotalSupply
        uint vaultSharesBalance = 1000e18;
        uint vaultTvl = 5000e18;
        uint vaultTotalSupply = 0; // New vault has zero totalSupply

        console.log("New vault scenario - totalSupply:", vaultTotalSupply);

        try this.simulateDivision(vaultSharesBalance, vaultTvl, vaultTotalSupply) {
            console.log("Unexpected: Division succeeded");
        } catch {
            console.log("VULNERABILITY: Division by zero reverts");
            console.log("Impact: All MetaVault operations fail (deposit/withdraw/rebalance)");
            console.log("Result: User funds trapped until vault manually seeded");
        }
    }

    /**
     * @notice Simulate the division calculation that fails in currentProportions()
     */
    function simulateDivision(uint balance, uint tvl, uint totalSupply) external pure returns (uint) {
        return balance * tvl / totalSupply; // This will revert when totalSupply = 0
    }


}
```

Logs:

```
[PASS] test_divideByZeroVulnerability() (gas: 609301)
Logs:
  === DIVIDE BY ZERO VULNERABILITY ===
  Normal: currentProportions() works, returned 5 proportions
  New vault scenario - totalSupply: 0
  VULNERABILITY: Division by zero reverts
  Impact: All MetaVault operations fail (deposit/withdraw/rebalance)
  Result: User funds trapped until vault manually seeded

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.47s (45.17ms CPU time)

Ran 1 test suite in 1.57s (1.47s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: In `currentProportions()`, skip or zero out any vault whose `totalSupply` is zero:

3.2.5 The first vault deposit being an underlying token can cause Share Under-Minting and Silent Value Loss

Submitted by 0xDeoGratias

Severity: Medium Risk

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

Description:`VaultBase::depositAssets()` has two execution branches:

1. "Underlying" branch (single-asset deposit where asset == underlying).

```
v.value    = amountsMax[0];
safeTransferFrom(user -> strategy, v.value);
v.amountsConsumed = strategy.depositUnderlying(v.value);
```

2. "Assets" branch (multi-asset deposit).

```
(v.amountsConsumed, v.value) = strategy.previewDepositAssetsWrite(...);
for (...) safeTransferFrom(user -> strategy, v.amountsConsumed[i]);
```

In the underlying branch the vault blindly transfers `v.value` tokens to the strategy before it knows how many tokens the strategy will actually consume. `depositUnderlying()` is allowed to consume less than the amount supplied and simply returns the smaller `amountsConsumed[0]`.

Later, share-minting uses `amountsConsumed` -- not the full `value` already taken from the user -- to price the deposit:

```
mintAmount = priceReader.getAssetsPrice(assets, amountsConsumed); // first deposit
// or
mintAmount = value_ * totalSupply / totalValue;                   // later deposits
```

A user may pay for N tokens while receiving shares priced on M < N tokens; the surplus silently remains in the strategy and is never credited. Typical scenarios where the mismatch appears:

- The strategy caps the usable amount (liquidity limit, max TVL, etc...).
- Fee-on-transfer or rebasing underlying tokens reduce the received balance.
- Malicious or buggy strategy returns a lower `amountsConsumed`.

Impact:

- First deposit — vault share price is initialised with `amountsConsumed`, so the depositor immediately loses `(value  amountsConsumed)/value` in share value.
- Subsequent deposits -– no dilution, but "excess" tokens accumulate in the strategy, skewing accounting and potentially blocking withdrawals or causing unexpected behaviour.

Proof of Concept: To run this proof of concept create a file called `FirstDepositMismatch.t.sol` and place it under `test/core`. Then run the POC using the command `forge test --mt test_FirstDeposit-SharePriceIsWrong -vv`. What this proof of concept does is that it:

1. Sets-up a fake enviroment.
   - Deploys a `CVault` instance plus a custom strategy called HalfConsumeStrategy.
   - This strategy lies: when the vault forwards X underlying tokens, it pretends it "consumed" only X / 2 and returns that figure from depositUnderlying() and the preview helper.
2. Performs the first deposit.
   - The test account mints 2 LP tokens (representing $2), approves the vault, and calls depositAssets with a single-asset array `[LP]`.
3. Observes Share Minting.
   - Because this is the very first deposit, _calcMintShares() mints shares priced on the reported 1 LP.
   - The test logs and assertions show:
   - `minted` shares 1e18 – 1e15 (1 LP of value).
     - `totalSupply` 1e18.
     - The user therefore paid 2 LP but received shares worth only 1 LP.
4. Assertions pass → bug confirmed.
   - The proof of concept confirms the mismatch between value sent (2 LP) and value credited (1 LP), demonstrating the share under-minting / silent value loss exactly as our report describes.

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

/**
 * @title FirstDepositMismatch.t.sol
 * @notice Minimal PoC that shows the *first-deposit* share-price bug in `VaultBase.depositAssets()`.
 *
 * Scenario
 * --------
 * 1. A special strategy (`HalfConsumeStrategy`) lies and says it only "consumed" **half** of the
 *    underlying the vault already forwarded to it.
 * 2. On the very first deposit the vault mints shares priced with `amountsConsumed` (0.5USD) even
 *    though TVL increased by the full 1USD.
 *
 * Result: the share-price doubles (2USD) and the depositor immediately loses 50%.
 */

import {Test, console}   from "forge-std/Test.sol";
import {FullMockSetup}   from "../base/FullMockSetup.sol";
import {Proxy}           from "../../src/core/proxy/Proxy.sol";
import {CVault}          from "../../src/core/vaults/CVault.sol";
import {MockStrategy}    from "../../src/test/MockStrategy.sol";
import {IVault}          from "../../src/interfaces/IVault.sol";
import {VaultStatusLib}  from "../../src/core/libs/VaultStatusLib.sol";

/// ---------------------------------------------------------------------------
///                              Fake strategy
/// ---------------------------------------------------------------------------
contract HalfConsumeStrategy is MockStrategy {
    uint private _fakeTotal;

    /// Consume exactly half the tokenA tokens that were sent, pretend assets() length = 2
    function depositUnderlying(uint amountMax) public override returns (uint[] memory consumed) {
         _fakeTotal += amountMax;
        consumed = new uint[](2);
        consumed[0] = amountMax / 2; // report half consumed
        consumed[1] = 0;             // none for tokenB
    }

    /// Allow single-asset preview without the usual length check reverting
    function previewDepositAssets(
        address[] memory assets_,
        uint[] memory amountsMax
    ) public view override returns (uint[] memory consumed, uint value) {
        if (assets_.length == 1) {
            consumed = new uint[](2);
            consumed[0] = amountsMax[0] / 2;
            consumed[1] = 0;
            value = consumed[0];
        } else {
            return super.previewDepositAssets(assets_, amountsMax);
        }
    }
    function total() public view override returns (uint) {
        return _fakeTotal;
    }

    function assetsAmounts() public view override returns (address[] memory a, uint[] memory amts) {
        a = new address[](1);
        amts = new uint[](1);
        a[0] = underlying();
        amts[0] = _fakeTotal;           // all value stored
    }
}

/// ---------------------------------------------------------------------------
///                                 Test
/// ---------------------------------------------------------------------------
contract FirstDepositMismatchTest is Test, FullMockSetup {
    CVault              internal vault;
    HalfConsumeStrategy internal strategy;

    function setUp() public {
        // 1. Deploy vault proxy first
        Proxy vaultProxy = new Proxy();
```

```
        vaultProxy.initProxy(address(vaultImplementation));
        vault = CVault(payable(address(vaultProxy)));

        // 2. Deploy and init strategy proxy
        Proxy stratProxy = new Proxy();
        stratProxy.initProxy(address(new HalfConsumeStrategy()));
        strategy = HalfConsumeStrategy(address(stratProxy));

        address[] memory initAddrs = new address[](4);
        initAddrs[0] = address(platform); // platform
        initAddrs[1] = address(vaultProxy); // vault
        initAddrs[2] = address(lp); // mock pool (not needed)
        initAddrs[3] = address(tokenA); // underlying token
        strategy.initialize(initAddrs, new uint[](0), new int24[](0));

        // 3. Initialise the vault
        vault.initialize(
            IVault.VaultInitializationData({
                platform: address(platform),
                strategy: address(strategy),
                name: "Buggy CVault",
                symbol: "bVAULT",
                tokenId: 0,
                vaultInitAddresses: new address[](0),
                vaultInitNums: new uint[](0)
            })
        );

        // 4. Activate vault so deposits do not revert
        address[] memory v = new address[](1);
        v[0] = address(vault);
        uint[] memory st = new uint[](1);
        st[0] = VaultStatusLib.ACTIVE;
        factory.setVaultStatus(v, st);
    }

    /* ---------------------------------------------------------------------- */
    /*                                 P  O  C                                */
    /* ---------------------------------------------------------------------- */

    function test_FirstDepositSharePriceIsWrong() public {
        uint256 depositAmount = 2e18; // 2 tokenA ($2) so that half (1 tokenA) passes min-supply guard
        tokenA.mint(depositAmount);
        tokenA.approve(address(vault), depositAmount);

        address[] memory assets = new address[](1);
        assets[0] = address(tokenA);
        uint[] memory amounts = new uint[](1);
        amounts[0] = depositAmount;

        // Preview should reflect *half* consumption
        (uint[] memory consumed,,) = vault.previewDepositAssets(assets, amounts);
        assertEq(consumed[0], depositAmount / 2, "preview must show half consumed");

        // Do the deposit
        vault.depositAssets(assets, amounts, 0, address(0));

        //  Assertions
        uint256 minted = vault.balanceOf(address(this));
        uint256 totalSupply = vault.totalSupply();
        (uint price_,) = vault.price();

        console.log("Minted shares  ", minted);
        console.log("Total supply   ", totalSupply);
        console.log("Share price USD", price_ / 1e18, ".", price_ % 1e18);

        // Depositor only got roughly half the value back in shares (-initialShares dust)
        uint256 expectedMint = depositAmount / 2 - 1e15; // _INITIAL_SHARES is 1e15
        assertEq(minted, expectedMint, "minted shares should equal half deposit minus initialShares");
        assertEq(totalSupply, depositAmount / 2, "totalSupply should equal half the deposit (expected $1)");

        //The supply / minting mismatch proves the bug., 1e18, also note our share price);
    }
}
```

Output:

```
[PASS] test_FirstDepositSharePriceIsWrong() (gas: 431271)
Logs:
  Minted shares   999000000000000000
  Total supply    1000000000000000000
  Share price USD 2 . 0
```

Recommendation:

1. Enforce full consumption.
2. Implement a similar flow to the multi asset branch.

3.2.6 Withdrawals blocking by griefing attack

Submitted by realsung, also found by willycode20, falconhoof, pyk, Arno, nikhil840096, IAM0TI, rscodes, la0t0ng, CodexBugmeNot, YanecaB, sabanaku, Ravindu Santhush, ABDul Rehman, BratZ and BengalCatBalu

Severity: Medium Risk

Context: MetaVault.sol#L39, MetaVault.sol#L605, MetaVault.sol#L613, MetaVault.sol#L648

Summary: A global withdrawal delay is applied to all users, allowing any user to block withdrawals for everyone by repeatedly triggering the lock, resulting in griefing.

Finding Description: The `MetaVault` contract enforces a 5-block withdrawal delay per user address using the `lastTransferBlock` mapping. However, the `WrappedMetaVault` contract interacts with `MetaVault` using its own contract address (`address(this)`) for all deposits and withdrawals. As a result, all users of the wrapper share the same withdrawal delay state. If any user interacts with the wrapper (deposit/withdraw), the `lastTransferBlock[address(WrappedMetaVault)]` is updated, resetting the withdrawal delay for everyone. This design flaw allows any user to grief all other users by repeatedly depositing or transferring small amounts, effectively blocking all withdrawals from the wrapper for as long as the attacker continues. This breaks the security guarantee of user isolation: one user's actions should not affect another's ability to withdraw.

Impact Explanation:

- The attack is cheap and easy to execute, requiring only repeated small deposits.

Likelihood Explanation:

- The attack can be performed by anyone with access to the wrapper, with no special permissions or large capital required.
- also for this vulnerability to be exploitable, the `lastBlockDefenseDisabled` flag in MetaVault must not be set to true.

Proof of Concept:

1. User A wants to withdraw from WrappedMetaVault but fails if within 5 blocks of last deposit.
2. User B (attacker) repeatedly calls: `deposit(1, attackerAddress); // Wait 5 block, repeat.`
3. This keeps resetting `lastTransferBlock[address(WrappedMetaVault)]`.
4. User A (and all others) can never withdraw as long as this continues.

```
function test_DoS_GlobalWithdrawalLock_PoC() public {
    address metavault = metaVaults[0];
    address wrapper = metaVaultFactory.wrapper(metavault);
    address[] memory assets = IMetaVault(metavault).assetsForDeposit();
    uint[] memory depositAmounts = _getAmountsForDeposit(1000, assets);

    address victim = address(0xC0FFEE);
    address attacker = address(0xBADBEEF);

    _dealAndApprove(victim, metavault, assets, depositAmounts);
    _dealAndApprove(attacker, metavault, assets, depositAmounts);

    vm.startPrank(victim);
    IStabilityVault(metavault).depositAssets(assets, depositAmounts, 0, victim);
    vm.stopPrank();
```

```
    deal(assets[0], victim, depositAmounts[0]);
    vm.startPrank(victim);
    IERC20(assets[0]).approve(wrapper, 0);
    IERC20(assets[0]).approve(wrapper, depositAmounts[0]);
    IWrappedMetaVault(wrapper).deposit(depositAmounts[0], victim);
    uint wrapperShares = IERC20(wrapper).balanceOf(victim);
    vm.stopPrank();

    vm.roll(block.number + 6);

    vm.startPrank(victim);
    IWrappedMetaVault(wrapper).redeem(wrapperShares/10, victim, victim);
    vm.stopPrank();

    vm.roll(block.number + 6);

    _dealAndApprove(victim, metavault, assets, depositAmounts);
    vm.startPrank(victim);
    IStabilityVault(metavault).depositAssets(assets, depositAmounts, 0, victim);
    vm.stopPrank();
    deal(assets[0], victim, depositAmounts[0]);
    vm.startPrank(victim);
    IERC20(assets[0]).approve(wrapper, 0);
    IERC20(assets[0]).approve(wrapper, depositAmounts[0]);
    IWrappedMetaVault(wrapper).deposit(depositAmounts[0], victim);
    wrapperShares = IERC20(wrapper).balanceOf(victim);
    vm.stopPrank();

    uint griefAmount = 1;
    _dealAndApprove(attacker, metavault, assets, depositAmounts);
    deal(assets[0], attacker, depositAmounts[0]);
    vm.startPrank(attacker);
    IERC20(assets[0]).approve(wrapper, 0);
    IERC20(assets[0]).approve(wrapper, depositAmounts[0]);
    vm.stopPrank();

    for (uint i = 0; i < 5; i++) {
        vm.roll(block.number + 6);
        vm.startPrank(attacker);
        IWrappedMetaVault(wrapper).deposit(griefAmount, attacker);
        vm.stopPrank();

        // --- Victim tries to redeem, but should revert due to global lock ---
        vm.startPrank(victim);
        vm.expectRevert();
        IWrappedMetaVault(wrapper).redeem(wrapperShares/10, victim, victim);
        vm.stopPrank();
    }
}
```

```
audit/stability-contracts [ forge test --match-path test/core/MetaVault.Sonic.t.sol --match-test test_DoS_GlobalWithdrawalLock_PoC -vvv
[⠊] Compiling...
[⠒] Compiling 1 files with Solc 0.8.28
[⠢] Solc 0.8.28 finished in 810.65ms
Compiler run successful!

Ran 1 test for test/core/MetaVault.Sonic.t.sol:MetaVaultSonicTest
[PASS] test_DoS_GlobalWithdrawalLock_PoC() (gas: 20213169)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 453.06ms (162.27ms CPU time)

Ran 1 test suite in 462.30ms (453.06ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Track last interaction per user in the wrapper.

3.2.7 Mint for RevenueRouter doesnt check max supply limit

Submitted by BengalCatBalu, also found by IAM0TI

Severity: Medium Risk

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

Summary: Lets consider `HardWorkMintFeeCallback`:

```
function hardWorkMintFeeCallback(address[] memory revenueAssets, uint[] memory revenueAmounts) external
↪  virtual {
    IPlatform _platform = IPlatform(platform());
    uint feeShares =
        VaultBaseLib.hardWorkMintFeeCallback(_platform, revenueAssets, revenueAmounts, _getVaultBaseStorage());
    if (feeShares != 0) {
        address revenueRouter = _platform.revenueRouter();
        _approve(address(this), revenueRouter, feeShares);
        _mint(address(this), feeShares);
        IRevenueRouter(revenueRouter).processFeeVault(address(this), feeShares);
    }
}
```

This feature mints a portion of vault shares in favour of the revenue router as position growth commissions. However, vault shares have a clear limit on the maximum supplu - maxSupply, which is checked at the time of minting during deposits.

```
function _mintShares(
    VaultBaseStorage storage $,
    uint totalSupply_,
    uint value_,
    uint totalValue_,
    uint[] memory amountsConsumed,
    uint minSharesOut,
    address[] memory assets_,
    address receiver
) internal returns (uint mintAmount) {
    uint initialShares;
    (mintAmount, initialShares) = _calcMintShares(totalSupply_, value_, totalValue_, amountsConsumed, assets_);
    uint _maxSupply = $.maxSupply;
    // nosemgrep
    if (_maxSupply != 0 && mintAmount + totalSupply_ > _maxSupply) {
        revert ExceedMaxSupply(_maxSupply);
    }
    if (mintAmount < minSharesOut) {
        revert ExceedSlippage(mintAmount, minSharesOut);
    }
    if (initialShares > 0) {
        _mint(ConstantsLib.DEAD_ADDRESS, initialShares);
    }
    if (receiver == address(0)) {
        receiver = msg.sender;
    }
    _mint(receiver, mintAmount);
}
```

However, as we can see, in `hardWorkMintFeeCallback` the chasing is done via a direct call to `_mint` without checking if maxSupply is exceeded. Thus, it is obvious that max supply will be exceeded sooner or later due to such calls.

Impact Explanation: `MaxSupply` invariant protocol is sooner or later violated.

Likelihood Explanation: The `hardWorkMintFeeCallback` is called once an hour during a deposit, or at any time by a trusted person. That is, this mint happens often enough - so shares will increase unlimitedly.

Recommendation: Add max supply check in `hardWorkMintFeeCallback`.

3.2.8 Incorrect Exchange Asset Used in `StrategyBase::doHardWork`

Submitted by 0xDeoGratias, also found by Aamirusmani1552

Severity: Medium Risk

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

Description: In the `StrategyBase:doHardWork()` function, when `autoCompoundingByUnderlyingProtocol()` returns true, the protocol calls `_liquidateRewards` with the first asset from the assets() array (`assets[0]`) as the exchange asset, rather than the asset at `_exchangeAssetIndex`. This means that in strategies with multiple assets, the wrong asset may be used for reward liquidation, leading to missed or failed swaps, inefficient compounding, or unexpected reverts.

```
// maybe this is not final logic
// vault shares as fees can be used not only for autoCompoundingByUnderlyingProtocol strategies,
// but for many strategies linked to CVault if this feature will be implemented

if (StrategyLib.isPositiveAmountInArray(__amounts)) {
    IVault(_vault).hardWorkMintFeeCallback(__assets, __amounts);
} else {
    (, uint[] memory __assetsAmounts) = assetsAmounts();
    uint[] memory virtualRevenueAmounts = new uint[](__assets.length);
    virtualRevenueAmounts[0] = __assetsAmounts[0] * (block.timestamp - $.lastHardWork) / 365 days / 30;
    IVault(_vault).hardWorkMintFeeCallback(__assets, virtualRevenueAmounts);
}
// call empty method only for coverage or them can be overriden
_liquidateRewards(__assets[0], __rewardAssets, __rewardAmounts); // <@ Always called with assets[0]
_processRevenue(__assets, __amounts);
_compound();
```

Impact: If the strategy overrides `_liquidateRewards()` and relies on the exchange asset argument to build the swap path, passing the wrong asset can lead to a revert or to swapping into the wrong token.

Because the return value of _liquidateRewards() is purposely ignored in this branch, there is no direct loss of principal, but harvest / hard-work transactions can fail or rewards can be left un-swapped and never reach the vault, reducing APY.

Not a direct loss of principal, but users may lose accrued rewards and the protocol's compounding logic can be silently broken in affected strategies.

Proof of Concept: This proof of concept shows that the liquidateFunction is always called with `assets[0]`, even when the exchange asset differs. To run this proof of concept, create a file called `LiquidateRewardWrongArgs.t.sol` place the file under `test/core`. Run the proof of concept using `forge test --mt test_liquidateRewardsGetsWrongExchangeAsset -vv`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import {Test, console}   from "forge-std/Test.sol";
import {FullMockSetup}   from "../base/FullMockSetup.sol";
import {Proxy}           from "../../src/core/proxy/Proxy.sol";
import {CVault}          from "../../src/core/vaults/CVault.sol";
import {MockStrategy}    from "../../src/test/MockStrategy.sol";
import {IVault}          from "../../src/interfaces/IVault.sol";
import {VaultStatusLib}  from "../../src/core/libs/VaultStatusLib.sol";
import {StrategyBase}    from "../../src/strategies/base/StrategyBase.sol";
import {LPStrategyBase} from "../../src/strategies/base/LPStrategyBase.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "../../src/test/interfaces/IMockERC20.sol";
import "../../src/strategies/libs/StrategyIdLib.sol";
import "../../src/core/libs/CommonLib.sol";

/// ------------------------------------------------------------------------
///                           CaptureLiquidateStrategy
///
///  We inherit the on-chain MockStrategy (which itself extends StrategyBase)
///  and override just two pieces:
///    1) `autoCompoundingByUnderlyingProtocol()`  `true` (to hit the buggy branch)
///    2) `_liquidateRewards(...)`  record the args for our assertions
/// ------------------------------------------------------------------------
contract CaptureLiquidateStrategy is LPStrategyBase {
    string public constant VERSION = "10.99.99";

    uint private _depositedToken0;
    uint private _depositedToken1;
    uint private _fee0;
    uint private _fee1;

    bool private _depositReturnZero;

    function description() external pure returns (string memory) {
        return "";
    }

    function setLastApr(uint apr) external {
        StrategyBaseStorage storage $ = _getStrategyBaseStorage();
```

```
        $.lastApr = apr;
    }

    function initialize(
        address[] memory addresses,
        uint[] memory, /*nums*/
        int24[] memory /*ticks*/
    ) public initializer {
        require(addresses[3] != address(0), "Strategy: underlying token cant be zero for this strategy");

        __LPStrategyBase_init(
            LPStrategyBaseInitParams({
                id: StrategyIdLib.DEV,
                platform: addresses[0],
                vault: addresses[1],
                pool: addresses[2],
                underlying: addresses[3]
            })
        );
    }

    function toggleDepositReturnZero() external {
        _depositReturnZero = !_depositReturnZero;
    }

    function isHardWorkOnDepositAllowed() external pure returns (bool) {
        return true;
    }

    function isReadyForHardWork() external pure returns (bool isReady) {
        isReady = true;
    }

    function initVariants(address)
        public
        pure
        returns (string[] memory variants, address[] memory addresses, uint[] memory nums, int24[] memory ticks)
    {
        variants = new string[](2);
        variants[0] = "Collect fees in mock pool A";
        variants[1] = "Collect fees in mock pool B";
        addresses = new address[](4);
        addresses[0] = address(1);
        addresses[1] = address(2);
        addresses[2] = address(3);
        addresses[3] = address(4);
        nums = new uint[](0);
        ticks = new int24[](0);
    }

    function getSpecificName() external pure override returns (string memory, bool) {
        return ("Good Params", true);
    }

    function extra() external pure returns (bytes32) {
        bytes3 color = 0x558ac5;
        bytes3 bgColor = 0x121319;
        return CommonLib.bytesToBytes32(abi.encodePacked(color, bgColor));
    }

    function ammAdapterId() public pure override returns (string memory) {
        return "MOCKSWAP";
    }

    function strategyLogicId() public pure override returns (string memory) {
        return StrategyIdLib.DEV;
    }

    function triggerFuse() external {
        StrategyBaseStorage storage $ = _getStrategyBaseStorage();
        $.total = 0;
    }

    /*    function untriggerFuse(uint total_) external {
        total = total_;
    }*/
```

```
function setFees(uint fee0_, uint fee1_) external {
    _fee0 = fee0_;
    _fee1 = fee1_;
}

function getAssetsProportions() external view returns (uint[] memory proportions) {
    proportions = new uint[](2);
    proportions[0] = _getProportion0(pool());
    proportions[1] = 1e18 - proportions[0];
}

function _assetsAmounts() internal view override returns (address[] memory assets_, uint[] memory
↪  amounts_) {
    StrategyBaseStorage storage $ = _getStrategyBaseStorage();
    assets_ = $._assets;
    amounts_ = new uint[](2);
}

function _getProportion0(address /*pool*/ ) internal pure returns (uint) {
    return 5e17;
}

function _depositAssets(uint[] memory amounts, bool /*claimRevenue*/ ) internal override returns (uint
↪  value) {
    // no msg.sender checks
}

function depositUnderlying(uint amount) external override virtual returns (uint[] memory amountsConsumed) {
    // no msg.sender checks
    // require(_depositedToken0 > 0, "Mock: deposit assets first");
     _fakeTotal += amount;
    amountsConsumed = new uint[](1);
    amountsConsumed[0] = amount;

}

function _withdrawAssets(uint value, address receiver) internal override returns (uint[] memory
↪  amountsOut) {
    // no msg.sender checks
}

// function total() public view override returns (uint) {
//     StrategyBaseStorage storage $ = _getStrategyBaseStorage();
//     return $.total;
// }

function withdrawUnderlying(uint amount, address receiver) external override {
    // no msg.sender checks
}

function _claimRevenue()
    internal
    virtual
    override
    returns (
        address[] memory __assets,
        uint[] memory __amounts,
        address[] memory __rewardAssets,
        uint[] memory __rewardAmounts
    )
{
    StrategyBaseStorage storage $ = _getStrategyBaseStorage();
    IMockERC20($._assets[0]).mint(_fee0);
    IMockERC20($._assets[1]).mint(_fee1);
    __amounts = new uint[](2);
    __amounts[0] = _fee0;
    __amounts[1] = _fee1;
    _fee0 = 0;
    _fee1 = 0;
    __assets = $._assets;

    __rewardAssets = new address[](0);
    __rewardAmounts = new uint[](0);
}
```

```
    function _compound() internal virtual override {}

    function getRevenue() external pure override returns (address[] memory __assets, uint[] memory amounts) {
        __assets = new address[](0);
        amounts = new uint[](0);
    }

    function assets() public view override returns (address[] memory) {
        address[] memory one = new address[](1);
        one[0] = underlying();
        return one;
    }

    function total() public view override returns (uint) {
        return _fakeTotal;
    }

    function previewDepositAssets(
        address[] memory assets_,
        uint[] memory amountsMax
    ) public view override returns (uint[] memory consumed, uint value) {
        consumed = new uint[](2);
        consumed[0] = amountsMax[0];
        consumed[1] = 0;
        value = consumed[0];

    }

    bool     public liquidated;
    address  public exchangeAssetArg;
    address[] public rewardAssetsArg;
    uint[]   public rewardAmountsArg;
    uint private _fakeTotal;


    /// force the "else" branch in StrategyBase.doHardWork()
    function autoCompoundingByUnderlyingProtocol() public pure override returns (bool) {
        return true;
    }

    /// record whatever gets passed in here
    function _liquidateRewards(
        address          _exchangeAsset,
        address[] memory _rewardAssets,
        uint[] memory    _rewardAmounts
    ) internal override returns (uint) {
        liquidated       = true;
        exchangeAssetArg = _exchangeAsset;
        rewardAssetsArg  = _rewardAssets;
        rewardAmountsArg = _rewardAmounts;
        return 0;
    }

    function getExchangeAssetIndex() external view returns (uint) {
        return 1;
    }
}

/// ------------------------------------------------------------------------
///                             The POC Test
/// ------------------------------------------------------------------------
contract LiquidateRewardsWrongArgsTest is Test, FullMockSetup {
    CVault                   vault;
    CaptureLiquidateStrategy strategy;

    function setUp() public {
        // 1) deploy vault proxy
        Proxy vaultProxy = new Proxy();
        vaultProxy.initProxy(address(vaultImplementation));
        vault = CVault(payable(address(vaultProxy)));

        // 2) deploy our capturing strategy proxy
        Proxy stratProxy = new Proxy();
        CaptureLiquidateStrategy strategyImplementation = new CaptureLiquidateStrategy();
        stratProxy.initProxy(address(strategyImplementation));
        strategy = CaptureLiquidateStrategy(payable(address(stratProxy)));
```

```
        // 3) initialize the strategy (4 addresses per the MockStrategy convention)
        address[] memory stratAddrs = new address[](4);
        stratAddrs[0] = address(platform);
        stratAddrs[1] = address(vaultProxy);
        stratAddrs[2] = address(lp);
        stratAddrs[3] = address(tokenA);

        strategy.initialize(stratAddrs, new uint[](0), new int24[](0));

        // 4) initialize the vault
        vault.initialize(
            IVault.VaultInitializationData({
                platform:           address(platform),
                strategy:           address(strategy),
                name:               "POC Vault",
                symbol:             "POCV",
                tokenId:            0,
                vaultInitAddresses: new address[](0),
                vaultInitNums:      new uint[](0)
            })
        );

        // 5) activate the vault
        {
            address[] memory v = new address[](1);
            uint[] memory s = new uint[](1);
            v[0] = address(vault);
            s[0] = VaultStatusLib.ACTIVE;
            factory.setVaultStatus(v, s);
        }

        uint256 deposit = 1e18;
        tokenA.mint(deposit);
        tokenA.approve(address(vault), deposit);

        vm.prank(address(lp));
        tokenA.mint(deposit);
        tokenB.mint(deposit);
        tokenB.approve(address(vault), deposit);


        address[] memory assts = new address[](1);
        uint[] memory amts = new uint[](1);
        assts[0] = address(tokenA);
        amts[0]  = deposit;
        vault.depositAssets(assts, amts, 0, address(this));
    }

    function test_liquidateRewardsGetsWrongExchangeAsset() public {
        assertFalse(strategy.liquidated(), "should not have liquidated yet");

        vm.prank(platform.hardWorker());
        vault.doHardWork();

        // now our override must have been invoked
        assertTrue(strategy.liquidated(), "liquidateRewards was never called");

        // fetch the strategy's declared assets and exchange index
        address[] memory stratAssets = strategy.assets();
        uint idx = strategy.getExchangeAssetIndex(); //returns 1 (hard coded)

        console.log("EXCHANGE ASSET INDEX", idx);

        //Note how liquidateRewards got called with assets[0], not the designated exchangeAssetIndex
        assertEq(
            strategy.exchangeAssetArg(),
            stratAssets[0],
            "got called with assets[0], not the designated exchangeAssetIndex"
        );
    }
}
```

Recommendation: Instead of always passing `assets[0]`, pass `assets[_exchangeAssetIndex]`, ensuring that the intended asset is used for reward liquidation in multi-asset strategies.

3.2.9 The whole vault could lose money in `rebalance` due to lack of slippage protection

Submitted by rscodes, also found by FEDORA, rscodes, bl4ck4non, YanecaB, BengalCatBalu, CodexBugmeNot, RektOracle and deeney

Severity: Medium Risk

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

Summary: In `MetaVault.sol`:

```
function rebalance(
    uint[] memory withdrawShares,
    uint[] memory depositAmountsProportions
) external onlyAllowedOperator returns (uint[] memory proportions, int cost) {
    _checkProportions(depositAmountsProportions);

    MetaVaultStorage storage $ = _getMetaVaultStorage();
    uint len = $.vaults.length;
    require(
        len == withdrawShares.length && len == depositAmountsProportions.length,
        IControllable.IncorrectArrayLength()
    );

    (uint tvlBefore,) = tvl();

    if (CommonLib.eq($._type, VaultTypeLib.MULTIVAULT)) {
        address[] memory _assets = $.assets.values();
        for (uint i; i < len; ++i) {
            if (withdrawShares[i] != 0) {
                IStabilityVault($.vaults[i]).withdrawAssets(_assets, withdrawShares[i], new uint[](1)); // <<<
                require(depositAmountsProportions[i] == 0, IncorrectRebalanceArgs());
            }
        }
        uint totalToDeposit = IERC20(_assets[0]).balanceOf(address(this));
        for (uint i; i < len; ++i) {
            address vault = $.vaults[i];
            uint[] memory amountsMax = new uint[](1);
            amountsMax[0] = depositAmountsProportions[i] * totalToDeposit / 1e18;
            if (amountsMax[0] != 0) {
                IERC20(_assets[0]).forceApprove(vault, amountsMax[0]);
                IStabilityVault(vault).depositAssets(_assets, amountsMax, 0, address(this));
                require(withdrawShares[i] == 0, IncorrectRebalanceArgs());
            }
        }
        // ...
    }
    // ...
}
```

As pointed out by the line with an arrow, the slippage paramter in `withdrawAssets` is passed in as 0. And later on the `IERC20(_assets[0]).balanceOf(address(this))` is used as the total number to be distributed according to the proportionate ratio. Hence, if less than optimal assets are extracted by `withdrawAssets` then the whole vault ends up losing money.

Impact: Lack of slippage protection causes potential loss as the `withdrawShares[i]` may end up withdrawing less than optimal assets.

Recommendation: Add a simple slippage check.

3.2.10 `MetaVault.depositAssets()` can revert if selected vault exceeds `maxSupply`

Submitted by newspacexyz, also found by YanecaB

Severity: Medium Risk

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

Summary: The `MetaVault` chooses a single vault for depositing user assets based on target proportion logic. However, it does not account for the selected vault's `maxSupply` constraint. If the calculated `mintAmount` of shares for that vault exceeds its `maxSupply`, the deposit reverts, causing a denial of service to users even if other vaults are capable of accepting the deposit.

Finding Description: In the `MetaVault`, `depositAssets()` routes all user deposits to a single vault selected by `vaultForDeposit()`. However, `vaultForDeposit()` does not account for whether that vault can accept more deposits without violating `maxSupply`. This leads to a scenario where:

- A vault is selected.
- Deposit is attempted.
- But the vault enforces this check:

```
function _mintShares(
    VaultBaseStorage storage $,
    uint totalSupply_,
    uint value_,
    uint totalValue_,
    uint[] memory amountsConsumed,
    uint minSharesOut,
    address[] memory assets_,
    address receiver
) internal returns (uint mintAmount) {
    uint initialShares;
    (mintAmount, initialShares) = _calcMintShares(totalSupply_, value_, totalValue_, amountsConsumed,
    ↪  assets_);
    uint _maxSupply = $.maxSupply;
    // nosemgrep
    if (_maxSupply != 0 && mintAmount + totalSupply_ > _maxSupply) { // <<<
        revert ExceedMaxSupply(_maxSupply);
    }
    if (mintAmount < minSharesOut) {
        revert ExceedSlippage(mintAmount, minSharesOut);
    }
    if (initialShares > 0) {
        _mint(ConstantsLib.DEAD_ADDRESS, initialShares);
    }
    if (receiver == address(0)) {
        receiver = msg.sender;
    }
    _mint(receiver, mintAmount);
}
```

This causes the entire `MetaVault` deposit transaction to revert. Even if whale depositor deposits large amount, but every vaults doesn't have enough capacity, but their total capacity is sufficient. Due to this, protocol loses great opportunity.

Impact Explanation: Denial of Service: The system enters a state where users can't deposit funds into the `MetaVault`—even though other underlying vaults could have accepted deposits.

Likelihood Explanation: This will happen when the total supply of the selected deposit vault reaches at `maxSupply`.

Proof of Concept: The proof of concept is written in `test/Core/MetaVault.Sonic.t.sol`.

```
  function test_universal_metavault() public {
+     vm.prank(multisig);
+     IVault(SonicConstantsLib.VAULT_C_USDC_SiF).setMaxSupply(500e18);
      // ...
  }
```

```
[FAIL: ExceedMaxSupply(500000000000000000000 [5e20])] test_universal_metavault() (gas: 1511245)
Suite result: FAILED. 2 passed; 1 failed; 0 skipped; finished in 1.25s (46.38ms CPU time)

Ran 1 test suite in 1.27s (1.25s CPU time): 2 tests passed, 1 failed, 0 skipped (3 total tests)

Failing tests:
Encountered 1 failing test in test/core/MetaVault.Sonic.t.sol:MetaVaultSonicTest
[FAIL: ExceedMaxSupply(500000000000000000000 [5e20])] test_universal_metavault() (gas: 1511245)
```

Recommendation: Check vault capacity before selecting for deposit in `vaultForDeposit()`. Update the mechanism that if available capacity is greater than `maxSupply`, deposit available amount for this vault and remaining amount for next available vault.