shieldify

Multipli

Vault

SECURITY REVIEW

Date: 8 July 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Multipli Vault

Multipli is a Real World Asset (RWA) yield protocol that employs delta neutral strategies to generate consistent yield. Delta neutral strategies maintain a balanced position where the portfolio's value remains relatively stable regardless of market price movements, allowing the protocol to capture yield from various sources while minimizing directional risk. This repository contains the code for making the protocol ERC-4626 compatible, providing standardized vault interfaces that integrate seamlessly with the broader DeFi ecosystem. The protocol is deployed using the UUPS proxy pattern, which enables secure upgradability while maintaining a single contract address.

Learn more: Docs

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days, dedicated by the Shieldify core team.

The audit report identified one High, three Medium and one Low severity findings, mainly concerning exploitable yield distribution, redeeming initial deposits, potential price manipulation, griefing attacks, and no slippage protection.

The Multipli team has done a great job with the development and has been highly responsive to the Shieldify research team's inquiries.

5.1 Protocol Summary

Project Name	**Multipli Vault**
Repository	MultipliVault
Type of Project	DeFi, Vault
Audit Timeline	5 days
Review Commit Hash	c6ff40b438fe6d0ec01a6630f3ef2c9020ea4f8d
Fixes Review Commit Hash	664fd70bd9f09dd362eacd0b3bbaa5eee2013038

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/base/AuthUpgradeable.sol	58
src/base/FundMovementHelperUpgradeable.sol	57
src/base/VaultFeeUpgradeable.sol	72
src/common/Role.sol	9
src/fees/VariableVaultFee.sol	132
src/libraries/Errors.sol	21
src/managers/VaultFundManager.sol	245
src/vault/MultipliVault.sol	379
Total	**973**

6. Findings Summary

The following issues have been identified, sorted by their severity:

- **High** issues: **1**
- **Medium** issues: **3**
- **Low** issues: **1**

ID	Title	Severity	Status
[H-01]	Attackers can Exploit Yield Distribution Through `onUnderlyingBalanceUpdate()` Sandwiching	High	Fixed
[M-01]	Underflow Bug in `addFundsAndFulfillRedeem()` Prevents Redemption of Initial Deposits	Medium	Fixed
[M-02]	Price Manipulation Occur Through Order-Dependent State Updates	Medium	Fixed
[M-03]	Lack of Slippage Protection in `deposit()` and `mint()` Functions	Medium	Fixed
[L-01]	Duplicate Event Emission and Unnecessary Memory Variable in `updateUnderlyingBalance()`	Low	Acknowledged

7. Findings

[H-01] Attackers can Exploit Yield Distribution Through `onUnderlyingBalanceUpdate()` Sandwiching

Severity

High Risk

Description

The `onUnderlyingBalanceUpdate()` function in the `MultipliVault` contract is vulnerable to sandwich attacks that allow malicious actors to extract yield intended for legitimate vault participants. This vulnerability stems from the predictable nature of yield distribution updates and the lack of protection against front-running attacks.

The attack vector exploits the following sequence: when operators call `onUnderlyingBalanceUpdate()` to report yield generated from external strategies, the transaction becomes visible in the mempool before execution. Attackers can observe this pending transaction and construct a sandwich attack by frontrunning with a large deposit transaction, capturing a disproportionate share of the yield update, and then immediately withdrawing their position.

The vulnerability is particularly severe because the `onUnderlyingBalanceUpdate()` function directly increases the `aggregatedUnderlyingBalances` storage variable, which affects the `totalAssets()` calculation. Since ERC4626 vault shares are priced based on the ratio of total assets to total supply, any increase in total assets immediately benefits all current shareholders

proportionally. An attacker who deposits just before this update captures yield they did not earn, while legitimate long-term depositors receive diluted returns.

The attack becomes economically viable when the expected yield from the update exceeds the transaction costs and any applicable fees. Given that the vault supports different fee structures for normal versus instant redemptions, attackers can optimize their strategy by choosing the most cost-effective withdrawal method, potentially using instant redemptions if the fees are insufficient to deter such behavior.

Location of Affected Code

File: src/vault/MultipliVault.sol#L490

```
function onUnderlyingBalanceUpdate (uint256 newAggregatedBalance)
    external requiresAuth {
     MultipliVaultStorage storage $ = _getMultipliVaultStorage();

     // Fail fast - ensure this is the first update in this block
     require(block.number > $.lastBlockUpdated, Errors.
        UpdateAlreadyCompletedInThisBlock());

     emit UnderlyingBalanceUpdated($.aggregatedUnderlyingBalances,
        newAggregatedBalance);
     $.aggregatedUnderlyingBalances = newAggregatedBalance;

     // Calculate new price per share and check for excessive volatility
     // todo: should this be replaced with `convertToAssets(1e6)`?
     uint256 newPricePerShare = totalAssets().mulDiv(DENOMINATOR,
        totalSupply());

     uint256 percentageChange = _calculatePercentageChange($.
        lastPricePerShare, newPricePerShare);

     // Pause the vault if the percentage change exceeds the threshold (
        works in both directions)
     if (percentageChange > $.maxPercentageChange) {
         _pause();
     }

     $.lastPricePerShare = newPricePerShare;
     $.lastBlockUpdated = block.number;
}
```

File: src/managers/VaultFundManager.sol#L305

```
function updateUnderlyingBalance (uint256 newAggregatedBalance) external
    nonReentrant onlyVault {
     uint256 oldBalance = vault.aggregatedUnderlyingBalances();

     vault.onUnderlyingBalanceUpdate(newAggregatedBalance);

     emit UnderlyingBalanceUpdated(oldBalance, newAggregatedBalance);
}
```

Impact

Legitimate vault participants suffer direct financial losses as their proportional share of generated yield is diluted by attackers who contribute no value to the vault's investment strategy. The attack creates an unfair wealth transfer from honest users to sophisticated MEV operators who can monitor mempool activity and execute complex transaction sequences.

Recommendation

The protocol should implement deposit and withdrawal delays that prevent immediate exploitation of yield updates. By introducing a minimum holding period for new deposits before they become eligible for yield distributions, the protocol can ensure that only committed participants benefit from investment returns.

Team Response

Fixed.

[M-01] Underflow Bug in `addFundsAndFulfillRedeem()` Prevents Redemption of Intial Deposits

Severity

Medium Risk

Description

The `addFundsAndFulfillRedeem()` function in `VaultFundManager.sol` contains a critical underflow vulnerability that prevents users from redeeming their first deposits/ initialDeposit.

```
uint256 oldAggregatedUnderlyingBalances = vault.
    aggregatedUnderlyingBalances();

IERC20(asset).safeTransfer(address(vault), assetsWithFee);

// Step 2: Update the aggregated balance to reflect assets moved from
    external strategies
uint256 newAggregatedBalance = oldAggregatedUnderlyingBalances -
    assetsWithFee; // BUG: Underflow here
vault.onUnderlyingBalanceUpdate(newAggregatedBalance);
```

- When a user makes their first deposit, **aggregatedUnderlyingBalances** starts at 0 (default uint256 value)
- The function assumes assets are being moved from external strategies and subtracts **assetsWithFee** from **aggregatedUnderlyingBalances**
- This causes an underflow when **oldAggregatedUnderlyingBalances = 0** and **assetsWithFee > 0**
- The transaction reverts due to the **uint256** underflow protection
- The user's funds are stuck in the contract

Location of Affected Code

File: src/managers/VaultFundManager.sol#L345

```
function addFundsAndFulfillRedeem (address receiver, uint256 shares,
    uint256 assetsWithFee) external nonReentrant onlyVault {
  // code
  uint256 oldAggregatedUnderlyingBalances = vault.
     aggregatedUnderlyingBalances();

  IERC20(asset).safeTransfer(address(vault), assetsWithFee);

  // Step 2: Update the aggregated balance to reflect assets moved from
     external strategies
  uint256 newAggregatedBalance = oldAggregatedUnderlyingBalances -
     assetsWithFee;
  vault.onUnderlyingBalanceUpdate(newAggregatedBalance);
  // code
}
```

Impact

- All initial redemption transactions will revert until `oldAggregatedUnderlyingBalances > assetsWithFee`
- Also, in the future, it can not check about the `oldAggregatedUnderlyingBalances > assetsWithFee`. Therefore, whenever the redemption is greater than the `oldAggregatedUnderlyingBalances`, this will be reverted even if the contract has sufficient funds.

Recommendation

Add a check that `oldAggregatedUnderlyingBalances > assetsWithFee`.

Team Response

Fixed.

[M-02] Price Manipulation Occurs Through Order-Dependent State Updates

Severity

Medium Risk

Description

The `addFundsAndFulfillRedeem()` function in the `VaultFundManager` contract contains a critical ordering issue that can lead to inconsistent `lastPricePerShare` calculations. The vulnerability arises from the sequence of operations where the aggregated underlying balance is updated before the redemption is fulfilled, creating a temporary state where the vault's total assets and share calculations become misaligned.

In the current implementation, Step 2 calls `vault.onUnderlyingBalanceUpdate(newAggregatedBalance)`, which decreases the aggregated underlying balances by `assetsWithFee`, effectively reducing the total assets of the vault. However, Step 3 then calls `vault.fulfillRedeem(receiver, shares, assetsWithFee)` which burns the shares and transfers the assets. This creates an intermediate state where the total assets have been reduced but the shares have not yet been burned, leading to an artificially deflated price per share calculation during the brief window between these operations.

The same vulnerability exists in the vault's `fulfillRedeem()` and `fulfillInstantRedeem()` functions, where similar state update ordering issues can occur.

Location of Affected Code

File: src/managers/VaultFundManager.sol#L345

```
function addFundsAndFulfillRedeem (address receiver, uint256 shares,
    uint256 assetsWithFee) external nonReentrant onlyVault {
     if (receiver == address(0)) {
         revert ZeroAddress();
     }
     if (shares == 0 || assetsWithFee == 0) {
         revert ZeroAmount();
     }

     uint256 contractBalance = IERC20(asset).balanceOf(address(this));
     if (assetsWithFee > contractBalance) {
         revert InsufficientBalance();
     }

     uint256 oldAggregatedUnderlyingBalances = vault.
        aggregatedUnderlyingBalances();

     // Step 1: Transfer the required assets from this contract to the
        vault
     IERC20(asset).safeTransfer(address(vault), assetsWithFee);

     // Step 2: Update the aggregated balance to reflect assets moved from
         external strategies
     uint256 newAggregatedBalance = oldAggregatedUnderlyingBalances -
        assetsWithFee;
     vault.onUnderlyingBalanceUpdate(newAggregatedBalance);

     // Step 3: Fulfill the redemption request
     vault.fulfillRedeem(receiver, shares, assetsWithFee);

     emit FundsAddedAndRedemptionFulfilled(receiver, shares, assetsWithFee
        , newAggregatedBalance);
}
```

Impact

During the window where `lastPricePerShare` is artificially low, subsequent operations that rely on this value for calculations could result in unfavourable exchange rates for legitimate users.

Recommendation

The recommended fix involves reordering the operations in the `addFundsAndFulfillRedeem()` function to ensure that share burning occurs before the underlying balance is updated. The corrected sequence should be: Step 1 remains unchanged (transfer assets to vault), Step 2 should call `vault.fulfillRedeem(receiver, shares, assetsWithFee)` to burn shares and transfer assets, and Step 3 should then call `vault.onUnderlyingBalanceUpdate(newAggregatedBalance)` to update the aggregated balance.

Team Response

Fixed.

[M-03] Lack of Slippage Protection in `deposit()` and `mint()` Functions

Severity

Medium Risk

Description

The `MultipliVault` contract implements the ERC4626 standard but lacks essential slippage protection mechanisms for direct EOA interactions in the `deposit()` and `mint()` functions. According to `EIP-4626`, implementors should consider adding slippage controls when supporting direct EOA access, as these users have no alternative means to revert transactions if the exact output amount is not achieved.

The vulnerability manifests in two critical functions. In the deposit function, users specify an asset amount and receive shares based on the current exchange rate calculated through `previewDeposit()`. However, this preview calculation includes fee deductions via `_feeOnTotalDeposit`, and the actual shares received can differ significantly from user expectations if the vault's underlying asset values change between transaction submission and execution. The function lacks a minimum shares output parameter that would allow users to specify their slippage tolerance.

Similarly, the mint function allows users to specify a desired number of shares but calculates the required asset amount through `previewMint()`, which includes additional fees via `_feeOnRawDeposit()`. Users cannot specify a maximum asset amount they are willing to spend, leaving them vulnerable to paying more than expected if market conditions change unfavourably.

Location of Affected Code

File: src/vault/MultipliVault.sol#L639

```
function deposit (uint256 assets, address receiver) public override
   whenNotPaused nonReentrant returns (uint256) {
    uint256 maxAssets = maxDeposit(receiver);
    if (assets > maxAssets) {
        revert ERC4626ExceededMaxDeposit(receiver, assets, maxAssets);
    }

    MultipliVaultStorage storage $ = _getMultipliVaultStorage();
    uint256 currentMinDepositAmount = $.minDepositAmount;

    if (assets < currentMinDepositAmount) {
        revert Errors.DepositAmountLessThanThreshold(assets,
           currentMinDepositAmount);
    }
    uint256 shares = previewDeposit(assets);
    _deposit(_msgSender(), receiver, assets, shares);

    return shares;
}
```

File: src/vault/MultipliVault.sol#L674

```
/**
 * @notice Override the default `mint` function with additional checks
 * @param shares The amount of shares to mint
 * @param receiver The address that will receive the shares
 * @return assets The amount of assets required
 * @dev Adds pause protection and minimum deposit amount validation
 */
function mint( uint256 shares, address receiver ) public override
   whenNotPaused nonReentrant returns (uint256) {
    uint256 maxShares = maxMint(receiver);
    if (shares > maxShares) {
        revert ERC4626ExceededMaxMint(receiver, shares, maxShares);
    }
    uint256 assets = previewMint(shares);

    MultipliVaultStorage storage $ = _getMultipliVaultStorage();
    uint256 currentMinDepositAmount = $.minDepositAmount;

    if (assets < currentMinDepositAmount) {
        revert Errors.DepositAmountLessThanThreshold(assets,
           currentMinDepositAmount);
    }

    _deposit(_msgSender(), receiver, assets, shares);

    return assets;
}
```

Impact

Direct EOA users face significant financial risks when interacting with the vault. They may receive substantially fewer shares than anticipated when depositing assets, particularly during periods of high volatility or when the vault's underlying assets experience losses.

Recommendation

Implement comprehensive slippage protection by adding overloaded versions of the `deposit()` and `mint()` functions that include slippage parameters. For the deposit function, add a `minSharesOut` parameter that specifies the minimum number of shares the user is willing to accept. The function should revert if the amount of the calculated shares falls below this threshold. For the `mint()` function, add a `maxAssetsIn` parameter that specifies the maximum number of assets the user is willing to spend. The function should revert if the calculated asset requirement exceeds this limit.

Team Response

Fixed.

[L-01] Duplicate Event Emission and Unnecessary Memory Variable in `updateUnderlyingBalance()`

Severity

Low Risk

Description

- The `updateUnderlyingBalance()` function in `VaultFundManager` contains an unnecessary memory variable and emits a duplicate event.
- The `onUnderlyingBalanceUpdate()` function in `MultipliVault` already emits the `UnderlyingBalanceUpdated` event.
- Then `updateUnderlyingBalance()` emits the same event again.
- Also, the `oldBalance` variable is only used for the duplicate event emission and serves no other purpose.

Location of Affected Code

File: src/managers/VaultFundManager.sol#L305

```
function updateUnderlyingBalance(uint256 newAggregatedBalance) external
    nonReentrant onlyVault {
     uint256 oldBalance = vault.aggregatedUnderlyingBalances();
     vault.onUnderlyingBalanceUpdate(newAggregatedBalance); // just
        confirm that this also needs checksum regarding the

     emit UnderlyingBalanceUpdated(oldBalance, newAggregatedBalance);
}
```

File: src/vault/MultipliVault.sol#L490

```
 function onUnderlyingBalanceUpdate(uint256 newAggregatedBalance)
    external requiresAuth {
    MultipliVaultStorage storage $ = _getMultipliVaultStorage();
    require(block.number > $.lastBlockUpdated, Errors.
       UpdateAlreadyCompletedInThisBlock());

    emit UnderlyingBalanceUpdated($.aggregatedUnderlyingBalances,
       newAggregatedBalance);
    $.aggregatedUnderlyingBalances = newAggregatedBalance;
}
```

Impact

- Extra SLOAD: Reading vault.aggregatedUnderlyingBalances() unnecessarily
- Extra MSTORE: Storing value in memory variable
- Duplicate Event: Emitting the same event twice
- Extra LOG: Additional log operation

Recommendation

Consider removing the duplicate event emission and the unnecessary variable:

```
function updateUnderlyingBalance(uint256 newAggregatedBalance) external
   nonReentrant onlyVault {
-    uint256 oldBalance = vault.aggregatedUnderlyingBalances();
      vault.onUnderlyingBalanceUpdate(newAggregatedBalance);
-    emit UnderlyingBalanceUpdated(oldBalance, newAggregatedBalance);
 }
```

Team Response

Acknowledged.

shieldify

Thank you!