shieldify

Colb Finance

Vault

SECURITY REVIEW

Date: 4 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 have 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 Colb Finance - Vault

Colb is the first native non-custodial tokenization solution that enables peerless access to Swiss-grade wealth management strategies, pre-IPO opportunities, and premium investment funds. It offers a bankruptcy remote Trust structure and native ownership of real-world assets, all on-chain. Colb reduces the entrance threshold to such investments by removing constraints such as the minimum investment amount to get exposure to them. The protocol is designed with security at its core, boasting compliance with Swiss regulations and DeFi composability. Colb envisions a future rooted in transparency where every individual has equitable access to premium RWA investments.

The single Vault contract seamlessly manages the full lifecycle of the Pre-IPO investment vault, from configuration, deposits, token minting, to withdrawals — with admin control and future upgradeability built-in.

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 with a total of 80 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one High, nine Medium and one Low severity issues, mainly related to deposit accounting, share distribution, and withdrawal logic, allowing malicious or unintended user actions to disrupt vault operations, bypass limits, or cause asset loss.

The Colb Finance team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Colb Finance**
Repository	SmartContracts
Type of Project	RWA, Pre-IPO, Vault
Audit Timeline	5 days
Review Commit Hash	809055574cb7d7bd7c711b87745c081524547eb4
Fixes Review Commit Hash	20d8c1cd6ad00d43c4b619ebabec5f74f773f816

5.2 Scope

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

File	nSLOC
contracts/vault/Vault.sol	449
contracts/vault/VaultFactory.sol	122
Total	**571**

6. Findings Summary

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

- **High** issues: **1**
- **Medium** issues: **9**
- **Low** issues: **1**

ID	Title	Severity	Status
[H-01]	Malicious Users Can Brick Vault Operations by Exploiting Incorrect Deposit Accounting	High	Fixed
[M-01]	User Max Deposit Count Is Prone To an Off by One	Medium	Fixed
[M-02]	Pausable Vault Does Not Override Necessary Pause Methods	Medium	Fixed
[M-03]	Withdrawing a Partial Deposit Amount Leads to Loss of Assets	Medium	Fixed
[M-04]	Operator Can Prematurely Distribute Shares Before Investment Period Ends	Medium	Fixed
[M-05]	Attacker Can Disrupt Batch Processing and Spam Deposits	Medium	Fixed
[M-06]	Users Can Bypass the Minimum Deposit Check via Partial Withdrawal	Medium	Fixed
[M-07]	User Can Bypass Maximum Deposit Limit by Exploiting Incorrect Deposit Counting Logic	Medium	Fixed
[M-08]	Late Depositors Can Manipulate Share Allocation to the Detriment of Early Investors	Medium	Fixed
[M-09]	Operators Can Cause Incorrect Share Distribution Due to Off-Chain Calculation Errors Affecting Depositors	Medium	Fixed
[L-01]	The `_removeWithdrawToProcess()` Function Is Left Unused	Low	Fixed

7. Findings

[H-01] Malicious Users Can Brick Vault Operations by Exploiting Incorrect Deposit Accounting

Severity

High Risk

Description

The vulnerability stems from improper state management in the vault's withdrawal processing logic. The contract tracks total deposits through the `totalDepositToProcess` variable, which is incremented in `userDeposit()` but never decremented when funds are withdrawn via `processWithdrawDeposit()`. This creates a critical accounting mismatch where the contract believes more funds are deposited than actually exist in the vault.

During normal operation, `userDeposit()` increases both the user's `totalDeposit` in their `UserStat` and the global `totalDepositToProcess`. However, when processing early withdrawals through `processWithdrawDeposit()`, only the user-specific `totalDeposit` is decreased while the global counter remains unchanged. This inconsistency allows an attacker to artificially inflate the vault's perceived deposits equal to its `maxTotalDeposit` capacity by depositing and then withdrawing funds repeatedly.

Location of Affected Code

File: contracts/vault/Vault.sol#L251

```
function userDeposit(uint256 amount) external onlyWhitelist {
    if (block.timestamp > investmentEnd) {
        revert InvestmentPeriodFinish();
    }

    address user = msg.sender;

    IERC20(uscToken).safeTransferFrom(user, address(this), amount);

    UserStat storage currentStat = usersStat[msg.sender];
    currentStat.totalDeposit += amount;

    if (currentStat.totalDeposit > maxDepositByUser) {
        revert DepositOverflowCap();
    }
    totalDepositToProcess += amount;
    if (totalDepositToProcess > maxTotalDeposit) {
        revert DepositOverflowTotalCap();
    }
// code
}
```

File: contracts/vault/Vault.sol#L530

```
function processWithdrawDeposit(
    uint256 processIndex
) external onlyOperator {
    uint256 indexWithdraw = withdrawToProcess[processIndex];
    WithdrawInfo storage withdrawInfo = withdraws[indexWithdraw];
    if (withdrawInfo.processed) {
        revert AlreadyProcessed();
    }
    if (withdrawInfo.nativeToken) {
        revert IncorrectWithdrawMethod();
    }
    withdrawInfo.processed = true;
    address user = withdrawInfo.sender;
    uint256 amount = withdrawInfo.amount;
    uint256 restToReimbourse = amount;

    UserStat storage userStat = usersStat[user];
    userStat.totalDeposit -= amount;

    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit == 0) {
        revert NothingToReimboursed();
    }
```

```
// if  user make deposit in different token, we withdraw from these
   different deposit if it's possible
    for (uint256 index = 0; index < totalDifferentDeposit; index++) {
        uint256 indexDeposit = userDepos[index];
        DepositInfo storage deposit = deposits[indexDeposit];
        if (deposit.processed == false && deposit.reimboursed == false) {
            uint256 amountDeposited = deposit.amount;
// we reimbourse in token deposited
            uint256 amountToUse = restToReimbourse > amountDeposited
                ? amountDeposited
                : restToReimbourse;
            deposit.amount = 0;
            restToReimbourse -= amountToUse;
            IERC20(deposit.token).safeTransfer(user, amountToUse);
        }

        if (restToReimbourse == 0) {
            break;
        }
    }

    factory.emitWithdraw(msg.sender, amount, indexWithdraw);
}
```

Impact

The exploitation of this vulnerability leads to severe and permanent consequences for the vault's operation. Once `totalDepositToProcess` equals `maxTotalDeposit`, all subsequent deposit attempts will fail due to the `DepositOverflowTotalCap` revert in `userDeposit()`, effectively freezing the vault in a non-functional state.

Legitimate users are completely blocked from participating, and operators would need to deploy an entirely new vault contract and migrate all assets to restore functionality, resulting in significant operational disruption.

Recommendation

Modify `processWithdrawDeposit()` to properly decrement `totalDepositToProcess` by the withdrawn amount, maintaining synchronisation between global and user-specific accounting.

Team Response

Fixed.

[M-01] User Max Deposit Count Is Prone To an Off by One

Severity

Medium Risk

Description

The `Vault` contract handles gas usage via limiting each individual user's allowed deposits, via the variable `maxDifferentDepositByUser`, which is set to 10 by default. It is used inside `userDeposit()` when checking the length of deposits, if it turns out this is a new deposit and does not update an already existing one in the loop prior.

However, the code checks if the current length of deposits is above the max, instead of the +1 length. For example, if the limit is 10 and we have 10 deposits, the if check does `10 > 10` and passes, creating an 11th deposit.

Location of Affected Code

File: contracts/vault/Vault.sol#L290

```
function userDeposit(uint256 amount) external onlyWhitelist {
// code
  if (totalDifferentDeposit > maxDifferentDepositByUser) {
      revert UserMakeTooMuchDeposit();
  }
// code
}
```

Impact

The vault's limit of maximum user deposits is off by one, increasing each individual's gas cost.

Recommendation

The if check should either be inclusive `>=` or be `totalDifferentDeposit + 1 > maxDifferentDepositByUser` to actually check the new length against the limit.

Team Response

Fixed.

[M-02] Pausable Vault Does Not Override Necessary Pause Methods

Severity

Medium Risk

Description

The `Vault` contract inherits Openzeppelin's `ERC20PausableUpgradable`, however it does not override any of the necessary pausing methods, nor does it utilize the bonus functionality. In case of contract emergencies, operations cannot be paused.

Quoted directly from the original contract: ERC20PausableUpgradeable.sol

```
/*
* IMPORTANT: This contract does not include public pause and unpause
   functions. In
* addition to inheriting this contract, you must define both functions,
   invoking the
* {Pausable-_pause} and {Pausable-_unpause} internal functions, with
   appropriate
* access control, e.g. using {AccessControl} or {Ownable}. Not doing so
   will
* make the contract pause mechanism of the contract unreachable, and thus
    unusable.
*/
```

Location of Affected Code

File: contracts/vault/Vault.sol

Impact

- Unutilized functionality
- Unnecessary contract size
- No protection against emergencies

Recommendation

Either implement the necessary pausing mechanism or use a different ERC20 contract

Team Response

Fixed.

[M-03] Withdrawing A Partial Deposit Amount Leads To Loss Of Assets

Severity

Medium Risk

Description

The `Vault` contract allows users to send 2 types of withdrawal requests: withdraw vault shares for underlying tokens or withdraw their unprocessed deposits and reclaim their initial transfer. The latter is done by first invoking `requestWithdrawDeposit()`, which creates the request with the amount, which could be used to only partially reduce our initial unprocessed deposit. Then, the operator is meant to call `processWithdrawDeposit()`, which iterates over all of the user's unprocessed deposits and checks if `requestAmount > currentDeposit` and uses the smaller amount to reduce the `requestAmount`. The code, however, then completely resets the deposit amount: `deposit.amount = 0`, no matter if not all of the deposit was required to reimburse the user.

For example: 1. A user accidentally deposits 110 USC instead of 100 2. Before their request gets processed, they invoke a deposit withdrawal of 10 USC 3. The code goes over their only request and checks if 10 > 100 4. 10 is smaller, so the code reduces the amount to reimburse to 0, transfers 10 USC to the user and resets their entire 110 deposit to 0 5. 100 tokens are stuck inside the contract.

Currently, the manual operator processing of deposits is unrestricted, so they can mint shares to back those 100 stuck tokens and rescue them, which makes the freezing of funds only temporary.

Location of Affected Code

File: contracts/vault/Vault.sol#L565

```
function processWithdrawDeposit( uint256 processIndex ) external
    onlyOperator {
// code
  uint256 amountDeposited = deposit.amount;

  uint256 amountToUse = restToReimbourse > amountDeposited
      ? amountDeposited
      : restToReimbourse;

  deposit.amount = 0; //<--------
  restToReimbourse -= amountToUse;
  IERC20(deposit.token).safeTransfer(user, amountToUse);
// code
}
```

Impact

- Incorrect internal accounting
- Temporarily frozen funds inside the contract

Recommendation

Do not fully reset the deposit amount, reduce it according to the reimbursement.

Team Response

Fixed.

[M-04] Operator Can Prematurely Distribute Shares Before Investment Period Ends

Severity

Medium Risk

Description

The contract's batch processing functionality lacks validation of the investment period end time in both `processDeposit()` and batch processing functions (`startBatchProcess()` ,

`continueBatchProcess()`). While the `userDeposit()` function correctly enforces the investment period check via `require(block.timestamp <= investmentEnd)` , the distribution phase functions omit this critical validation. This allows operators to prematurely trigger share distribution before the official investment period concludes, as evidenced by the absence of timestamp checks in the processing logic. The vulnerability stems from an incomplete implementation of the investment period constraints across all state-changing functions.

Location of Affected Code

File: contracts/vault/Vault.sol#L361

```
function processDeposit(
    uint256 processIndex,
    uint256 amountMinted,
    uint256 amountReimbursed
) external onlyOperator {
    if (inBatchProcess) {
// if we process during a batch, the caclculation will be incorrect
        revert BatchNotFinished();
    }
    _mint(address(this), amountMinted);

    uint256 indexDeposit = depositToProcess[processIndex];
    _processDeposit(indexDeposit, amountMinted, amountReimbursed);
    _removeDepositToProcess(processIndex);
}
```

File: contracts/vault/Vault.sol#L382

```
/// @notice Process many deposit by operator
/// @dev batch process base of total token distrubuted for amount
   accepted
/// @param totalAmountAccepted amount accepted for this round, overflow
   amount will be reimboursed
/// @param totalAmountMinted share of token minted for all deposits
/// @param maxProcess number of deposit to handle by batch, we put a max
   number to prevent from gas usage failure
function startBatchProcess(
    uint256 totalAmountAccepted,
    uint256 totalAmountMinted,
    uint256 maxProcess
) external onlyOperator {
    if (inBatchProcess) {
        revert BatchNotFinished();
    }

    _mint(address(this), totalAmountMinted);
// code
}
```

Another aspect of the issue is that users can maliciously or accidentally increase `depositToProcess()`

while the investment is still going. This will force the `inBatchProcess` boolean to always be true and DOS other operations.

Impact

This vulnerability enables the unfair distribution of shares before the investment period concludes, potentially favouring certain investors. Early distribution could lead to imbalanced token allocations, as later legitimate depositors may receive reduced shares.

Recommendation

The fix requires adding investment period validation in all share distribution functions. In `processDeposit()`, `startBatchProcess()`, and `continueBatchProcess()`, include a require statement verifying `block.timestamp > investmentEnd` before processing any distributions.

Team Response

Fixed.

[M-05] Attacker Can Disrupt Batch Processing and Spam Deposits

Severity

Medium Risk

Description

The `userDeposit()` function fails to enforce the `minDepositByUser` requirement defined in the contract, despite this being documented as a protocol invariant. The absence of this validation in the deposit processing logic (specifically the missing check against `minDepositByUser` in the deposit amount validation sequence) allows users to create an unlimited number of minimal-value deposits. This architectural oversight becomes particularly dangerous when combined with the batch processing mechanism, as each deposit requires individual processing regardless of its size.

Location of Affected Code

File: contracts/vault/Vault.sol#L251

```
function userDeposit(uint256 amount) external onlyWhitelist {
    if (block.timestamp > investmentEnd) {
        revert InvestmentPeriodFinish();
    }

    address user = msg.sender;

    IERC20(uscToken).safeTransferFrom(user, address(this), amount);

    UserStat storage currentStat = usersStat[msg.sender];
    currentStat.totalDeposit += amount;

    if (currentStat.totalDeposit > maxDepositByUser) {
        revert DepositOverflowCap();
    }

    totalDepositToProcess += amount;
    if (totalDepositToProcess > maxTotalDeposit) {
        revert DepositOverflowTotalCap();
    }

    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit > 0) {
        for (uint256 index = 0; index < totalDifferentDeposit; index++) {
            uint256 indexDeposit = userDepos[index];
            DepositInfo storage deposit = deposits[indexDeposit];
            if (deposit.processed == false && deposit.token == uscToken)
               {
// if the user has an active deposit with the same token, we reuse it
                deposit.amount += amount;
                deposit.updatedDate = uint32(block.timestamp);

                factory.emitDeposit(user, indexDeposit, amount, uscToken)
                   ;
// we leave the function here to don't duplicate deposit
                return;
            }
        }
    }
```

```
    if (totalDifferentDeposit > maxDifferentDepositByUser) {
        revert UserMakeTooMuchDeposit();
    }

    DepositInfo memory depositInfo = DepositInfo(
        user,
        amount,
        uscToken,
        uint32(block.timestamp),
        uint32(block.timestamp),
        false,
        false,
        0,
        0
    );

    uint256 newIndex = deposits.length;

    deposits.push(depositInfo);

    depositToProcess.push(newIndex);
    usersDeposit[user].push(newIndex);

    factory.emitDeposit(user, newIndex, amount, uscToken);
}
```

File: contracts/vault/Vault.sol#L382

```
function startBatchProcess(
    uint256 totalAmountAccepted,
    uint256 totalAmountMinted,
    uint256 maxProcess
) external onlyOperator {
    if (inBatchProcess) {
        revert BatchNotFinished();
    }

    _mint(address(this), totalAmountMinted);

    uint256 totalProcess = depositToProcess.length;
    uint256 length = maxProcess > totalProcess ? totalProcess :
       maxProcess;

// store information in the contract to continue the batch
    totalRoundMint = totalAmountMinted;
    totalRoundAccepted = totalAmountAccepted;
    totalRoundProcessed = 0;

    _batchProcess(length);
```

```
// if not all deposit was processed we stay in batch process
    inBatchProcess = depositToProcess.length > 0;
}

/// @notice Continue to process many deposits by the operator
/// @dev call it if you don't have a process for all deposits in the
   first batch method
/// @param maxProcess number of deposits to handle by batch, we put a max
    number to prevent gas usage failure
function continueBatchProcess(uint256 maxProcess) external onlyOperator {
    if (!inBatchProcess) {
        revert BatchNotStarted();
    }
    uint256 totalProcess = depositToProcess.length;
    uint256 length = maxProcess > totalProcess ? totalProcess :
       maxProcess;

    _batchProcess(length);

    // if not all deposit was processed, we stay in batch process
    inBatchProcess = depositToProcess.length > 0;
}
```

Impact

Attackers can exploit this vulnerability to create numerous dust deposits (e.g., 1 wei), effectively causing a denial-of-service condition for the batch processing system. During critical operations like `startBatchProcess()`, malicious actors could continuously inject new small deposits, keeping `inBatchProcess()` perpetually true and blocking normal protocol operations. This spam attack would force operators to process an impractical number of transactions, wasting gas and potentially preventing legitimate deposits from being processed in a timely manner.

Recommendation

The solution requires implementing strict deposit size validation in the `userDeposit()` function. Add a require statement `require(amount >= minDepositByUser, "Deposit below minimum")` immediately after the amount parameter is received.

Team Response

Fixed.

[M-06] Users Can Bypass the Minimum Deposit Check via Partial Withdrawal

Severity

Medium Risk

Description

The `userDeposit()` function is meant to validate the amount against the contract's specified minimum and maximum values, to make sure the position is correctly created. However, this mechanism can be bypassed in terms of the minimum deposit, since the contract allows users to withdraw their unprocessed deposits partially.

This creates the scenario in which: 1. Bob deposits a value X, which is perfectly `X > minDeposit` and `X < maxDeposit` 2. Bob invokes `requestWithdrawDeposit()` and passes `X - 1` as the amount 3. Bob's original deposit is reduced below the minimum

This has the same impact as being able to spam the system with dust deposits to overwhelm the processing during the minting phase.

Location of Affected Code

File: contracts/vault/Vault.sol#L457

```
function requestWithdrawDeposit(uint256 amount) external {
    if (block.timestamp > investmentEnd) {
        revert InvestmentPeriodFinish();
    }

    address user = msg.sender;
    UserStat memory userStat = usersStat[user];
    if (userStat.totalDeposit < amount) {
        revert OverflowAmountToWithdraw();
    }

    WithdrawInfo memory withdrawInfo = WithdrawInfo(
        user,
        uint32(block.timestamp),
        false,
        false,
        amount
    );

    uint256 index = withdraws.length;

    withdraws.push(withdrawInfo);

    withdrawToProcess.push(index);
    usersWithdraw[user].push(index);
    factory.emitRequestWithdraw(user, amount, index);
}
```

File: contracts/vault/Vault.sol#L530

```
function processWithdrawDeposit(
    uint256 processIndex
) external onlyOperator {
    uint256 indexWithdraw = withdrawToProcess[processIndex];
    WithdrawInfo storage withdrawInfo = withdraws[indexWithdraw];
    if (withdrawInfo.processed) {
        revert AlreadyProcessed();
    }
    if (withdrawInfo.nativeToken) {
        revert IncorrectWithdrawMethod();
    }
    withdrawInfo.processed = true;
    address user = withdrawInfo.sender;
    uint256 amount = withdrawInfo.amount;
    uint256 restToReimbourse = amount;

    UserStat storage userStat = usersStat[user];
    userStat.totalDeposit -= amount;

    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit == 0) {
        revert NothingToReimboursed();
    }

// if  user make deposit in different token, we withdraw from these
   different deposit if it's possible
    for (uint256 index = 0; index < totalDifferentDeposit; index++) {
        uint256 indexDeposit = userDepos[index];
        DepositInfo storage deposit = deposits[indexDeposit];
        if (deposit.processed == false && deposit.reimboursed == false) {
            uint256 amountDeposited = deposit.amount;
            // we reimbourse in token deposited
            uint256 amountToUse = restToReimbourse > amountDeposited
                ? amountDeposited
                : restToReimbourse;
            deposit.amount = 0;
            restToReimbourse -= amountToUse;
            IERC20(deposit.token).safeTransfer(user, amountToUse);
        }

        if (restToReimbourse == 0) {
            break;
        }
    }

    factory.emitWithdraw(msg.sender, amount, indexWithdraw);
}
```

Impact

- Bypassing the minimum deposit per position.
- Ability to maliciously spam low-value deposits.

Recommendation

There are two ways of mitigation: - Force the `requestWithdrawDeposit()` to withdraw the entire deposit amount, removing the partial withdraw option. - The `processWithdrawDeposit()` should check at the end if the non-depleted deposits touched by the function remain above the minimum required amount.

Team Response

Fixed.

[M-07] User Can Bypass Maximum Deposit Limit by Exploiting Incorrect Deposit Counting Logic

Severity

Medium Risk

Description

The `userDeposit()` function in the `Vault` contract contains a critical flaw in how it tracks and enforces the maximum number of different deposits per user. The vulnerability stems from the incorrect placement of the deposit limit check relative to the early return logic that consolidates deposits of the same token type.

When a user calls `userDeposit()`, the function retrieves the current number of different deposits using `uint256 totalDifferentDeposit = userDepos.length` and then iterates through existing deposits to check if there's an unprocessed deposit with the same token. If such a deposit exists, the function consolidates the new amount with the existing deposit and returns early via return statement, completely bypassing the deposit limit validation. The critical check `if (totalDifferentDeposit > maxDifferentDepositByUser)` occurs after this consolidation logic, meaning it's only evaluated when creating entirely new deposit entries.

The fundamental issue is that deposit processing (`processed = true`) can only occur after the `investmentEnd` timestamp through the `processDeposit()` or batch processing functions. Since users can only make deposits before `investmentEnd` due to the check `if (block.timestamp > investmentEnd) revert InvestmentPeriodFinish()`, all deposits made during the investment period will have `processed = false`. This means the consolidation logic will always find matching unprocessed deposits for the same token (uscToken), causing the function to return early and bypass the limit check entirely.

The code segment responsible for this vulnerability shows that once a user makes their first deposit, all subsequent deposits of the same token will be consolidated into that existing deposit entry, and the validation logic `if (totalDifferentDeposit > maxDifferentDepositByUser)`

becomes completely unreachable. Since processing only happens after the investment period ends, users can make unlimited deposits during the investment window, effectively rendering the `maxDifferentDepositByUser` protection meaningless.

Location of Affected Code

File: contracts/vault/Vault.sol#L251

```
function userDeposit(uint256 amount) external onlyWhitelist {
    if (block.timestamp > investmentEnd) {
        revert InvestmentPeriodFinish();
    }

    address user = msg.sender;

    IERC20(uscToken).safeTransferFrom(user, address(this), amount);

    UserStat storage currentStat = usersStat[msg.sender];
    currentStat.totalDeposit += amount;

    if (currentStat.totalDeposit > maxDepositByUser) {
        revert DepositOverflowCap();
    }
    totalDepositToProcess += amount;
    if (totalDepositToProcess > maxTotalDeposit) {
        revert DepositOverflowTotalCap();
    }

    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit > 0) {
        for (uint256 index = 0; index < totalDifferentDeposit; index++) {
            uint256 indexDeposit = userDepos[index];
            DepositInfo storage deposit = deposits[indexDeposit];
            if (deposit.processed == false && deposit.token == uscToken)
               {
                // if the user has an active deposit with the same token,
                    we reuse it
                deposit.amount += amount;
                deposit.updatedDate = uint32(block.timestamp);

                factory.emitDeposit(user, indexDeposit, amount, uscToken)
                   ;
                // we leave the function here to don't duplicate deposit
                return;
            }
        }
    }

    if (totalDifferentDeposit > maxDifferentDepositByUser) {
        revert UserMakeTooMuchDeposit();
    }
```

```
    DepositInfo memory depositInfo = DepositInfo(
        user,
        amount,
        uscToken,
        uint32(block.timestamp),
        uint32(block.timestamp),
        false,
        false,
        0,
        0
    );

    uint256 newIndex = deposits.length;

    deposits.push(depositInfo);

    depositToProcess.push(newIndex);
    usersDeposit[user].push(newIndex);

    factory.emitDeposit(user, newIndex, amount, uscToken);
}
```

File: contracts/vault/Vault.sol#L317

```
/// @notice User call this method to claim his share of token
function claim() external onlyWhitelist nonReentrant {
    address user = msg.sender;
    UserStat storage userStat = usersStat[user];
    uint256 claimAmount = userStat.amountToMint;
    userStat.amountToMint = 0;
    _transfer(address(this), user, claimAmount);

    factory.emitClaim(user, claimAmount);
}
```

File: contracts/vault/Vault.sol#L328

```
/// @notice User call this method to claim his reimboursed tokens
function claimReimbourse() external onlyWhitelist nonReentrant {
    address user = msg.sender;
    UserStat storage userStat = usersStat[user];
    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit == 0 || userStat.amountToReimbourse == 0) {
        revert NothingToReimboursed();
    }

    for (uint256 index = 0; index < totalDifferentDeposit; index++) {
        uint256 indexDeposit = userDepos[index];
        DepositInfo storage deposit = deposits[indexDeposit];
        if (deposit.processed == true && deposit.reimboursed == false) {
            deposit.reimboursed = true;
            uint256 amountToReimbourse = deposit.amountReimbursed;
            userStat.amountToReimbourse -= amountToReimbourse;
            IERC20(deposit.token).safeTransfer(user, amountToReimbourse);

            factory.emitClaimReimbourse(
                user,
                indexDeposit,
                amountToReimbourse,
                deposit.token
            );
        }
    }
}
```

Impact

This vulnerability completely nullifies the `maxDifferentDepositByUser` restriction during the investment period, which is the only time when users can make deposits. The protection mechanism becomes entirely ineffective, allowing users to bypass the intended safeguards against maximum different deposits by user.

Recommendation

The vulnerability can be resolved by restructuring the deposit limit validation logic to occur at the beginning of the function and implementing proper deposit counting that increments with each deposit attempt. The `totalDifferentDeposit` variable should be incremented every time a user makes a deposit, regardless of whether it gets consolidated with an existing deposit or creates a new entry.

Team Response

Fixed.

[M-08] Late Depositors Can Manipulate Share Allocation to the Detriment of Early Investors

Severity

Medium Risk

Description

The vulnerability stems from the `Vault` contract's share allocation mechanism, which processes deposits in FCFS order but calculates shares based solely on the final deposited amounts at the end of the investment window. Malicious users can exploit this by depositing a minimal amount (`minDepositByUser`) early to secure a position in the `depositToProcess` queue and then injecting the majority of their funds just before the `investmentEnd` deadline.

The `userDeposit()` function appends deposits to the `depositToProcess` array, preserving their priority, while the `_batchProcess` function distributes shares proportionally to the final amounts. Since the contract does not track deposit duration or enforce time-weighted contributions, late depositors receive the same share allocation as early depositors who committed funds for the entire period. This undermines the intended fairness of the FCFS system.

Location of Affected Code

File: contracts/vault/Vault.sol#L251

```
function userDeposit(uint256 amount) external onlyWhitelist {
    if (block.timestamp > investmentEnd) {
        revert InvestmentPeriodFinish();
    }

    address user = msg.sender;

    IERC20(uscToken).safeTransferFrom(user, address(this), amount);

    UserStat storage currentStat = usersStat[msg.sender];
    currentStat.totalDeposit += amount;

    if (currentStat.totalDeposit > maxDepositByUser) {
        revert DepositOverflowCap();
    }

    totalDepositToProcess += amount;
    if (totalDepositToProcess > maxTotalDeposit) {
        revert DepositOverflowTotalCap();
    }
```

```
    uint256[] memory userDepos = usersDeposit[user];
    uint256 totalDifferentDeposit = userDepos.length;
    if (totalDifferentDeposit > 0) {
        for (uint256 index = 0; index < totalDifferentDeposit; index++) {
            uint256 indexDeposit = userDepos[index];
            DepositInfo storage deposit = deposits[indexDeposit];
            if (deposit.processed == false && deposit.token == uscToken)
               {
                // if the user has an active deposit with the same token,
                    we reuse it
                deposit.amount += amount;
                deposit.updatedDate = uint32(block.timestamp);

                factory.emitDeposit(user, indexDeposit, amount, uscToken)
                   ;
                // we leave the function here to don't duplicate deposit
                return;
            }
        }
    }

    if (totalDifferentDeposit > maxDifferentDepositByUser) {
        revert UserMakeTooMuchDeposit();
    }
    DepositInfo memory depositInfo = DepositInfo(
        user,
        amount,
        uscToken,
        uint32(block.timestamp),
        uint32(block.timestamp),
        false,
        false,
        0,
        0
    );

    uint256 newIndex = deposits.length;

    deposits.push(depositInfo);

    depositToProcess.push(newIndex);
    usersDeposit[user].push(newIndex);

    factory.emitDeposit(user, newIndex, amount, uscToken);
}
```

Impact

Early investors who deposited meaningful amounts for the full duration receive no advantage, while latecomers can manipulate the system to claim a disproportionate share of minted tokens. In ex-

treme cases, if the vault reaches `maxTotalDeposit`, late large deposits could entirely displace earlier smaller ones, leaving honest participants with zero allocation despite their early commitment.

Recommendation

To restore fairness, the share allocation mechanism should incorporate time-weighted accounting, ensuring that deposits earn shares proportional to both their amount and duration.

Team Response

Fixed.

[M-09] Operators Can Cause Incorrect Share Distribution Due to Off-Chain Calculation Errors Affecting Depositors

Severity

Medium Risk

Description

The `Vault` contract contains a vulnerability where all share calculations are performed off-chain by operators and then input into the contract without any validation mechanisms. The vulnerability is present in the `processDeposit()` and `startBatchProcess()` functions, where operators manually specify the `amountMinted` and `amountReimbursed` parameters based on calculations performed outside the blockchain. In `processDeposit()`, operators can input arbitrary values for `amountMinted` and `amountReimbursed` without the contract verifying these amounts against the original deposit or any predetermined conversion formula.

Similarly, in `startBatchProcess()`, operators determine the `totalAmountAccepted` and `totalAmountMinted` values through external calculations, which then become the foundation for all individual share distributions in the `_batchProcess()` function. The contract uses the formula `mint = (restToProcess * totalRoundMint) / totalRoundAccepted` to calculate individual allocations, but since the core parameters `totalRoundMint` and `totalRoundAccepted` originate from off-chain computations, any errors in these calculations propagate to all affected users.

The Vault contract contains a centralization vulnerability as well, where operators possess complete control over share distribution calculations and minting processes. The vulnerability manifests in two primary functions: `processDeposit()` and `startBatchProcess()`.

Location of Affected Code

File: contracts/vault/Vault.sol#L361

```
function processDeposit(
    uint256 processIndex,
    uint256 amountMinted,
    uint256 amountReimbursed
) external onlyOperator {
    if (inBatchProcess) {
        // if we process during a batch, the caclculation will be
           incorrect
        revert BatchNotFinished();
    }
    _mint(address(this), amountMinted);

    uint256 indexDeposit = depositToProcess[processIndex];
    _processDeposit(indexDeposit, amountMinted, amountReimbursed);
    _removeDepositToProcess(processIndex);
}
```

File: contracts/vault/Vault.sol#L382

```
/// @notice Process many deposit by operator
/// @dev batch process base of total token distrubuted for amount
   accepted
/// @param totalAmountAccepted amount accepted for this round, overflow
   amount will be reimboursed
/// @param totalAmountMinted share of token minted for all deposits
/// @param maxProcess number of deposit to handle by batch, we put a max
   number to prevent from gas usage failure
function startBatchProcess(
    uint256 totalAmountAccepted,
    uint256 totalAmountMinted,
    uint256 maxProcess
) external onlyOperator {
    if (inBatchProcess) {
        revert BatchNotFinished();
    }

    _mint(address(this), totalAmountMinted);

    uint256 totalProcess = depositToProcess.length;
    uint256 length = maxProcess > totalProcess ? totalProcess :
       maxProcess;

// store information in the contract to continue the batch
    totalRoundMint = totalAmountMinted;
    totalRoundAccepted = totalAmountAccepted;
    totalRoundProcessed = 0;

    _batchProcess(length);

// if not all deposit was processed we stay in batch process
    inBatchProcess = depositToProcess.length > 0;
}
```

Impact

The reliance on off-chain calculations creates substantial risk for depositors who may receive incorrect share allocations due to computational errors or data processing mistakes. Users could receive significantly fewer or more shares than warranted if operators underestimate returns or make calculation errors when determining totalAmountMinted values.

Recommendation

Should ensure all the calculations in the backend are correct, and users should not be affected.

Team Response

Fixed.

[L-01] The `_removeWithdrawToProcess()` Function Is Left Unused

Severity

Low Risk

Description

The contract contains an internal `_removeWithdrawToProcess()` function, similar to its deposit counterpart, however, it is left completely unused in any operation. Judging by the current context, this function will either be: - left unused completely - intended for a batch withdrawal mechanism that does not exist in the system, and is probably a leftover

At present, the function is not required.

Location of Affected Code

File: contracts/vault/Vault.sol#L710

```
function _removeWithdrawToProcess(uint256 index) private {
    if (index >= withdrawToProcess.length) return;

    if (index > 0) {
// replace the current element by the latest
        uint256 lastElement = withdrawToProcess.length - 1;
        withdrawToProcess[index] = withdrawToProcess[lastElement];
    }
// remove the latest element from the list to reduce its size
    withdrawToProcess.pop();
}
```

Recommendation

Remove the function entirely. In case a batch withdrawal is decided to be implemented, the contracts can be upgraded accordingly.

Team Response

Fixed.

shieldify

Thank you!