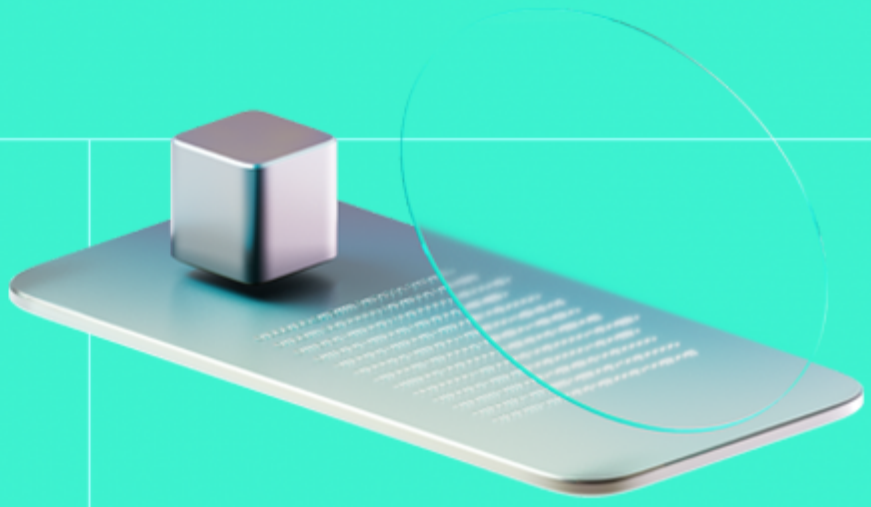
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Credbull

Date: 17/12/2024

We express our gratitude to the Credbull team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Credbull manages the first licensed on-chain private credit fund that emphasizes governance and transparency in strategy, risk management, and off-chain asset allocation. This is done through Vaults that will provide ERC1155 tokens (shares) to users in exchange for their ERC20 USDC (assets) deposits. Users will later be able to redeem their shares for to recover their deposits and some extra yield, generated from an off-chain strategy.

Document

Name	Smart Contract Code Review and Security Analysis Report for Credbull
Audited By	David Camps Novi, Paul Clemson
Approved By	Ataberk Yavuzer
Website	https://credbull.io/
Changelog	23/10/2024 - Preliminary Report
	17/12/2024 - Final Report
Platform	Plume Network
Language	Solidity
Tags	Vault, Proxy, ERC1155
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/credbull/credbull-defi
Commit	Initial commit - a3316f3; final commit - e38e285.

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

7	7	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	2
Medium	0
Low	3

Vulnerability	Severity
F-2024-6592 - Incorrect validation of spender approval in _withdraw allows theft of user funds	Critical
F-2024-6665 - Public method allows malicious users to cause DoS on other users withdrawals	High
F-2024-6713 - Users can earn yield while depositing funds for only a few seconds	High
F-2024-6693 - The optimize function fails for deposit/redeem amounts less than $1	Low
F-2024-6700 - Redeem Requests Cannot be Cancelled or Modified Until Redeem Period	Low
F-2024-6708 - Optimize function receives incorrect owner in requestRedeem leading to incorrect request data being stored	Low
F-2024-6699 - Incorrect Order of Parameters Result in Wrong Calculations	Info

Documentation quality

- Functional requirements are provided.
- Technical description are provided.

Code quality

- NatSpec is included.
- The development environment is configured.

Test coverage

Code coverage of the project is **95%** (branch coverage).

- Deployment and basic user interactions are covered with tests.

Table of Contents

System Overview **6**

Privileged Roles 6

Potential Risks **7**

Findings **8**

Vulnerability Details 8

Observation Details 27

Disclaimers 30

Appendix 1. Definitions **31**

Severities 31

Potential Risks 31

Appendix 2. Scope **32**

Appendix 3. Additional Valuables **34**

System Overview

Credbull manages the first licensed on-chain private credit fund that emphasizes governance and transparency in strategy, risk management, and off-chain asset allocation. This is done through Vaults that will provide ERC-1155 tokens (shares) to users in exchange for their ERC20 USDC (assets) deposits. Users will later be able to redeem their shares for to recover their deposits and some extra yield, generated from an off-chain strategy.

The project consists of the following contracts:

- **AbstractYieldStrategy.sol** - calculates the number of periods based on the input time bonds.
- **CalcDiscounted.sol** - helper library to calculate principals and yields.
- **CalcInterestMetadata.sol** - defines context variables.
- **CalcSimpleInterest.sol** - helper library to calculate yield.
- **RedeemOptimizerFIFO.sol** - provides the optimal withdrawal requests based on the desired shares or assets to be obtained.
- **Timer.sol** - helper library to calculate periods and timestamps.
- **TripleRateContext.sol** - defines context variables.
- **LiquidContinuousMultiTokenVault.sol** - main entry-point of the protocol for the users to deposit and withdraw assets in exchange for some shares/yield.
- **MultiTokenVault.sol** - baseline vault contract inherited by the LiquidContinuousMultiTokenVault.
- **TimelockAsyncUnlock.sol** - manages the requests to unlock assets from the vault.
- **SimpleInterestYieldStrategy.sol** - defines a yield strategy based on the number of periods.
- **TripleRateYieldStrategy.sol** - defines a yield strategy based on a combination of full maturity periods and the other periods.

Privileged roles

- **Operator** can
  - Change the addresses of `_redeemOptimizer`, `_yieldStrategy`.
  - Set a new interest rate and the vault start timestamp.
  - Pause and unpause the contract.
  - Lock the tokens of a user (requires ERC20 allowance).
- **Upgrader** can
  - Upgrade the vault contracts.
- **Asset Manager** can
  - Withdraw assets from the vault.

Potential Risks

- The project utilizes Solidity version `0.8.20` or higher, which includes the introduction of the `PUSH0` (`0x5f`) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The funds held by the contract depend on their correct management by the system admins: they must ensure the contract will always have the required funds to fulfil each redeem request with the corresponding interest.
- The project iterates over large dynamic arrays, which leads to excessive gas costs, risking denial of service due to out-of-gas errors, directly impacting contract usability and reliability.
- The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- The yield strategy used in the protocol is defined by the system role `OPERATOR_ROLE`, relying on their correct setup to have a proper functioning of the yield calculations.
- In the TripleRateYieldStrategy contract, if a deposit was made prior to the latest interest valid timestamp, the system will take into account the previous period's interest in order to compute the yield obtained by the user. However, this check assumes there is only a single previous interest period (i.e. current interest plus a previous period only); if there was an even earlier interest period (i.e. current interest plus two previous periods), or more, the earliest periods would not be taken into account. Although it is technically possible to have more than two interest periods, the development team communicated that each vault will never have more than two.
- The system includes a pausable feature, which allows the `OPERATOR_ROLE` to halt any transfer of shares at will: deposits, withdraws and token transfers cannot execute.
- Protocol users should note that yield will be calculated for deposits up to the period in which they perform a request for redeeming their funds. If, per example, the notice period to withdraw funds is 5 days, none of these 5 days will be accounted for yield calculations.

Findings

Vulnerability Details

F-2024-6592 - Incorrect validation of spender approval in _withdraw allows theft of user funds - Critical

Description:

The `MultiTokenVault` is an ERC-4626 inspired vault implementation which allows users to deposit ERC-20 `ASSET` tokens and earn a yield on the deposited principal. In return they receive ERC-1155 tokens equal to the amount of `ASSET` they deposited.

When later a user attempts to withdraw their `ASSET` token (as well as any earned yield) from the `MultiTokenVault` the internal `_withdraw` function contains a check to confirm that the user attempting to withdraw (the `caller`) is either the `owner` of the ERC-1155 shares tokens, or that they have been given approval by the owner to spend their tokens.

However the current implementation of this check instead allows users who have not been approved by the token's owner to transfer funds, while not allowing users who have received approval to proceed.

```
        if (caller != owner && isApprovedForAll(owner, caller)) {
            revert MultiTokenVault__CallerMissingApprovalForAll(caller, owner)
;
        }
```

This error, combined with the `caller` being allowed to specify their own `receiver` address in the user facing `redeemForDepositPeriod` function (which calls the previously mentioned `_withdraw` function) will mean that a malicious user is able to withdraw any other users tokens to an address that they control, leading to a significant risk of the loss of all funds in the protocol.

Assets:

- token/ERC1155/MultiTokenVault.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact:	5/5
Likelihood:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Critical

Recommendations

Remediation: The approval check in `_withdraw` should be changed to the `!isApprovedForAll(owner, caller)` in order to prevent this issue to occur:

```
        if (caller != owner && !isApprovedForAll(owner, caller)) {
            revert MultiTokenVault__CallerMissingApprovalForAll(caller, owner)
;
        }
```

Resolution: Fixed in commit ID `d860230`: the reported check was updated as recommended to

```
        if (caller != owner && !isApprovedForAll(owner, caller)) {
            revert MultiTokenVault__CallerMissingApprovalForAll(caller, owner)
;
        }
```

Evidences

Foundry Proof of Concept

Reproduce: Add the following tests to `MultiTokenVault.t.sol` to confirm this issue:

```
    // Confirm address with approval cannot spend users tokens
    function test_ApprovedWithdrawalNotAllowed() public {
        uint256 assetToSharesRatio = 1;
        IMultiTokenVault vault = _createMultiTokenVault(_asset, assetToSharesR
atio, 10);
        address vaultAddress = address(vault);
        uint256 depositPeriod = _testParams1.depositPeriod;
        _warpToPeriod(vault, depositPeriod);
        // Alice deposits into the vault
        vm.startPrank(_alice);
        _asset.approve(vaultAddress, _testParams1.principal);
```

```
        vault.deposit(_testParams1.principal, _alice);
        // Alice approves bob as a spender of her shares
        vault.setApprovalForAll(_bob, true);
        vm.stopPrank();

        // Move forward in time
        uint256 redeemPeriod = _testParams1.redeemPeriod;
        _warpToPeriod(vault, redeemPeriod);

        // Need to send the earned yield to the vault to cover the withdrawal
        deal(address(_asset), address(vault), 5e9);

        // Bob cannnot withdraw alices tokens despite having permission
        vm.prank(_bob);
        vm.expectRevert();
        vault.redeemForDepositPeriod(_testParams1.principal, _bob, _alice, _te
stParams1.depositPeriod, _testParams1.redeemPeriod);
    }

    // Confirm address without approval can withdraw users tokens
    function test_UnapprovedWithdrawalAllowed() public {
        uint256 assetToSharesRatio = 1;
        IMultiTokenVault vault = _createMultiTokenVault(_asset, assetToSharesR
atio, 10);
        address vaultAddress = address(vault);
        uint256 depositPeriod = _testParams1.depositPeriod;
        _warpToPeriod(vault, depositPeriod);
        // Alice deposits to the vault
        vm.startPrank(_alice);
        _asset.approve(vaultAddress, _testParams1.principal);
        vault.deposit(_testParams1.principal, _alice);
        vm.stopPrank();

        // Mo
```

See more

F-2024-6665 - Public method allows malicious users to cause DoS on other users withdrawals - High

Description:

The `LiquidContinuousMultiTokenVault` is an ERC-4626 inspired vault implementation which allows users to deposit ERC-20 `ASSET` tokens and earn a yield on the deposited principal. In return they receive ERC-1155 tokens equal to the amount of `ASSET` they deposited. This contract builds on the `MultiTokenVault` to add a number of features including a two step withdrawal process where the user must first call `requestSell`, then wait a specified time period before then calling `executeSell` to complete the withdrawal of their assets.

To handle this necessary wait period the contract inherits `TimelockAsyncUnlock` which stores data on the users unlock request during the `requestSell` logic flow, and then removes it once the user calls `executeSell`.

```
    function unlock(address owner, uint256 requestId) public virtual
        returns (uint256[] memory depositPeriods, uint256[] memory amounts) {
    }

    function _unlock(address owner, uint256 depositPeriod, uint256 requestId,
uint256 amountToUnlock) public virtual {
    }
```

However the problem arises because the `unlock` and `_unlock` functions in `TimelockAsyncUnlock` have public visibility allowing users to call them outside of the `LiquidContinuousMultiTokenVault` withdrawal context, meaning the requests will be deleted but the underlying tokens will not be transferred. Additionally these functions allow the caller to pass any `owner` meaning they can be used to delete the withdrawal request of any user.

It should be noted that this does not cause an instant permanent locking of funds because the affected user is able to create another withdrawal request, and attempt to withdraw their funds again after waiting the specified timelock period. However a sophisticated attacker could keep track of a user's unlock time and use a bot to repeatedly call `unlock` on their pending withdrawal request as soon as the request's timelock period is over, making it very difficult for a regular user to successfully withdraw their funds.

The economic viability of prolonging this type of attack depends on both the transaction costs on the blockchain in question as well as the sizes of the withdrawals the attacker is able to grief. As this protocol intends to launch on an Ethereum layer two (Plume) and

has intentions of handling multiple millions of dollars from institutional size investors it could be the case that a malicious user is able to seriously halt the withdrawal of significant amounts of capital for only a few cents per day.

Assets:

- timelock/TimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]
- yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: The `unlock` and `_unlock` functions in `TimelockAsyncUnlock` should both have `internal` visibility as a user calling them outside of the full context of token withdrawal causes unintended consequences in the system.

Resolution: Fixed in commit ID `d860230`: the `_unlock` function visibility was changed to `internal`, and a the following check was added into `unlock` to ensure shares can only be unlock by their owners.

```
function _authorizeCaller(address caller, address owner) internal virtual {
    if (caller != owner) {
        revert TimelockAsyncUnlock__AuthorizeCallerFailed(caller, owner);
    }
}
```

Evidences

Foundry Proof of Concept

Reproduce:

Add the following test to `LiquidContinuousMultiTokenVault.t.sol` to confirm this issue:

```
    function test_MaliciousUnlock() public {
        LiquidContinuousMultiTokenVault liquidVault = _liquidVault; // _create LiquidContinueMultiTokenVault(_vaultParams);
        TestParam memory testParams = TestParam({ principal: 2_000 * _scale, depositPeriod: 11, redeemPeriod: 70 });
        uint256 sharesAmount = testParams.principal; // 1 principal = 1 share
        // ---------------- buy (deposit) ----------------
        _warpToPeriod(liquidVault, testParams.depositPeriod);
        vm.startPrank(alice);
        _asset.approve(address(liquidVault), testParams.principal); // grant the vault allowance
        liquidVault.requestBuy(testParams.principal);
        vm.stopPrank();
        _warpToPeriod(liquidVault, testParams.redeemPeriod - liquidVault.noticePeriod());
        // Calc Alice's balance before the withdrawal attempt
        uint256 aliceAssetBalanceBefore = _asset.balanceOf(alice);
        console.log("Before", aliceAssetBalanceBefore);
        // requestSell
        vm.prank(alice);
        uint256 aliceRequest = liquidVault.requestSell(testParams.principal);
        // Send alices yield to contract
        deal(address(_asset), address(liquidVault), 5e9);
        _warpToPeriod(liquidVault, testParams.redeemPeriod - liquidVault.noticePeriod() + 1);

        // Bob calls unlock on alice's request
        vm.prank(bob);
        liquidVault.unlock(alice, aliceRequest);
        // Confirm alice's assets were not withdrawn by bob calling unlock
        assert(_asset.balanceOf(alice) == aliceAssetBalanceBefore);

        // Alice's sell now reverts because bobs call to unlock cleared the deposit request without withdrawing alice's tokens
        vm.prank(alice);
        vm.expectRevert(abi.encodeWithSelector(
            LiquidContinuousMultiTokenVault.LiquidContinuousMultiTokenVault__InvalidComponentTokenAmount.selector,
```

See more

F-2024-6713 - Users can earn yield while depositing funds for only a few seconds - High

Description:

In the `LiquidContinuousMultiTokenVault` contract, users earn yield on their deposits based on the number of periods (24 hours) that have passed since their funds were deposited into the protocol. The number of periods that will be taken into account are calculated by `AbstractYieldStrategy::_noOfPeriods()`:

```
/**
 * @notice Calculate the number of periods in effect for Yield Calculation.
 * @dev Encapsulates the algorithm for determining the number of periods to c
alculate yield with. The calculation is:
 *      noOfPeriods = (`to_` - `from_`)
 *
 * @param from_ The from period
 * @param to_ The to period
 * @return noOfPeriods_ The calculated effective number of periods.
 */
function _noOfPeriods(uint256 from_, uint256 to_) internal pure virtual retur
ns (uint256 noOfPeriods_) {
    return to_ - from_;
}
```

The protocol requires the users to go through a two-step redemption process: first, the user requests a withdrawal via `LiquidContinuousMultiTokenVault::requestRedeem()` during any of the 24 hours periods, and then they must wait until the `noticePeriod()` passed in order to execute the withdrawal through `LiquidContinuousMultiTokenVault::redeem()`.

```
function minUnlockPeriod() public view virtual returns (uint256 minUnlockPeri
od_) {
    return currentPeriod() + noticePeriod();
}
```

However, the current protocol implementation includes the redeem period as part of the yield calculation, resulting in an inflation of the interest obtained by the user:

```
function convertToAssetsForDepositPeriod(uint256 shares, uint256 depositPerio
d, uint256 redeemPeriod)
    public
    view
    override
```

```
    returns (uint256 assets)
{
    if (shares < SCALE) return 0; // no assets for fractional shares

    if (redeemPeriod < depositPeriod) return 0; // trying to redeem before de
positPeriod

    uint256 principal = shares; // 1 share = 1 asset.  in other words 1 share
= 1 principal

    return principal + calcYield(principal, depositPeriod, redeemPeriod);
}
```

As a result, it is possible for a user to deposit a large amount of funds shortly before the end of a period, immediately trigger a redemption request, and then be able to redeem their principal assets plus one full day's yield shortly afterwards.

This is problematic for two main reasons:

- As the funds will only be in the contract for a few seconds the protocol will not be able to use them for any off chain yield generating activities, whilst still having to pay out yield to the user.
- The funds used to cover this yield will be funds sitting in the smart contract, which will likely either be recent deposits of other users or funds earmarked for other users pending redemption requests.

Even though the yield accrued via this method is only a small amount percentage wise, a user with a large amount of capital could benefit from repeatedly earning a small profit every time with very little at risk.

This inflated interest exposed above is also affecting the public methods `convertToAssets()`, `convertToAssetsForDepositPeriodBatch()` and `redeemForDepositPeriod()`.

Assets:

- token/ERC1155/MultiTokenVault.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/RedeemOptimizerFIFO.sol [https://github.com/credbull/credbull-defi/]
- yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact:	3/5
Likelihood:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	High

Recommendations

Remediation: The protocol should calculate a deposits earned yield from the period in which they deposited until `redeemPeriod - 1`. This change would mean users would have to deposit their tokens into the protocol for at least one full 24 hour period before they start earning any yield.

Resolution: Fixed in commit ID `968c1f3`: the redeem period is now subtracting the `noticePeriod()`, which avoids the inclusion of any extra period beyond the request for yield calculations.

```
/// @dev yield accrues up to the `requestRedeemPeriod` (as opposed to the `re
deemPeriod`)
function calcYield(uint256 principal, uint256 depositPeriod, uint256 redeemPe
riod) public view returns (uint256 yield) {
    uint256 requestRedeemPeriod = redeemPeriod > noticePeriod() ? redeemPerio
d - noticePeriod() : 0;

    if (requestRedeemPeriod <= depositPeriod) return 0; // no yield when depo
sit and requestRedeems are the same period

    return _yieldStrategy.calcYield(address(this), principal, depositPeriod,
requestRedeemPeriod);
}
```

Evidences

Foundry Proof of Concept

Reproduce: Add the following test to `LiquidContinuousMultiTokenVaultTest.t.sol` to highlight this issue:

```
    function test_AbuseDepositTime() public {
      // Give Vault funds to cover yield
```

```
        deal(address(_asset), address(_liquidVault), 100e6);

        // Calc Alice's balance at start
        uint256 aliceAssetsStart = _asset.balanceOf(alice);
        vm.warp(block.timestamp + 48 hours - 1);

        // Deposit at the end of a period
        vm.startPrank(alice);
        _asset.approve(address(_liquidVault), 10_000e6);
        _liquidVault.deposit(10_000e6, alice);

        // Immediately request withdraw
        _liquidVault.requestRedeem(10_000e6, address(0), alice);
        vm.stopPrank();

        // New period begins a few seconds later
        vm.warp(block.timestamp + 2);

        // Withdraw
        vm.startPrank(alice);
        _liquidVault.redeem(10_000e6, alice, address(0));

        // Confirm Profit Made
        uint256 aliceAssetsEnd = _asset.balanceOf(alice);
        assert(aliceAssetsEnd > aliceAssetsStart);
        uint256 aliceProfit = aliceAssetsEnd - aliceAssetsStart;
        console.log("Alice Profit", aliceProfit);
    }
```

Results:

The test returns the following result, signifying alice earned ~$1.53 on her deposit, despite only having her funds in the protocol for 2 seconds.

```
Alice Profit 1527777
```

F-2024-6693 - The optimize function fails for deposit/redeem amounts less than $1 - Low

Description:

The `LiquidContinuousMultiTokenVault` allows users to deposit any amount of tokens and earn a stable yield on their deposits. Depending on the time period of a specific deposit, a users `shares` may be eligible to different rates of yield. The contract's logic includes a call to a `IRedeemOptimizer` which is responsible for selecting the shares that would generate the optimal yield when calling redeem. `RedeemOptimizerFIFO` is a basic version of this redeem contract which simply uses the oldest possible deposits to cover the users redeem request. During a users call to `redeem` a call is made to `IRedeemOptimizer::optimize` where this process takes place.

However a problem arises when the user is depositing/redeeming small amounts of tokens (less than $1). In these instances the optimizer will ignore the smaller values and revert with a `RedeemOptimizer__OptimizerFailed` error, suggesting that it has failed to find the necessary shares required to fulfil the redemption.

This means whenever a users deposit amount over a given period is a small amount (less than $1), these funds will not be redeemable causing small amounts of locked user funds.

Assets:

- token/ERC1155/RedeemOptimizerFIFO.sol [https://github.com/credbull/credbull-defi/]
- yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: The project should introduce a minimum deposit amount to the `LiquidContinuousMultiTokenVault` contract to ensure that users cannot unintentionally end up with stuck funds. All fuzz tests using amounts > 1e6 ($1 in USDC) did not have this problem, so any minimum amount of 1e6 or greater should sufficiently mitigate this risk.

Resolution: Fixed in commit ID `d860230`: a minimum `SCALE` value of `10 wei` was implemented, resulting in no dust values anymore.

```
/// minimum shares required to convert to assets and vice-versa.
function _minConversionThreshold() internal view returns (uint256 minConversi
onThreshold) {
    return SCALE < 10 ? SCALE : 10;
}
```

Evidences

Foundry Proof of Concept

Reproduce: The following foundry test will outline this issue

```
    function test_dust() public {
      // Give Vault funds to cover yield
      deal(address(_simpleAsset), address(_simpleYieldVault), 100_000e6);
      // Alice deposits a small amount
      uint256 dustAmount = 1e6 - 543;
      deal(address(_simpleAsset), alice, dustAmount);
      vm.startPrank(alice);
      _simpleAsset.approve(address(_simpleYieldVault), dustAmount);
      _simpleYieldVault.deposit(dustAmount, alice);
      vm.warp(block.timestamp + 24 hours);
      _simpleYieldVault.requestRedeem(dustAmount, address(0), alice);
      vm.warp(block.timestamp + 24 hours);

      // Redeem will fail for this small depost/redeem combination
      vm.expectRevert()
      _simpleYieldVault.redeem(dustAmount, alice, address(0));
  }
```

F-2024-6700 - Redeem Requests Cannot be Cancelled or Modified Until Redeem Period - Low

Description:

Users will redeem their vault shares in two steps: first, they will call `requestRedeem()` in order to create a redeem request, and then they will execute the request via `redeem()`. However, the creation of requests is additive (as far as shares are available) and cannot be deleted. If a user wants to change the amount to redeem, or cancel the redeem, it will not be possible; they will be forced to execute the redeem request.

When a user wants to sell their shares in exchange for assets, they will call `requestRedeem()` in `LiquidContinuousMultiTokenVault`. This will trigger a query to the `RedeemOptimizerFIFO` contract, which will return the arrays containing the user's available shares data (`depositPeriods` and `sharesAtPeriods`), obtained from `_sharesAvailableAtPeriod()`.

```
function _sharesAvailableAtPeriod(
    IMultiTokenVault vault,
    OptimizerParams memory optimizerParams,
    uint256 depositPeriod
) internal view returns (uint256 sharesAvailable_) {
    bytes4 timelockInterfaceId = type(ITimelockAsyncUnlock).interfaceId;

    if (vault.supportsInterface(timelockInterfaceId)) {
        ITimelockAsyncUnlock timelockVault = ITimelockAsyncUnlock(address(vau
lt));
        return timelockVault.maxRequestUnlock(optimizerParams.owner, depositP
eriod);
    } else {
        return vault.sharesAtPeriod(optimizerParams.owner, depositPeriod);
    }
}
```

These arrays of available shares will then be used in `requestUnlock()`, and `_handleSingleUnlockRequest()`, in order to create the unlock requests stored in the state variable `_unlockRequests`:

```
function _handleSingleUnlockRequest(address owner, uint256 depositPeriod, uin
t256 requestId, uint256 amount)
    internal
    virtual
{
    if (amount > maxRequestUnlock(owner, depositPeriod)) {
        revert TimelockAsyncUnlock__ExceededMaxRequestUnlock(
```

```
            owner, depositPeriod, amount, maxRequestUnlock(owner, depositPeri
od)
        );
    }

    EnumerableMap.UintToUintMap storage unlockRequestsForRequestId = _unlockR
equests[owner][requestId];
    EnumerableMap.UintToUintMap storage depositPeriodAmountCache = _depositPe
riodAmountCache[owner];

    uint256 unlockAmountByUnlockPeriod =
        unlockRequestsForRequestId.contains(depositPeriod) ? unlockRequestsFo
rRequestId.get(depositPeriod) : 0;
    unlockRequestsForRequestId.set(depositPeriod, unlockAmountByUnlockPeriod
+ amount);

    uint256 unlockAmountByOwner =
        depositPeriodAmountCache.contains(depositPeriod) ? depositPeriodAmoun
tCache.get(depositPeriod) : 0;

    depositPeriodAmountCache.set(depositPeriod, unlockAmountByOwner + amount)
;
}
```

Due to its design mechanism, the method `_handleSingleUnlockRequest()` will only be able to increase the amount of shares to withdraw from each period. It will not be able to either decrease or override the amount. Additionally, if a user (Alice) creates a request and sends some shares to another user (Bob), none of them can withdraw or create new requests until the redeem period is reached,

As a result, users will not be able to cancel or modify their requests until they are executed or fail within very specific scenarios (e.g. be able to increase the amount of shares to withdraw from a single period).

Assets:

- yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/RedeemOptimizerFIFO.sol [https://github.com/credbull/credbull-defi/]
- timelock/TimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact:	2/5
Likelihood:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: It is recommended to either implement a method to delete unlock requests or modify the mechanism to create unlock requests so that it overrides previous requests stored in `_unlockRequests`.

Resolution: Fixed in commit ID `d860230`: a request can now be cancelled before the notice period ends with the implementation of the following method.

```
/// @dev Cancel a pending request to unlock
function cancelRequestUnlock(address owner, uint256 requestId)
    public
    onlyAuthorized(owner)
{
    (uint256[] memory depositPeriods, uint256[] memory amounts) = unlockReque
sts(owner, requestId);

    for (uint256 i = 0; i < depositPeriods.length; ++i) {
        _unlock(owner, depositPeriods[i], requestId, amounts[i]);
    }

    emit CancelRedeemRequest(owner, requestId, _msgSender());
}
```

F-2024-6708 - Optimize function receives incorrect owner in requestRedeem leading to incorrect request data being stored - Low

Description:

Within the `LiquidContinuousMultiTokenVault` contract, withdrawing funds from the contract requires a two step process. The first of these steps is calling `requestRedeem`.

```
    function requestRedeem(uint256 shares, address, /* controller */ address
owner)
        public
        returns (uint256 requestId_)
    {
        // using optimize() variant in case "shares" represents the IComponen
t "principal + yield" which is our "assets".
        (uint256[] memory depositPeriods, uint256[] memory sharesAtPeriods) =
            _redeemOptimizer.optimize(this, owner, shares, shares, minUnlockP
eriod());

        uint256 requestId = requestUnlock(_msgSender(), depositPeriods, share
sAtPeriods);
        emit RedeemRequest(_msgSender(), owner, requestId, _msgSender(), shar
es);
        return requestId;
    }
```

This function allows the caller to pass an `owner` parameter. This causes an issue for two reasons:

- There is no verification that the caller is the `owner`.
- The `depositPeriods` and `sharesAtPeriods` arrays are built using this `owner` while the actual `requestUnlock` passed `_msgSender()` meaning incorrect data will be stored for the users unlock request if they pass any `owner` other than themselves.

This means that if a user passes any `owner` other than themselves to `requestRedeem` their eventual `redeem` call will fail because the user will very likely have different `depositPeriods` and `sharesAtPeriods` than those stored during `requestRedeem`.

Status: Fixed

Classification

Impact:	2/5
Likelihood:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: This issue should be fixed by either:

- Not using the `owner` parameter and instead using `_msgSender()` when calling `optimize`
- Requiring that `owner == _msgSender()` when calling `requestRedeem`

Resolution: Fixed in commit ID `d860230`: the modifiers `onlyAuthorized` and `onlyController` were added into the reported method, making sure only the owner of the shares can create their own requests.

F-2024-6699 - Incorrect Order of Parameters Result in Wrong Calculations - Info

Description:

The `public` method `calcPrice()` provides information to any caller about the pricing obtained for a certain time period in the corresponding vault contract:

```
function calcPrice(address contextContract, uint256 numPeriodsElapsed)
    public
    view
    virtual
    returns (uint256 price)
{
    if (address(0) == contextContract) {
        revert IYieldStrategy_InvalidContextAddress();
    }
    ITripleRateContext context = ITripleRateContext(contextContract);

    return CalcSimpleInterest.calcPriceFromInterest(
        numPeriodsElapsed, context.rateScaled(), context.frequency(), context
.scale()
    );
}
```

However, the call `CalcSimpleInterest.calcPriceFromInterest()` will return incorrect calculations, resulting in a wrong pricing information. This is due to the inverted order of the first to parameters introduced, `numPeriodsElapsed` and `context.rateScaled()`, compared to the function `calcPriceFromInterest()`:

```
function calcPriceFromInterest(
    uint256 interestRatePercentScaled,
    uint256 numTimePeriodsElapsed,
    uint256 frequency,
    uint256 scale
) internal pure returns (uint256 priceScaled) {
    uint256 parScaled = 1 * scale;

    uint256 interest = calcInterest(parScaled, interestRatePercentScaled, num
TimePeriodsElapsed, frequency, scale);

    return parScaled + interest;
}
```

Assets:

- yield/strategy/TripleRateYieldStrategy.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Classification

Impact: 1/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: The order of the first two parameters should be inverted to follow the required order in `calcPriceFromInterest()`

from:

```
CalcSimpleInterest.calcPriceFromInterest(
    numPeriodsElapsed, context.rateScaled(), context.frequency(), context.sca
le()
);
```

to

```
CalcSimpleInterest.calcPriceFromInterest(
    context.rateScaled(), numPeriodsElapsed, context.frequency(), context.sca
le()
);
```

Resolution: Fixed in commit ID `d860230`: the function call was inverted to follow the required order in `calcPriceFromInterest()`:

```
CalcSimpleInterest.calcPriceFromInterest(
    context.rateScaled(), numPeriodsElapsed, context.frequency(), context.sca
le()
);
```

Observation Details

F-2024-6678 - Floating Pragma - Info

Description:

The project uses the floating `pragma ^0.8.20`.

This may result in the contracts being deployed using the wrong `pragma` version, which is different from the one they were tested with. For example, they might be deployed using an outdated `pragma` version which may include bugs that affect the system negatively.

Assets:

- yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]
- yield/CalcSimpleInterest.sol [https://github.com/credbull/credbull-defi]
- yield/CalcDiscounted.sol [https://github.com/credbull/credbull-defi]
- yield/context/TripleRateContext.sol [https://github.com/credbull/credbull-defi]
- yield/strategy/SimpleInterestYieldStrategy.sol [https://github.com/credbull/credbull-defi]
- yield/strategy/TripleRateYieldStrategy.sol [https://github.com/credbull/credbull-defi]
- timelock/Timer.sol [https://github.com/credbull/credbull-defi]
- yield/ICalcInterestMetadata.sol [https://github.com/credbull/credbull-defi]
- yield/context/ITripleRateContext.sol [https://github.com/credbull/credbull-defi]
- yield/strategy/IYieldStrategy.sol [https://github.com/credbull/credbull-defi]
- token/component/IComponentToken.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/IRedeemOptimizer.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/IMultiTokenVault.sol [https://github.com/credbull/credbull-defi]
- timelock/ITimelock.sol [https://github.com/credbull/credbull-defi]
- timelock/ITimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]
- timelock/ITimelockOpenEnded.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/RedeemOptimizerFIFO.sol [https://github.com/credbull/credbull-defi/]
- timelock/TimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]

- yield/strategy/AbstractYieldStrategy.sol [https://github.com/credbull/credbull-defi]
- token/ERC1155/MultiTokenVault.sol [https://github.com/credbull/credbull-defi]
- yield/CalcInterestMetadata.sol [https://github.com/credbull/credbull-defi]

Status: Accepted

Recommendations

Remediation: It is recommended to lock the pragma version as `0.8.20` instead of `^0.8.20`.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6721 - Missing Storage Gaps - Info

Description:

When working with upgradeable contracts, it is necessary to introduce storage gaps to allow for storage extension during upgrades.

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

Note: OpenZeppelin Upgrades checks the correct usage of storage gaps.

Assets:

- yield/CalcInterestMetadata.sol [https://github.com/credbull/credbull-defi]

Status: Fixed

Recommendations

Remediation:

- Introduce Storage Gaps in the affected contracts.

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of `uint256` so that each element reserves a 32 byte slot. Use the name `__gap` or a name starting with `__gap_` for the array so that OpenZeppelin Upgrades will recognize the gap.

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using `upgradeProxy` or just run the validations with `validateUpgrade` (see docs for Hardhat or Truffle). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

Resolution:

Fixed in commit ID `d860230`: storage gaps were implemented in `CalcInterestMetadata`.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/credbull/credbull-defi/
Commit	a3316f3
Whitepaper	https://docs.credbull.io/docs/litepaper
Requirements	https://github.com/credbull/credbull-defi/blob/docs/liquid-audit/packages/contracts/docs/src/SUMMARY.md
Technical Requirements	https://github.com/credbull/credbull-defi/tree/docs/liquid-audit/packages/contracts#readme

Asset	Type
timelock/ITimelock.sol [https://github.com/credbull/credbull-defi]	Smart Contract
timelock/ITimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]	Smart Contract
timelock/ITimelockOpenEnded.sol [https://github.com/credbull/credbull-defi]	Smart Contract
timelock/TimelockAsyncUnlock.sol [https://github.com/credbull/credbull-defi]	Smart Contract
timelock/Timer.sol [https://github.com/credbull/credbull-defi]	Smart Contract
token/component/IComponentToken.sol [https://github.com/credbull/credbull-defi]	Smart Contract
token/ERC1155/IMultiTokenVault.sol [https://github.com/credbull/credbull-defi]	Smart Contract
token/ERC1155/IRedeemOptimizer.sol [https://github.com/credbull/credbull-defi]	Smart Contract
token/ERC1155/MultiTokenVault.sol [https://github.com/credbull/credbull-defi]	Smart Contract
token/ERC1155/RedeemOptimizerFIFO.sol [https://github.com/credbull/credbull-defi/]	Smart Contract
yield/CalcDiscounted.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/CalcInterestMetadata.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/CalcSimpleInterest.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/context/ITripleRateContext.sol [https://github.com/credbull/credbull-defi]	Smart Contract

Asset	Type
yield/context/TripleRateContext.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/ICalcInterestMetadata.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/LiquidContinuousMultiTokenVault.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/strategy/AbstractYieldStrategy.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/strategy/IYieldStrategy.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/strategy/SimpleInterestYieldStrategy.sol [https://github.com/credbull/credbull-defi]	Smart Contract
yield/strategy/TripleRateYieldStrategy.sol [https://github.com/credbull/credbull-defi]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of Credbull, Hacken followed its methodology by performing fuzz-testing on the project's main functions. Forge foundry fuzz tests were used to test how the protocol handles a wide variety of inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Foundry fuzzing suite was prepared for this task, and throughout the assessment, 4 invariants were tested over 40,000 runs. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
Earned yield should match expected earned yield (SimpleInterestYieldStrategy)	Passed*	10,000
Earned yield should match expected earned yield (TripleRateYieldStrategy)	Passed*	10,000
Withdrawing any amount less than deposit amount should cause no issues	Passed*	10,000
Users should be able to withdraw full amount after a partial withdraw	Passed*	10,000

*Tests passed after ensuring the minimum amount deposited/withdrawn was greater than 1e6 (this issue was raised as part of the audit's findings)

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth

operations.