CD SECURITY

AUDIT REPORT

Ultra Markets
December 2025

Prepared by
ZanyBonzy
Varun_Sharma

Introduction

A time-boxed security review of the **Ultra Markets** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Ultra Markets**

Ultra markets is leverage protocol facilitating capital-efficient trading on prediction markets like Polymarket. Liquidity Providers (LPs) deposit USDC into the UmVault to earn yield, where their rewards are boosted by an age-weighted multiplier based on deposit duration. Traders interact with the PositionManager to borrow this liquidity, combining it with their own collateral to execute leveraged trades. Upon position settlement, the protocol pulls the proceeds back from the trader; the Vault is repaid its principal plus a configurable share of the profits, while the remaining gains are returned to the Trader.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 6242058aa02aff492afc722eecdc2a4b0af92686**

***fixes review commit hash* - 91c03e749041f3be2001dc827209c534fa3be094**

Scope

The following folders were in scope of the audit:

- `src/PositionManager.sol`
- `src/Vault.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 5 issues
- Medium: 2 issues
- Low & Info: 8 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Incorrect pulling of funds by the vault prevents repaying the funds	Critical	Fixed
[C-02]	Users Have No Incentive To Settle	Critical	Acknowledged
[C-03]	`repay` Incorrectly Updates `utilizedAssets`	Critical	Fixed
[H-01]	Share Inflation Attack via direct asset donation allows stealing user funds	High	Fixed
[H-02]	Traders Can Frontrun Liquidations to Avoid Penalties	High	Acknowledged
[M-01]	Failed Polymarket Orders Cannot Be Cancelled	Medium	Fixed
[M-02]	Sandwich Attacks on Position Settlements	Medium	Fixed
[L-01]	Incorrect assets are returned in the convert to assets function	Low	Fixed
[L-02]	Protocol Allows 1x Leverage	Low	Partially Fixed
[L-03]	Full Loss Positions Cannot Be Settled	Low	Acknowledged
[L-04]	APR Can be easily Inflated	Low	Fixed
[L-05]	Manipulation of `_entryTimestamp` via Share Transfers	Low	Fixed
[L-06]	Deposit/Withdraw Slippage Missing Protection	Low	Fixed
[I-01]	`borrow` and `repay` emits msg.sender instead of trader	Informational	Fixed
[I-02]	Use `memory` rather than `storage` for queries	Informational	Fixed

Detailed Findings

[C-01] Incorrect pulling of funds by the vault prevents repaying the funds

Severity

Impact: High

Likelihood: High

Description

Whenever a position is settled in the position manager, it repays the vault the borrowed funds, after calculating how much to send to the vault. Now the issue is when the amount in the vault is greater than zero then the contract sends that amount to vault but the main issue is in repay function in vault contract where it tries to pull the funds from the position manager contract but as there is no approval of funds from the position manager to vault this call fails and hence the settle position function also fails.

Following is settle position function

```
function settlePosition(uint256 positionId, uint256 exitPrice, uint256
returnedAmount, bool isLiquidation)
        external
        onlyOperator
        whenNotPaused
        nonReentrant
        validPositionId(positionId)
    {
       ......
................ Rest of the code
        if (amountToVault != 0) {
===>            collateralToken.safeTransfer(address(vault),
amountToVault);
===>            vault.repay(amountToVault);
        }

    .......rest of the code.
...............
    }
```

We can clearly see that the amount to vault is sent to the vault by a safe transfer call and then the repay function is called on the vault, which is as follows

```
 function repay(uint256 amount) external nonReentrant onlyPositionManager
{
        if (amount == 0) revert ZeroAssets();

        // Pull USDC from Position Manager.
===>        asset.safeTransferFrom(msg.sender, address(this), amount);
```

```
        uint256 utilized = utilizedAssets;
        if (amount >= utilized) {
            // Principal fully repaid; any excess is profit (stays in
vault balance).
            utilizedAssets = 0;
        } else {
            // Partial principal repayment.
            utilizedAssets = utilized - amount;
        }

        emit Repay(msg.sender, amount);

        // Snapshot after profit/repayment changes share price
        _snapshotYieldInternal();
    }
```

As we can see it tries to pull usdc from the position manager contract but this call fails because there is no approval of usdc to the vault contract plus the funds have already been transferred to the vault contract therefore there is no need for pulling the funds.

The settlePosition tests pass because a mock vault is used which doesn't attempt to pull tokens from the position manager.

Recommendations

Remove the safeTransferFrom call from the repay function in vault contract.

[C-02] Users Have No Incentive To Settle

Severity

Impact: High

Likelihood: High

Description

`openPosition` transfers collateral amount + borrowed amount to the trader. This is used for order execution. However, when position is closed, either by trader, or forcefully and `settlePosition` is called, the `returnedAmount` isattempted to be drawn from the trader.

This requires on pre-approval from the trader, which they may NOT do.

```
function settlePosition(...) {
    collateralToken.safeTransferFrom(p.trader, address(this),
returnedAmount);
    // ... assumes approval granted
}
```

This prevents settlements while the traders hold protocol funds, allowing them to rob the protocol.

Recommendations

Implement an alternative token handling system. A potential w

[C-03] `repay` Incorrectly Updates `utilizedAssets`

Severity

Impact: High

Likelihood: High

Description

Repayments deduct the full amount (i.e including profit, loss, or liquidation penalty) from utilizedAssets, understating outstanding debt and deferring profit recognition.

```
    function repay(uint256 amount) external nonReentrant
onlyPositionManager {
        // ...
        uint256 utilized = utilizedAssets;
        if (amount >= utilized) {
            // Principal fully repaid; any excess is profit (stays in
vault balance).
            utilizedAssets = 0;
        } else {
            // Partial principal repayment.
            utilizedAssets = utilized - amount; // <<===
        }
        // ...
    }
```

When the repaid amount exceeds the outstanding principal for a given position but other positions remain open, the function over-reduces `utilizedAssets` by treating the entire amount (principal + profit share or principal + liquidation penalty) as a debt reduction, rather than isolating the excess as an addition to free liquidity in the vault's balance. This does not align with the expected accounting model, as `utilizedAssets` is defined as the amount of underlying currently lent out.

Consider the following scenario:

LPs deposit 1000 USDC (total assets = 1000, shares supply = 1000, share price = 1.00). Two positions each borrow 100 USDC (post-borrows: liquidity = 800, `utilizedAssets` = 200, total assets = 1000). This tracks.

- Position 1 settles with a 200 USDC profit, returnedAmount 400, (amountToVault = 200 USDC total to vault after LP share allocation).

- Actual behavior, based on current code: `repay(200)` increases liquidity to (800 + 200)1000, reduces `utilizedAssets` to (200 - 200), 0, (over-reduction by 100), total assets remain (1000 + 0), share price remains 1.00. Position 2's 100 USDC is now absent.
- Expected behavior: Reduce `utilizedAssets` by only the principal (100) to 100, liquidity = 1000, total assets = 1100, share price == 1.1 Profit for LPs

The above behaviour also occurs if a liquidation occurs, as the liquidation penalty is technically a profit to LPs.

The opposite happens when a loss occurs, the share price should decrease, but it doesn't, which may end up causing liquidity issues/bank run when LPs decide to withdraw.

Recommendations

Reduce `utilizedAssets` only by principal. Bad debt is automatically socialized that way, and profit is automatically made available.

[H-01] Share Inflation Attack via direct asset donation allows stealing user funds

Severity

Impact: High

Likelihood: Medium

Description

The UmVault contract is vulnerable to a "Share Inflation" (or "Donation") attack. This vulnerability allows an attacker to manipulate the exchange rate of shares to be extremely expensive, causing subsequent depositors to lose a significant portion of their funds due to rounding errors in the share calculation.

The issue stems from the deposit function, which calculates the number of shares to mint based on the current totalAssets() of the vault.

The totalAssets() function simply checks the token balance of the contract:

```
function totalAssets() public view virtual returns (uint256) {
        // On-chain USDC + USDC that has been lent out but is still owed
to the vault.
==>        return asset.balanceOf(address(this)) + utilizedAssets;
    }
```

Since balanceOf can be increased by anyone sending tokens directly to the contract (bypassing the deposit function), an attacker can artificially inflate the denominator in the share calculation formula used in deposit():

```
function deposit(uint256 assets) external nonReentrant whenNotPaused
returns (uint256 shares) {
        // ... (checks) ...

        uint256 totalAssetsBefore = totalAssets();
        uint256 supply = totalSupply();

        if (supply == 0 || totalAssetsBefore == 0) {
            // First liquidity: 1:1 mapping.
            shares = assets;
        } else {
            // shares = assets * totalShares / totalAssets
            shares = FixedPointMathLib.fullMulDiv(assets, supply,
totalAssetsBefore); ===>
        }

        // ...
    }
```

Because FixedPointMathLib.fullMulDiv rounds down (truncates), if an attacker inflates totalAssets significantly, the ratio (assets * supply) / totalAssets will suffer from extreme precision loss.

Although the contract includes a check if (shares == 0) revert ZeroShares();, this only protects against 100% loss. It does not protect against significant partial loss (e.g., 25-50%). An attacker can make the inflation such that a victim receives exactly 1 share but pays significantly more than its worth.

Proof of Concept

1. Attacker Setup: Attacker calls deposit(1) (1 wei of USDC). totalSupply = 1. totalAssets = 1. Attacker manually transfers 100 USDC (100*10^6) to the vault address. New totalAssets = 100,000,001. New Share Price: 1 Share approx 100 USDC

2. Victim Deposit: Victim intends to deposit 200 USDC (200,000,000 wei), expecting roughly 2 shares. Contract calculates shares: Shares = 200,000,000 * 1/100,000,001 = 1.9999 Due to Solidity integer division, this rounds down to 1 Share.

3. The Theft: The transaction succeeds (does not revert) because shares (1) is not zero. New Vault Assets: 100,000,001 + 200,000,000 = 300,000,001 USDC. New Total Shares: 1 (Attacker) + 1 (Victim) = 2. New Value per Share: 150 USDC.

Result:

Victim: Deposited 200 USDC, but holds 1 share worth 150 USDC. (Loss: 50 USDC or 25%).

Attacker: Deposited ~100 USDC, holds 1 share worth 150 USDC. (Profit: 50 USDC or 50%).

The attacker has successfully stolen 50 USDC from the victim's deposit instantly.

Recommendations

Implement "Virtual Offsets" (also known as the "Offset" or "Cornerstone" pattern) in the share calculation. This makes the inflation attack prohibitively expensive by artificially increasing the denominator.

Modify the convertToShares logic (or the math inside deposit) to include a virtual offset, similar to the OpenZeppelin ERC4626 implementation:

```
// Add a constant for the offset
    uint256 private constant _DECIMAL_OFFSET = 10 ** 6; // Match asset
decimals

    function _convertToShares(uint256 assets, uint256 supply, uint256
totalAssets_) internal pure returns (uint256) {
        // Formula: assets * (supply + offset) / (totalAssets + 1)
        return FixedPointMathLib.fullMulDiv(
            assets,
            supply + _DECIMAL_OFFSET,
            totalAssets_ + 1
        );
    }
```

Alternatively, implement the "Dead Shares" mechanism by burning the first 1000 shares of the first depositor to the zero address, forcing a minimum initial liquidity cost.

[H-02] Traders Can Frontrun Liquidations to Avoid Penalties

Severity

Impact: High

Likelihood: Medium

Description

Traders/Operator can close their positions, whereas only `liquidatePosition` can only be called by the operator.

```
      function closePosition(uint256 positionId) external ... {
          if (p.status != PositionStatus.OPEN) revert InvalidStatus();
          if (msg.sender != p.trader && msg.sender != operator) revert
NotAuthorizedForClose();
          p.status = PositionStatus.CLOSING;
      }
```

In `settlePosition`, liquidated positions are handled differently from ordinarily closed positions. If liquidated, all of the returned amount goes to the vault.

```
    } else if (returnedAmount < totalCost && isLiquidation) {
            // Liquidate without bad debt, but Vault gets full principal
back + liquidation penalty
            amountToVault = returnedAmount;
            amountToTrader = 0;
        } else if (returnedAmount < totalCost && !isLiquidation) {
            // Position closed as a loss NOT liquidation, but Vault gets
full principal back and trader gets the rest
            amountToVault = borrowed;
            amountToTrader = returnedAmount - borrowed;
        } else {
```

Traders can observe pending liquidation transactions and frontrun them by calling `closePosition`, setting status to CLOSING without triggering liquidation logic.

The frontrunning enables traders to evade liquidation penalties in loss scenarios, where `returnedAmount < totalCost` would otherwise assign all funds to the vault. Instead, non-liquidation closure allows traders to recover `returnedAmount - borrowed`. This undermines risk management, as failing positions avoid forced closure penalties, potentially increasing bad debt socialization to LPs

NB: This works on the assumption that the backend differentiates positions liquidation status based on how it is handled on-chain, rather than some other metric.

Recommendations

Restrict `closePosition` to the operator only.

[M-01] Failed Polymarket Orders Cannot Be Cancelled

Severity

Impact: High

Likelihood: Low

Description

In openPosition, the position status is set as PENDING.

```
p.status = PositionStatus.PENDING;
```

To close/liquidate a position, it's status has to be OPEN

```
function closePosition(uint256 positionId) external ... {
    if (p.status != PositionStatus.OPEN) revert InvalidStatus();
    // ...
}
```

From discussions, a failed polymarket order is retried twice before the position is closed. If the order isn't fulfilled the second time, `recordExecution` is skipped and `closePosition` is called. This fails and as a result, the position remains in PENDING state indefinitely, with funds locked in the trader's wallet. This prevents settlement or closure, as `closePosition` and `liquidatePosition` require OPEN status, and `settlePosition` requires CLOSING.

A way to bypass this is to call `recordExecution` but that depends on the backend's implementation and also comes with its caveats.

Recommendations

Add a `cancelPendingPosition` function callable by the operator, which reverts funds back to the PositionManager, repays any borrowed amount to the vault, and deletes the position record.

[M-02] Sandwich Attacks on Position Settlements

Severity

Impact: Medium

Likelihood: Medium

Description

`settlePosition` can be sandwiched by users, arbitrageurs for quick profit. How long a user has been holding shares in the vault doesn't really matter during withdrawals. Opportunistic users can just front-run profitable settlements (where returnedAmount > totalCost, or during liquidations leading to LP profit share increasing vault totalAssets), acquiring shares at the pre-profit price and redeeming at the post-profit price for risk-free gains. This dilutes existing LPs' yields, and undermines protocol's economic participation as the attacker extracts value without contributing liquidity.

Recommendations

Implement some sort of cooldown/time tracking mechanism to handle deposits/withdrawals. Maybe requiring deposit for a certain time period before tokens can be withdrawn.

[L-01] Incorrect assets are returned in the convert to assets function

Description

When there is no supply of tokens then the convert to assets function returns assets equal to the shares passed in as input to the function which is wrong because it should return 0.

```
 function convertToAssets(uint256 shares) public view returns (uint256) {
        uint256 supply = totalSupply();
        if (supply == 0) {
            return shares;
        }

        return FixedPointMathLib.fullMulDiv(shares, totalAssets(),
supply);
    }
```

Recommendations

Return zero when supply == 0.

[L-02] Protocol Allows 1x Leverage

Description

The contract permits leverage of 1, resulting in zero borrowed funds, yet processes the position as if leveraged, exposing users to liquidation penalties and profit sharing even though, they technically didn't borrow anything.

```
    function openPosition(
        address trader,
        uint256 collateralAmount,
        uint256 leverage,
        string calldata marketId,
        bool isLong
    ) external onlyOperator whenNotPaused nonReentrant returns (uint256
positionId) {
        if (trader == address(0)) revert InvalidTrader();
        if (collateralAmount == 0) revert InvalidCollateral();
        if (leverage < 1) revert InvalidLeverage(); // at least 1x //
<<==1
        // ... (abbreviated for brevity)
        uint256 borrowed = collateralAmount * (leverage - 1); //<<==2
        // ... transfers collateral + borrowed (0 for 1x) to trader
    }
```

And in `settlePosition`

```
        } else if (returnedAmount < totalCost && isLiquidation) {
            // Liquidate without bad debt, but Vault gets full principal
back + liquidation penalty
            amountToVault = returnedAmount;//<<==3
            amountToTrader = 0;
        } else if (returnedAmount < totalCost && !isLiquidation) {
            // Position closed as a loss NOT liquidation, but Vault gets
full principal back and trader gets the rest
            amountToVault = borrowed;
            amountToTrader = returnedAmount - borrowed;
        } else {
            // Position won: LPs share in profit
            uint256 netProfit = returnedAmount - totalCost;//<<==4

            uint256 lpShare = (netProfit * lpProfitShareBps) /
BPS_DENOMINATOR;
            uint256 traderShare = netProfit - lpShare;

            amountToVault = borrowed + lpShare;
            amountToTrader = collateralAmount + traderShare;
        }
```

Allowing 1x leverage permits creation of positions without vault borrowing, yet subjects them to full lifecycle. In a loss scenario (returnedAmount < collateral), a liquidated position assigns all returned funds to the vault, causing loss of funds to the trader. If the position makes profit, the profit is shared despite the protocol "technically" not having any claim to the profit.

Recommendations

Revert if leverage <= 1 instead.

[L-03] Full Loss Positions Cannot Be Settled

Description

The `settlePosition` function rejects `returnedAmount = 0`, making it impossible to settle positions where the trader loses everything.

```

    function settlePosition(uint256 positionId, uint256 exitPrice, uint256
returnedAmount, bool isLiquidation)
        external onlyOperator whenNotPaused nonReentrant
validPositionId(positionId)
    {
        if (returnedAmount == 0) revert InvalidReturnedAmount(); //<<==
        // ...
    }
```

Total loss positions remain in `CLOSING` state forever, while losses cannot be properly recorded and socialized. If the operator backend is automated, it may continue to retry settling these positions, potentially causing a denial of service error.

Recommendations

Incorporate a special logic of 0 returned amount. Such positions may be written off directly as bad debt.

[L-04] APR Can be easily Inflated

Description

Anyone can call `snapshotYield` just before repayments are made to minimize `elapsed` and potentially inflate reported APR.

```
function snapshotYield() external {
    _snapshotYieldInternal(); // Can be called immediately after profit
}
```

`_snapshotYieldInternal` calls `_aprSinceLastSnapshot` which relies on the elapsed time between the time of the last call and the current time. It is also called in `repay`

```
function _aprSinceLastSnapshot() internal view ... {
    uint256 elapsed = block.timestamp - t0;
    if (elapsed > SECONDS_PER_YEAR) elapsed = SECONDS_PER_YEAR;
    // ... APR = gain * SECONDS_PER_YEAR / elapsed
}
```

As a result, attackers can frontrun repay by calling snapshotYield, creating a tiny elapsed period and exaggerating APR (e.g., 1% gain over 1 second annualizes to extreme values). This misleads users relying on `currentAprWad` for decisions, potentially attracting deposits at inflated yields. Temporary spikes distort historical metrics via YieldSnapshot events, affecting integrations or analytics.

Recommendations

Restrict `snapshotYield` to owner or positionManager, or add a minimum elapsed time check before allowing snapshots.

[L-05] Manipulation of `_entryTimestamp` via Share Transfers

Description

When users deposit, their previous share balance is obtained, which is used to set the initial `_entryTimestamp`. But via transfers to msg.sender, the previous balance becomes > 0 and, `_entryTimestamp` is now calculated, which gives a timestamp much earlier than current time.

```
if (prevShareBalance == 0) {
    _entryTimestamp[msg.sender] = nowTs;
} else {
    uint256 weighted = uint256(_entryTimestamp[msg.sender]) *
prevShareBalance + uint256(nowTs) * shares;
    _entryTimestamp[msg.sender] = uint64(weighted / newBalance);
}
```

The same can be seen in withdrawals in which `_entryTimestamp` is deleted if a user is fully withdrawing. But users can just withdraw all but 1 wei, to still maintain the `_entryTimestamp`.

```
if (shares == prevShareBalance) {
    delete _entryTimestamp[msg.sender];
}
```

Users can transfer dust shares from old accounts to new ones, then deposit large amounts, weighting the entryTimestamp toward the older time and artificially inflating ageMultiplier (up to 1.3x). This boosts `ageOf` and `effectiveBalanceOf`, potentially skewing any dependent features (e.g., if used for rewards or voting). System-wide, sybil attacks could distribute old shares across multiple accounts, diluting legitimate users' multipliers and leading to unfair resource allocation.

Recommendations

Do not track timestamp with the previous balance, check if the timestamp had been updated before. i.e if `_entryTimestamp[msg.sender] = 0` set it to `nowTs`. Also, if a user is not fully withdrawing, they must maintain a substantial minimum amount.

[L-06] Deposit/Withdraw Slippage Missing Protection

Description

Both `deposit` and `withdraw` functions lack minimum/maximum output parameters, leaving users vulnerable to unfavorable price movements during high volatility periods.

```
function deposit(uint256 assets) external nonReentrant whenNotPaused
returns (uint256 shares) {
    // No minimumShares parameter for slippage protection
    shares = FixedPointMathLib.fullMulDiv(assets, supply,
totalAssetsBefore);
}
```

```
function withdraw(uint256 shares) external nonReentrant whenNotPaused
returns (uint256 assets) {
    // No minimumAssets parameter for slippage protection
    assets = FixedPointMathLib.fullMulDiv(shares, totalAssets(), supply);
}
```

Users can be frontrun by other users manipulating share price (e.g., via sandwiches around profits or losses), receiving fewer shares on deposit or fewer assets on withdrawal than expected. In volatile conditions, this leads to direct fund loss, as conversions use pre-transaction totalAssets/supply.

Recommendations

Add `minShares` and `minAssets` parameters with slippage tolerance.

[I-01] `borrow` and `repay` emits msg.sender instead of trader

Description

`borrow` and `repay` are called by the position manager. But the borrower and repayer is the trader.

```
    function borrow(uint256 amount) external nonReentrant whenNotPaused
onlyPositionManager returns (bool) {
//...
        emit Borrow(msg.sender, amount);
        return true;
    }
```

```
    function repay(uint256 amount) external nonReentrant
onlyPositionManager {
//...
        emit Repay(msg.sender, amount);
//...
```

Recommendations

Pass in and emit the trader instead.

[I-02] Use `memory` rather than `storage` for queries

Description

`openPosition` queries market config and doesn't make any update to the config.

```
//...

        // Load config directly, no intermediate `key`
        MarketConfig storage cfg =
marketConfigs[_marketKey(marketId)];//<<===

        if (!cfg.isApproved) revert MarketNotApproved();
        if (cfg.maxLeverage != 0 && leverage > uint256(cfg.maxLeverage))
revert LeverageTooHigh();
```

Recommendations

It's more gas efficient to use `memory` for queries than `storage`