Ulti Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Shaka, ast3ros

November 11th - November 16th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **ulti-org/ulti-protocol-contract** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ulti

ULTI is a DeFi protocol where users can deposit the native currency of a blockchain, such as ETH on the Ethereum Mainnet, in exchange for ULTI tokens. Ulti launches its token by creating a liquidity pool on Uniswap while handling initial token distribution. Users can deposit input tokens to earn ULTI tokens, with systems in place to claim rewards and manage referrals.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 2024dbadc12f6c4efea1d5ec29e6d994c6bc6636

fixes review commit hash - f603232cdb5f06bea18ccf2c69f4479d71449871

Scope

The following smart contracts were in scope of the audit:

- `ULTI`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Shaka, ast3ros engaged with Ulti to review Ulti. In this period of time a total of **25** issues were uncovered.

Protocol Summary

Protocol Name	Ulti
Repository	https://github.com/ulti-org/ulti-protocol-contract
Date	November 11th - November 16th
Protocol Type	Farming protocol

Findings Count

Severity	**Amount**
Critical	2
High	3
Medium	8
Low	12
Total Findings	**25**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Critical precision loss in slippage protection calculation	Critical	Resolved
[C-02]	The Lowest contributor of the top contributor's list can be removed unfairly	Critical	Resolved
[H-01]	Wrong active pumper bonus calculation	High	Resolved
[H-02]	DoS on the pump function due to excessively low slippage tolerance	High	Resolved
[H-03]	DoS on deposits during the early bird period	High	Resolved
[M-01]	(Out of scope) Missing unchecked block in TickMath library	Medium	Resolved
[M-02]	Potential overflow risk due to not using mulDiv in _getSpotPrice()	Medium	Resolved
[M-03]	Referral bonus cap will be lower than what was intended	Medium	Resolved
[M-04]	Wrong rounding of TWAP when tickCumulativeDelta is negative	Medium	Resolved
[M-05]	Referrer's contributions are not updated correctly for non-native deposits	Medium	Resolved
[M-06]	Wrong assumption that the input token will be WETH	Medium	Resolved
[M-07]	DoS on the deposit function due to excessively low slippage tolerance	Medium	Resolved

[M-08]	TWAP can be manipulated	Medium	Resolved
[L-01]	Rounding error in early bird deposit	Low	Resolved
[L-02]	Contract incompatibility with non-standard ERC20 tokens like USDT	Low	Resolved
[L-03]	Using spot price inside _calculateTWAP() makes code vulnerable to price manipulations	Low	Resolved
[L-04]	Attacker can create circular referrals with lengths of three and higher	Low	Acknowledged
[L-05]	User specified slippage check for pump/swap is inefficient	Low	Resolved
[L-06]	Incorrect calculation in getGlobalData	Low	Resolved
[L-07]	No support for fee-on-transfer input token	Low	Acknowledged
[L-08]	getUserData returns the wrong streak input upper boundary for cycle 1	Low	Resolved
[L-09]	DoS vector in the launch function	Low	Acknowledged
[L-10]	Native tokens received by the receive and fallback can be locked	Low	Resolved
[L-11]	Potential overflow in _getSpotPrice() during price calculation	Low	Resolved
[L-12]	Losing some part of the streak bonus if depositing in multiple transactions	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Critical precision loss in slippage protection calculation

Severity

Impact: High

Likelihood: High

Description

The ULTI contract contains a precision error in its slippage protection mechanism during token swaps (using 18 instead of 1e18). In the `_swapInputTokenForUlti` function, the calculation of `expectedUltiAmountWithoutSlippage` uses an incorrect scaling factor that results in a significant precision loss.

```
function _swapInputTokenForUlti(
        uint256inputAmountToSwap,
        uint256minUltiAmount,
        uint256twap,
        uint256deadline
    )
        private
        returns (uint256 ultiAmount)
    {
        ...
        // 2. Calculate expected output without slippage
        uint256 expectedUltiAmountWithoutSlippage = inputAmountToSwap * 18 /
        // twap; // @audit should be inputAmountToSwap * 1e18 / twap

        // 3. Choose the higher minimum amount between user-specified and
        // internal slippage protection
        uint256 minUltiAmountInternal = (expectedUltiAmountWithoutSlippage *
          (10000 - MAX_SWAP_SLIPPAGE_BPS)) / 10000;

                uint256 effectiveMinUltiAmount = minUltiAmount > minUltiAmountInterna
        ...

        return ultiAmount;
    }
```

The error leads to:

- Precision Loss: Using 18 instead of 1e18 as the scaling factor causes `expectedUltiAmountWithoutSlippage` to be drastically undervalued or rounded down to 0 for most practical input amounts.
- Broken Slippage Protection: When `expectedUltiAmountWithoutSlippage` is 0, `minUltiAmountInternal` also becomes 0, effectively disabling the contract's internal slippage protection.

Malicious actors can:

- Set a very low minUltiAmount (e.g., 1)
- Manipulate the ULTI/Input token price through market operations
- Execute swaps with extreme slippage
- Extract value from the protocol's reserves

Recommendations

Fix the calculation of `expectedUltiAmountWithoutSlippage` by multiplying `inputAmountToSwap` by `1e18` instead of `18`.

[C-02] The Lowest contributor of the top contributor's list can be removed unfairly

Severity

Impact: High

Likelihood: High

Description

When a new deposit is made the list of top contributors is updated in the `_updateTopContributors` function. If the list is full (33 contributors), the lowest contributor is searched. If the contribution of the current contributor is greater than the one of the lowest contributor, the lowest contributor is removed and the current contributor is added to the list.

```
            // Replace lowest contributor if new contribution is higher
            if (updatedContribution > minContribution) {
                _topContributors.remove(minContributor);
                _topContributors.set
  (contributorAddress, updatedContribution);
            }
```

However, it is not checked if the current contributor is already on the list. So, if the current contributor is already on the list, the lowest contributor is unfairly removed, leaving the list with 32 contributors.

This has several implications:

- The lowest contributor can be DoSed for the `pump` function.
- The lowest contributor can be removed from the rewards distribution for top distributors.
- It can be used by a contributor present in the list to occupy another position by removing the lowest contributor and using a different account to make a deposit for the minimum amount allowed, even if there were many other contributors with higher contributions.

Recommendations

```
-           // Replace lowest contributor if new contribution is higher
+
+            // Add or update contributorAddress contributor if is higher than minCont
            if (updatedContribution > minContribution) {
-               _topContributors.remove(minContributor);
-               _topContributors.set(contributorAddress, updatedContribution);
+               if (_topContributors.set
+ (contributorAddress, updatedContribution)) {
+                   // Remove minContributor if contributorAddress is added
+                   _topContributors.remove(minContributor);
+               }
            }
```

8.2. High Findings

[H-01] Wrong active pumper bonus calculation

Severity

Impact: Medium

Likelihood: High

Description

According to the whitepaper, Active Pumpers receive a +3% of their Guardian bonus.

However, the current implementation results in a significantly lower bonus of 0.03% due to incorrect scaling in the calculation:

- ACTIVE_PUMPERS_BONUS_PERCENTAGE = ULTI_NUMBER / 10 = 3.
- So the bonus is 3 / 10000 = 0.03%.

Active Pumpers receive only a 0.03% additional bonus instead of the intended 3%, this represents a 99% reduction in the promised bonus. It's a significant loss of value for active pumpers.

```
function _allocateTopContributorsBonuses(uint32 cycle) private {
        ...
            // Apply active pumper bonus (+3.3%) if they qualify
            if (_isActivePumper(cycle, contributor) && bonus > 0) {
                bonus = bonus *
                //(10000 + ACTIVE_PUMPERS_BONUS_PERCENTAGE) / 10000; // @audit using 1
            }

        ...
    }
```

Recommendations

Fix the bonus calculation:

```
if (_isActivePumper(cycle, contributor) && bonus > 0) {
-                   bonus = bonus *
- (10000 + ACTIVE_PUMPERS_BONUS_PERCENTAGE) / 10000;
+                   bonus = bonus * (100 + ACTIVE_PUMPERS_BONUS_PERCENTAGE) / 100;
               }
```

[H-02] DoS on the `pump` function due to excessively low slippage tolerance

Severity

Impact: High

Likelihood: Medium

Description

In the pump process, a fraction of the input tokens held by the ULTI contract are swapped for ULTI tokens. The system calculates the expected amount of ULTI tokens based on the TWAP and applies a slippage protection that causes the swap to fail if a slippage greater than `MAX_SWAP_SLIPPAGE_BPS` is detected.

```
     /// @notice Maximum allowed slippage for swaps in basis points (0.99%)
     uint256 public constant MAX_SWAP_SLIPPAGE_BPS = 3 * ULTI_NUMBER;
    (...)
          // 3. Choose the higher minimum amount between user-specified and
// internal slippage protection
          uint256 minUltiAmountInternal =
  (expectedUltiAmountWithoutSlippage * (10000 - MAX_SWAP_SLIPPAGE_BPS)) / 10000;

          uint256 effectiveMinUltiAmount = minUltiAmount > minUltiAmountInternal ? mi

```

The current value for `MAX_SWAP_SLIPPAGE_BPS` is 99 basis points (0.99%). Taking into account that the pool fee is 1%, even in the best case scenario (TWAP equal to spot price and small amount of input tokens sent), the swap will fail.

As a result, the pump process can not be executed. Given that this is a fundamental part of the mechanics of the ULTI protocol, this issue is considered high severity.

Note: There is another issue related to the `_swapInputTokenForUlti` function that causes the expected amount of ULTI tokens to be calculated incorrectly (they are multiplied by 18 instead of 1e18). This heavily underestimates the number of ULTI tokens that should be received, which prevents this issue from causing the DoS of the `pump` function. However, once the other issue is fixed, the effects of this issue will cause the DoS of the `pump` function. For this reason, the likelihood of this issue is considered medium.

Recommendations

Allow a maximum slippage percentage that is high enough to cover:

- Pool fee (1%).
- A reasonable divergence of the TWAP from the spot price.
- The slippage due to the size of the swap (the higher the more input tokens are held by the ULTI contract and the lower the liquidity of the pool).
- Potential protocol fees in the pool (Uniswap might enable fees in the future).

[H-03] DoS on deposits during the early bird period

Severity

Impact: Medium

Likelihood: High

Description

On deposits, the deposit price is calculated using the TWAP, but in the first 24 hours after the launch, a fixed price is used instead. This fixed price is the initial price in the pool.

```
        if (block.timestamp < launchTimestamp + EARLY_BIRD_PRICE_DURATION) {
            depositPrice = 1e18 / INITIAL_RATIO;
        }
```

A portion of the input tokens deposited by the user, together with an equivalent amount of ULTI tokens, are added to the pool as liquidity. The amount of ULTI tokens is calculated using the mentioned deposit price.

```
        inputTokenForLP = (inputTokenAmount * LP_CONTRIBUTION_PERCENTAGE) / 100;
        ultiForLP = inputTokenForLP * 1e18 / depositPrice;
```

When the two tokens are added as liquidity, the contract ensures that the amounts provided are not reduced to more than `MAX_ADD_LP_SLIPPAGE_BPS` basis points with respect to the desired amounts. This means that a difference between the deposit price and the spot price greater than `MAX_ADD_LP_SLIPPAGE_BPS` basis points will revert the transaction. Given that the deposit price is fixed in the first 24 hours, any change in the spot price greater than `MAX_ADD_LP_SLIPPAGE_BPS` will cause all deposits to revert until the early bird period ends.

```
        uint256 minUltiAmount = (ultiForLP * (10000 - MAX_ADD_LP_SLIPPAGE_BPS)) / 10000;
        uint256 minInputTokenAmount = (inputTokenForLP * (10000 - MAX_ADD_LP_SLIPPAGE_BPS)) / 10000;
```

This issue has some important implications:

- Given the relevance of the initial adoption to the project, locking the main functionality of the contract for the first 24 hours might significantly diminish the engagement of the users.
- A malicious actor could make a deposit and afterward swap the pool to block any other users from depositing, being the only one able to benefit from the early bird deposit price and selling those tokens at a higher price once the early bird period ends.

Proof of Concept

```
function test_DoSDepositsAfterLaunch() public {
    // Swap to move spot price
    uint256 wethIn = 0.06 ether;

        tokenIn: address(weth),
        tokenOut: address(ulti),
        fee: 10_000,
        recipient: address(this),
        deadline: block.timestamp,
        amountIn: wethIn,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    });
    v3Router.exactInputSingle(swapParams);

    // Deposit on ULTI reverts due to price slippage
    vm.expectRevert("Price slippage check");
    ulti.depositNative{value: 0.00001 ether}(address
      (0), 0, block.timestamp, false);
}
```

Recommendations

During the early bird period, use the spot price to calculate the amount of ULTI tokens to be added to the pool as liquidity.

8.3. Medium Findings

[M-01] (Out of scope) Missing unchecked block in TickMath library

Severity

Impact: High

Likelihood: Low

Description

Uniswap math libraries rely on wrapping behavior for conducting arithmetic operations. Solidity version 0.8.0 introduced checked arithmetic by default where operations that cause an overflow would revert.

Since the `lib/uniswap/TickMath.sol` library was adapted from Uniswap and written in Solidity version 0.8.24, these arithmetic operations should be wrapped in an unchecked block.

Recommendations

In `getSqrtRatioAtTick` and `getTickAtSqrtRatio`, wrap the arithmetic operations in unchecked blocks to maintain compatibility with Uniswap math libraries.

[M-02] Potential overflow risk due to not using `mulDiv` in `_getSpotPrice()`

Severity

Impact: High

Likelihood: Low

Description

When ULTI is `token0` and the price increases significantly:

- `priceX192` could exceed `2^256 / 1e18`
- The multiplication `priceX192 * 1e18` would overflow before the right shift
- This causes the transaction to revert.

The overflow would cause the transaction to revert if ULTI is token 0, preventing:

- Deposits that check spot price
- Pumps that rely on price calculations
- Any other functions using `_getSpotPrice()`

The error is not using safe arithmetic operations like Uniswap's `FullMath.mulDiv()`, `FullMath.mulDiv()` allow intermediate overflow while preserving precision.

```
function _getSpotPrice() internal view returns (uint256 spotPrice) {
        ...

        if (isUltiToken0) {
            // 3. Calculate price as input token amount needed per 1 ULTI
            spotPrice = priceX192 * 1e18 >> 192; // @audit potential overflow if
            // priceX192 > 2^256 / 1e18
        } else {
            // 4. Calculate inverse price ratio for consistent pricing format
            spotPrice = (1 << 192) * 1e18 / priceX192;
        }
    }
```

Recommendations

In Uniswap v3 periphery library, FullMath.mulDiv is used to allow overflow of an intermediate value without any loss of precision, and allows multiplication and division where an intermediate value overflows 256 bits ("phantom overflow").

It's recommended to use `FullMath.mulDiv` to calculate the spot price to prevent overflow and ensure accurate price calculations:

```
if (isUltiToken0) {
-            spotPrice = priceX192 * 1e18 >> 192;
+            spotPrice = FullMath.mulDiv(1e18, priceX192, 1 << 192);
        ...
```

[M-03] Referral bonus cap will be lower than what was intended

Severity

Impact: Low

Likelihood: High

Description

According to the documentation and the comments inside the code, the maximum bonus a referrer can accumulate is 10 times the total amount of ULTI they have accumulated from direct deposits since the start. Code ensures this max cap by keeping track of unclaimed bonuses in `claimableBonuses[]` and by checks that `claimableBonuses[]` doesn't go higher than the max cap:

```
// 3a. Cap referral bonus based on skin-in-game limit
            // Note: if the cap is reached, no referrer and referred bonuses
            // will be acculmulated. This is intentional
            uint256 skinInAGameCap = _getSkinInTheGameCap(effectiveReferrer);

                          uint256 remainingBonusAllowance = skinInAGameCap > claimableB
                ? skinInAGameCap - claimableBonuses[effectiveReferrer]
                : 0;

                          referrerBonus = referrerBonus > remainingBonusAllowance ? rem
```

The issue is that `claimableBonuses[]` contains other bonus types too. So if a user receives a big amount of streak/referred/top contributor bonuses then they can't receive referrer bonuses up to the cap that is defined because `claimableBonuses[]` contains those other bonus types. This would discourage the top contributors and users with high streaks from making more referrals.

Recommendations

Keep track of referral bonuses in other variables and ensure the max cap based on that variable.

[M-04] Wrong rounding of TWAP when `tickCumulativeDelta` is negative

Severity

Impact: Low

Likelihood: High

Description

For the TWAP calculation, `tickCumulativeDelta` is calculated as the difference between the two tickCumulatives. This value is divided by the first tick cumulative to obtain the time-weighted average tick.

Given that Solidity division truncates the result, the effect of this truncation is different depending on the sign of `tickCumulativeDelta`. If `tickCumulativeDelta` is positive, a truncation will decrease the value of the tick. However, if `tickCumulativeDelta` is negative, truncating its value increases the value of the tick.

```
        try liquidityPool.observe(secondsAgos) returns
  (int56[] memory tickCumulatives, uint160[] memory) {
            // 3a. Calculate time-weighted average tick

            int56 tickCumulativeDelta = tickCumulatives[1] - tickCumulatives[0];
            int24 timeWeightedAverageTick = int24
  (tickCumulativeDelta / int32(secondsAgos[0]));

            // 3b. Convert tick to price quote
            // Always pass ULTI as base token
//(amount of 1e18 = 1 ULTI) and input token as quote token
            // This ensures we get the price in INPUT_TOKEN/ULTI format
// consistently
            twap = OracleLibrary.getQuoteAtTick(
                timeWeightedAverageTick,
                1e18, // amountIn: 1 ULTI token (18 decimals)
                address(this), // base token (ULTI)
                inputTokenAddress // quote token (INPUT_TOKEN)
            );
```

The current implementation does not adjust the TWAP value when `tickCumulativeDelta` is negative, which creates a discrepancy in the calculated TWAP value depending on the sign of `tickCumulativeDelta`.

Recommendations

```
int56 tickCumulativeDelta = tickCumulatives[1] - tickCumulatives[0];
        int24 timeWeightedAverageTick = int24(tickCumulativeDelta / int32
          (secondsAgos[0]));
+       if (tickCumulativesDelta < 0 && (tickCumulativesDelta % int56(uint56
+ (MIN_TWAP_TIME)) != 0)) {
+           timeWeightedAverageTick--;
+       }
```

[M-05] Referrer's contributions are not updated correctly for non-native deposits

Severity

Impact: Low

Likelihood: High

Description

On deposits, the `_updateReferrals` function processes the referral bonuses. In step 3 of the process, the `_updateContributors` function is called to update the referrer's contributions.

```
1661:                    _updateContributors
  (cycle, effectiveReferrer, 0, msg.value, referrerBonus);
```

The fourth argument of the `_updateContributors` function is the `inputTokenReferred` which is the amount of tokens deposited by the referred user. However, the `msg.value` is passed as the argument, which will be zero if the deposit is made using the `deposit` function and not the `depositNative` function.

As a result, the `totalInputTokenReferred` mapping will not be updated for the referrer. While this value is currently not used in the contract, it is exposed through view functions, so it could be used to calculate the distribution of additional rewards via external implementations. In this regard, it is also worth noting that the whitepaper mentions a "reputation system based on contribution (deposits, referrals, streaks)" for future implementations.

Recommendations

```
// 7. Calculate referral bonus and allocate it based on the ULTI just
        // allocated to the user including the streak bonus
        (
          addresseffectiveReferrer,
          uint256referrerBonus,
          uint256referredBonus
        ) =
-           _updateReferrals(referrer, ultiForUserWithStreakBonus, cycle);
+           _updateReferrals
+ (referrer, inputTokenAmount, ultiForUserWithStreakBonus, cycle);
    (...)
-   function _updateReferrals
- (address referrer, uint256 ultiToMint, uint32 cycle)
+   function _updateReferrals
+ (address referrer, uint256 inputTokenReferred, uint256 ultiToMint, uint32 cycle)
        private
        returns (
          addresseffectiveReferrer,
          uint256referrerBonus,
          uint256referredBonus
        )
    (...)
                // 3c. Update referrer's allocated bonuses and contributions
                claimableBonuses[effectiveReferrer] += referrerBonus;
-               _updateContributors
- (cycle, effectiveReferrer, 0, msg.value, referrerBonus);
+               _updateContributors
+ (cycle, effectiveReferrer, 0, inputTokenReferred, referrerBonus);
```

[M-06] Wrong assumption that the input token will be WETH

Severity

Impact: High

Likelihood: Low

Description

ULTI accepts any ERC20 token as the input token and in the constructor, the input token number of decimals is stored in the `inputTokenDecimals` variable.

```
    /// @notice Address of the input token (e.g. WETH, DAI, etc.)
    address public immutable inputTokenAddress;
    (...)
        inputTokenDecimals = IERC20(_inputTokenAddress).decimals();
```

However, this value is never used and some constants and calculations are based on the assumption that the input token will be WETH, using the expected value of ETH for calculations.

```
    // Deposit and economic constants
    /// @notice Minimum amount required for a deposit to be valid //(0.000001 ETH)
    /// @dev Economic considerations for minimum deposit:
    /// - At ETH=$1,000,000: minimum = $1 worth of ETH (0.000001 ETH)
    /// - Small enough to allow $1 deposits even if the price of ETH goes // to $1M per ETH
    /// - Prevents streak bonus gaming while maintaining accessibility
    uint256 public constant MINIMUM_DEPOSIT_AMOUNT = 0.000001 ether;

    /// @notice Initial ratio ULTI:ETH: 330000:1 Amount of ULTI for 1 ETH.
    uint256 public constant INITIAL_RATIO = ULTI_NUMBER * 1e4;
  (...)
        if (inputTokenAmount < ULTI_NUMBER * 1e18) revert LaunchMinimumFounderGiveawayRequired(
```

As a result, if the input token is not WETH, and especially if it does not have 18 decimals, the contract will not work as expected and may lead to unexpected behavior.

For example, using USDC (6 decimals) as the input token would require the owner to deposit 33 trillion USDC to launch the contract, which is not feasible.

On the other hand, if WPOL (Polygon's native token wrapper) was used as the input token, having the POL token a much lower value than ETH, the value of the ULTI token in USD terms would be much lower than expected.

Recommendations

Make the `MINIMUM_DEPOSIT_AMOUNT`, `INITIAL_RATIO`, and minimum deposit amount configurable on contract creation and use `10 ** inputTokenDecimals` instead of the hardcoded value of 1e18 for calculations involving the input token.

[M-07] DoS on the `deposit` function due to excessively low slippage tolerance

Severity

Impact: Medium

Likelihood: Medium

Description

On deposits, part of the input tokens and newly minted ULTI tokens are added to the pool as liquidity. As a slippage protection mechanism, adding liquidity reverts when the difference between the TWAP and the spot price is greater than `MAX_ADD_LP_SLIPPAGE_BPS` basis points.

The value of this constant is set to 33 basis points. This means that for very small variations in the spot price, the transaction will revert, not allowing users to deposit input tokens.

Proof of Concept

```
function test_depositSlippage() public {
    // Increase pool observation cardinality
    v3Pool.increaseObservationCardinalityNext(90);

    // Skip early bird period
    skip(1 days);

    // Simulate a swap every new block for 90 blocks
    for (uint256 i = 0; i < 90; i++) {

            tokenIn: address(weth),
            tokenOut: address(ulti),
            fee: 10_000,
            recipient: address(this),
            deadline: block.timestamp,
            amountIn: 1,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });
        v3Router.exactInputSingle(swapParams);
        skip(12 seconds);
        vm.roll(block.timestamp + 1);
    }

    // Create a swap that moves the price a small amount

        tokenIn: address(weth),
        tokenOut: address(ulti),
        fee: 10_000,
        recipient: address(this),
        deadline: block.timestamp,
        amountIn: 0.06 ether,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    });
    v3Router.exactInputSingle(swapParams);
    skip(12 seconds);
    vm.roll(block.timestamp + 1);

    // Deposit on ULTI reverts due to price slippage
    vm.expectRevert("Price slippage check");
    ulti.depositNative{value: 0.00001 ether}(address
      (0), 0, block.timestamp, false);
}
```

Recommendations

Increase the value of `MAX_ADD_LP_SLIPPAGE_BPS` to a more reasonable value which accounts for the price volatility of the price in the pool.

[M-08] TWAP can be manipulated

Severity

Impact: Medium

Likelihood: Medium

Description

The contract uses the TWAP oracle to calculate the price for deposits and swaps performed through the pump mechanism. The observation window is set to 18.15 minutes.

However, two issues cause the TWAP to be easily manipulated:

1. Uniswap V3 pools are initialized with an observation cardinality of 1, which means that the TWAP oracle will only consider the last price update.
2. Even if the observation cardinality is high enough for the observation window, the number of observations recorded might not be enough.

This means that an attacker could manipulate the price of the ULTI token. Given that both the `deposit` and the `pump` functions have slippage protection mechanisms, the outcome of the attack would be the DoS for these functions.

Recommendations

1. Call the `IUniswapV3Pool.increaseObservationCardinalityNext(uint16 observationCardinalityNext)` function after the initialization of the pool with a value high enough to cover the observation window. This value should be at least `observationWindow / averageBlockProductionRageOfTheNetwork`.
2. Ensure the number of observations recorded is high enough to cover the observation window.

8.4. Low Findings

[L-01] Rounding error in early bird deposit

During the early bird period, users may receive slightly more ULTI tokens than intended due to rounding errors in the price calculation.

The issue occurs in the `_allocateDeposit` function when calculating ULTI token amounts during the early bird period. The function performs two separate divisions that can lead to rounding errors:

```
function _allocateDeposit
      (uint256 inputTokenAmount, uint256 minUltiToAllocate, uint256 deadline)
        private
        returns
          (uint256 ultiForUser, uint256 ultiForLP, uint256 inputTokenForLP)
    {
        ...
        if (block.timestamp < launchTimestamp + EARLY_BIRD_PRICE_DURATION) {
            depositPrice = 1e18 / INITIAL_RATIO; // @audit First division -
            // slightly smaller deposit price due to integer division rounding down
        }

        ultiForUser = inputTokenAmount * 1e18 / depositPrice; // @audit Second
        // division - resulting in slightly more ULTI tokens being minted
        ...
        // 5. Calculate INPUT and ULTI portions for liquidity pool
        inputTokenForLP = (inputTokenAmount * LP_CONTRIBUTION_PERCENTAGE) / 100;
        ultiForLP = inputTokenForLP * 1e18 / depositPrice;
        ...
    }
```

It's recommended to directly multiply by `INITIAL_RATIO` to avoid rounding errors

```
ultiForUser = inputTokenAmount * INITIAL_RATIO
    ultiForLP = inputTokenForLP * INITIAL_RATIO
```

[L-02] Contract incompatibility with non-standard ERC20 tokens like USDT

The ULTI contract uses `IERC20.approve` and `IERC20.transferFrom` to interact with input tokens. However, there are some tokens that don’t follow the

IERC20 interface, such as USDT. This could lead to transaction reverts when deploying a contract, launching a new pool, or depositing tokens.

It's recommended to use functions such as forceApprove and safeTransferFrom from the SafeERC20 library from OpenZeppelin to interact with input tokens.

[L-03] Using spot price inside `_calculateTWAP()` makes code vulnerable to price manipulations

Code uses `_calculateTWAP()` to perform slippage checks and prevent malicious users from extracting values by manipulating the liquidity pool and interacting with the ULTI token. The issue is that `_calculateTWAP()` uses spot price when it can't calculate the TWAP price and this would make it vulnerable to price manipulation attacks. An attacker can manipulate the pool's price and then deposit tokens it would cause ULTI to provide liquidity while the spot price is wrong and the attacker can then extract value from the liquidity pool.

[L-04] Attacker can create circular referrals with lengths of three and higher

Code prevents circular referrals of length 1 and 2 but it's still possible to create bigger loop referrals with bigger length and game the system.

```
if (referrer == msg.sender) revert DepositCannotReferSelf();
        if (referrers[referrer] == msg.sender) revert DepositCircularReferral();
```

[L-05] User specified slippage check for pump/swap is inefficient

When a top contributor wants to pump the ULTI token they specify the `minUltiAmount` to prevent slippage when swapping the input token for the ULTI token. The issue is that the contract input token amount may increase while the user's pump transaction is inside the mempool and received ULTI tokens from the swap can be higher than `minUltiAmount` while the swap has bad slippage because the code would use more input token as input token amounts are increased. To limit the slippage it's better for a user to specify the price and the code should calculate `minUltiAmount` based on the user-specified price and the current contract's input token balance.

[L-06] Incorrect calculation in `getGlobalData`

The `getGlobalData` function exposes data from the contract, including the amounts of input tokens and ULTI tokens currently in the liquidity pool.

```
1972:
        * @param inputTokenInLP Amount of input token currently in Uniswap V3 liquidit
1973:
        * @param ultiInLP Amount of ULTI currently in Uniswap V3 liquidity pool
```

The calculation of these values is done in the `_getLiquidityAmounts` function:

```
786:     function _getLiquidityAmounts() internal view returns
  (uint256 inputTokenAmount, uint256 ultiAmount) {
787:         uint128 liquidity = liquidityPool.liquidity();
788:         (uint160 sqrtPriceX96,,,,,,) = liquidityPool.slot0();
789:
790:         // Get sqrt price at the boundaries of our position
791:         uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(LP_MIN_TICK);
792:         uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(LP_MAX_TICK);
793:
794:         (uint256 amount0, uint256 amount1) =
795:             LiquidityAmounts.getAmountsForLiquidity
  (sqrtPriceX96, sqrtRatioAX96, sqrtRatioBX96, liquidity);
```

However, the value obtained in line 787 represents the currently in-range liquidity available to the pool, so the liquidity of all positions currently not in range is not included in the calculation. As a result, the value returned by the `getGlobalData` function is not accurate.

If the intention is to return the amounts of tokens that represent the current liquidity in range, update the comments and NatSpec accordingly. Otherwise, calculate the amount of tokens in the pool using the `IERC20.balanceOf()` function.

[L-07] No support for fee-on-transfer input token

Deposits of input tokens assume that the amount of tokens received by the contract is the same as the amount of tokens sent by the user. This is not the case when the input token has a fee-on-transfer mechanism. If this was the cause, the deposits fail when there is not enough balance in the contract to cover the fee, or will consume tokens from the reserve when there is enough balance.

If a fee-on-transfer token is used as the input token, the `launch` function is expected to revert, but it might not be the case if there is a donation of the input token to the contract.

If the protocol wants to support fee-on-transfer tokens, it should check the contract balance before and after the deposit and use the difference as the deposited amount.

If there is no intention to support fee-on-transfer tokens, it is advisable to check in the `launch` function that the amount of tokens received by the contract is the same as the amount of tokens sent by the user.

[L-08] `getUserData` returns the wrong streak input upper boundary for cycle 1

The `streakInputTokenAmountBoundaries` value returned by the `getUserData` is a tuple of (min, max) input token amounts needed to maintain the streak in the next cycle.

When this function is called for cycle 1, the upper value of the boundary will be 0, which is incorrect, as the code does not apply any boundary for the first cycle.

Additionally, it would be recommended to check that the cycle number is greater than 0.

Consider applying the following changes:

```
function getUserData(uint32 cycle, address user) external view returns
      (UserData memory) {
-
-        uint256 inputTokenDepositedPreviousCycle = totalInputTokenDeposited[cycle - 1
-       uint256[2] memory streakInputTokenAmountBoundaries =
-
-             [inputTokenDepositedPreviousCycle, inputTokenDepositedPreviousCycle * 10
+       require(cycle > 0, "Cycle number must be greater than 0");
+
+       uint256[2] memory streakInputTokenAmountBoundaries;
+
+       if (cycle == 1) {
+           streakInputTokenAmountBoundaries = [0, type(uint256).max];
+       } else {
+
+            uint256 inputTokenDepositedPreviousCycle = totalInputTokenDeposited[cycle
+
+            streakInputTokenAmountBoundaries = [inputTokenDepositedPreviousCycle, inp
+       }
```

[L-09] DoS vector in the `launch` function

The provisional owner of the contract launches the ULTI token by calling the `launch` function. In the private function `_createLiquidity` it is checked that Uniswap pool for the ULTI token does not exist and if it does, the function reverts.

```
    function _createLiquidity(uint256 inputTokenForLP, uint256 deadline) private {
        // 1. Check if the Uniswap pool already exists
        // This check prevents a potential DoS attack on the `launch`
// function
        // If an attacker creates the pool before the contract owner,
        // it would revert here instead of allowing the attacker to
// manipulate the initial state
        address liquidityPoolAddress = uniswapFactory.getPool(address(this), inputTokenAddress, LP_FEE);

        // 2. Create and store pool if it doesn't exist
        if (liquidityPoolAddress == address(0)) {
            liquidityPoolAddress = uniswapFactory.createPool(address(this), inputTokenAddress, LP_FEE);
            liquidityPool = IUniswapV3Pool(liquidityPoolAddress);
        } else {
@>          revert LiquidityPoolAlreadyExists();
        }
```

While the comments state that a DoS attack is prevented, this is not the case. Reverting the transaction if the pool already exists prevents the manipulation of the initial price of the pool, but in doing so, it creates a DoS vector. If an attacker creates the pool before the contract owner, the `launch` function will revert, forcing the redeployment of the contract.

Consider executing the logic of the `launch` function in the constructor of the contract.

[L-10] Native tokens received by the `receive` and `fallback` can be locked

The ULTI contract implements the `receive` and `fallback` functions to receive native tokens, but does not implement any logic to handle them, so they will be locked in the contract indefinitely.

If the intention is that the contract accepts donations to the long-term reserve, consider implementing the following changes:

```
// Receive function
    receive() external payable {
        require(inputTokenAddress == wrappedNativeTokenAddress);
        IWrappedNative(wrappedNativeTokenAddress).deposit{value: msg.value}();
    }

    // Fallback function
    fallback() external payable {
        require(inputTokenAddress == wrappedNativeTokenAddress);
        IWrappedNative(wrappedNativeTokenAddress).deposit{value: msg.value}();
    }
```

Otherwise, consider removing the `receive` and `fallback` functions.

[L-11] Potential overflow in `_getSpotPrice()` during price calculation

The _getSpotPrice() function performs direct multiplication of `sqrtPriceX96` with itself to calculate `priceX192` without checking for overflow. Since `sqrtPriceX96` is a `uint160`, the multiplication could overflow if the value exceeds `2^128`.

The overflow would cause the transaction to revert, preventing:

- Deposits that check spot price
- Pumps that rely on price calculations
- Any other functions using `_getSpotPrice()`

```
function _getSpotPrice() internal view returns (uint256 spotPrice) {
        // 1. Get square root price from pool slot0
        (uint160 sqrtPriceX96,,,,,,) = liquidityPool.slot0();

        // 2. Square the price to get priceX192
        uint256 priceX192 = uint256(sqrtPriceX96) * uint256
        //(sqrtPriceX96); // @audit overflow if sqrtPriceX96 > 2^128

        ...
    }
```

Address the scenario where `sqrtpriceX96` exceeds 2^128. Refer to the `getQuoteAtTick` function in the Oracle library from Uniswap v3 for guidance.

```
// Calculate quoteAmount with better precision if it doesn't overflow
        // when multiplied by itself
        if (sqrtRatioX96 <= type(uint128).max) {
            // Calculate quoteAmount with higher precision
            ...
        } else {
            // Calculate quoteAmount with lower precision
            ...
        }
```

[L-12] Losing some part of the streak bonus if depositing in multiple transactions

When users deposit in continuous cycles and follow the deposit limit checks, the code gives streak bonus rewards based on the number of steak counts. One of the checks is that the current cycle deposits must be higher than the previous cycle deposits and if that's not the case then the code doesn't give a streak bonus for the user.

```
// 1. Cumulative input token deposits this cycle is too low compared to
    // previous cycle

                bool lowCumulativeDepositTooLow = inputTokenDepositedForCycleN < inpu

        // 2. Cumulative input token deposits this cycle is too high compared to
        // previous cycle

                bool cumulativeDepositTooHigh = inputTokenDepositedForCycleN > 10 * i

        // 4. Update streak count based on deposit validity
        if (!lowCumulativeDepositTooLow && !cumulativeDepositTooHigh) {
            streakCounts[cycle][user] = streakCounts[cycle - 1][user] + 1;
        } else {
            // Break the streak
            streakCounts[cycle][user] = 1;
        }
```

The issue is that if the user deposits his tokens in multiple transactions in one cycle then for the early deposits in the cycle user won't receive the streak bonus and only would receive the streak bonus when the amount of deposits in the current cycle gets bigger than the previous cycle. This would prevent users from receiving a full streak bonus if they deposit in multiple transactions and users have to deposit only when their deposit amount is bigger than their previous cycle deposits. While this would cause users to deposit late and their discounted contribution would be lower.

Change the bonus calculation so it doesn't depend on how many deposits users make. One solution would be to add a check that if `(totalInputTokenDeposited[cycle][user] < totalInputTokenDeposited[cycle - 1][user]) and (totalInputTokenDeposited[cycle][user] + inputTokenDeposited > totalInputTokenDeposited[cycle - 1][user])` then sum of current cycle deposits is entering the valid interval and code should calculating streak bonus for all the `totalInputTokenDeposited[cycle][user] + inputTokenDeposited`.