code4rena

Ulti

Smart Contract Security Assessment

Version 2.0

Audit dates: Sep 25 — Sep 27, 2024

Audited by: xiaoming9090
spicymeatball

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Ulti

ULTI is a decentralized protocol designed for growth. ULTI incentivizes community members to contribute their capital to a decentralized autonomous reserve that incrementally & perpetually purchases the ULTI token on the largest decentralized market. Consistent, competitive cycles reward the most dedicated users based on their contributions, both directly or via referred users.

2.2 Scope

Repository	ulti-org/ulti-protocol-contract
Commit Hash	3d90acf1e5bcc200005ddb9c0045b1bc939ac045
Mitigation Hash	a12e513edb4ea334948dac02db4329d84465ecb2

2.3 Audit Timeline

DATE	EVENT
Sep 25, 2024	Audit start
Sep 27, 2024	Audit end
Oct 25, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	9
Low Risk	3
Informational	4
Total Issues	20

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	ULTI uses raw cumulative values to calculate TWAP	Resolved

H-2	Referrer bonus can be abused to drain liquidity from the pool	Resolved
H-3	`_updateTWAP()` set the price in the reverse order	Resolved
H-4	Incorrect assumption related to token ordering	Resolved
M-1	Incorrect scaling of streak bonus with adoption multiplier	Resolved
M-2	A significant streak bonus can be achieved with minimal investment	Resolved
M-3	Insufficient slippage check in `_swapEthForUlti`	Resolved
M-4	The streak bonus is calculated incorrectly	Resolved
M-5	`deposit` function can be unavailable during volatile markets	Resolved
M-6	Seeding of ETH liquidity can be blocked	Resolved
M-7	`totalDirectlyMintedPerUser` can be bypassed to mint excessive referral bonuses	Resolved
M-8	Incorrect deadline parameter in Uniswap function calls	Resolved
M-9	Incorrect update to `cumulativeReferralBonuses`	Resolved
L-1	Referred rewards are excluded from `_updateContributors`	Resolved
L-2	Invalid `MAX_SLIPPAGE_BPS` value	Resolved
L-3	Streak bonus should been excluded when updating the `totalDirectlyMintedPerUser`	Resolved
I-1	Error in comments for `adoptionMultipliers`	Resolved
I-2	Pump cooldown does not align with the specification	Acknowledged
I-3	Dust amount of refunded ETH will remain in the `ULTI` contract	Resolved
I-4	Potential impact of TWAP during first 10 minutes	Acknowledged

4. Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] ULTI uses raw cumulative values to calculate TWAP

Severity: High	**Status:** Resolved

Context:

- ULTI.sol#L730-L731

Description: The `ULTI.sol` contract uses TWAP (time-weighted average price) to calculate the amount of ULTI tokens minted to a depositor:

```
    function deposit(address referrer) external payable nonReentrant
unstoppable returns (uint256, uint256) {
        _updateTWAP();

        require(msg.value > 0, "Some Ether is required");
        require(referrer != msg.sender, "Cannot refer yourself");
        require(referrers[referrer] != msg.sender, "Cannot be referred by
a user that you already referred");

        uint256 cycle = getCurrentCycle();

        // Calculate ULTI tokens to mint
        uint256 ethDeposited = msg.value;
>>      uint256 ultiMinted = ethDeposited * twapPrice;
```

The TWAP is determined using a fixed 10-minute time window:

```
    function _updateTWAP() internal {
        if (!lpReady) return;

        // Fetch the current cumulative prices and block timestamp from
Uniswap V2 Oracle Library
        (uint256 price0Cumulative, uint256 price1Cumulative, uint32
blockTimestamp) =
UniswapV2OracleLibrary.currentCumulativePrices(lpAddress);
```

```
        // Calculate time elapsed since the last update
        uint32 timeElapsed = blockTimestamp - blockTimestampLast;

        if (timeElapsed < MIN_TWAP_TIME) {
            return; // Not enough time has passed to calculate TWAP
        }

        // Calculate the average price (TWAP) based on cumulative prices
and time elapsed
        uint256 priceDelta = price1Cumulative - price1CumulativeLast;
>>      twapPrice = priceDelta / timeElapsed;
```

However, there is an issue in the calculation of the `priceDelta`, as it is done without decoding the cumulative values retrieved from the Uniswap V2 pair:

```
    function _update(uint balance0, uint balance1, uint112 _reserve0,
uint112 _reserve1) private {
        ---SNIP---
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast +=
uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
            price1CumulativeLast +=
uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
        }
```

In Uniswap V2, cumulative prices are encoded as Q112.112 numbers. Without decoding them, the calculated `twapPrice` will be incorrect.

Recommendation: Use Uniswap's FixedPoint.sol library to decode cumulative prices before calculating the `twapPrice`.

ULTI: Fixed with the following commit

Zenith: Verified.

[H-2] Referrer bonus can be abused to drain liquidity from the pool

Severity: High Status: Resolved

Context:

- ULTI.sol#L192

Description:

At the start, the price is 330000 ULTI per ETH.

Alice deposits 0.25 ETH and minted 82500 ULTI, and `totalDirectlyMintedPerUser[Alice] = 82500`.

Bob deposits 0.75 ETH with the referrer set to Alice and minted 247500 UTLI. Alice will receive 81675 ULTI (Did not exceed the referral cap). Bob will receive a referred fee of 4900 ULTI (6% of 81675)

In total, both Alice and Bob received 416575 UTLI for a total of 1 ETH deposited into the `ULTI` contract. Thus, this reflects an inflation of around 26.23% `((81675+4900)/330000)`. This is the first inflation observed.

Assume Alice and Bob sold all their 416575 UTLI to the Uniswap V2 pool. This will increase the number of UTLI tokens and decrease the number of ETH in the pool. As such, the price of ULTI tokens decreases, meaning more UTLI tokens can be obtained per ETH. Thus, when a user deposits into `UTLI` contract after the TWAP period (Actual pool's price takes effect), they will receive more ULTI per ETH compared to earlier (Increase from 330000 ULTI per ETH to 338384 ULTI per ETH). This is the second inflation observed.

[!NOTE]

Here, the Uniswap formula (r0 * r1 = K) is used, where K must remain constant after the swap. Ignoring the swap fee for simplicity's sake. Otherwise, the model will be complex.

Assume that in each round, the malicious user creates two (2) new accounts/wallets to deposit 1 ETH. All received ULTI tokens are then deposited to the Uniswap Pool for ETH. More than 1 ETH will be received, and any amount over 1 ETH is kept as profit. Repeat 80 times (takes around 800 minutes), and the malicious user will gain 16.86 ETH

Following is the spreadsheet where the impact is computed.

https://docs.google.com/spreadsheets/d/1R4VI0--pGVphlb_XBiIAz7CivujzbF0brpyLTcJvROw/edit?usp=sharing

Recommendation: Review the potential impact of ULTI inflation.

ULTI: Addressed the initial findings as per the thread in the following commit

Zenith: Verified. Each claiming of referrer bonus is now subjected to a 33-day delay, limiting the ability to cycle through the `deposit to ULTI > claim bonus > deposit to Uniswap pool > deposit to ULTI` sequence.

[H-3] `_updateTWAP()` set the price in the reverse order

Severity: High	Status: Resolved

Context:

- ULTI.sol#L720

Description:

Per the source code of `UniswapV2OracleLibrary.currentCumulativePrices` function, `price1Cumulative` is computed by $\frac{reserve0}{reserve1}$, and it will give the price of 1 unit of token1 in terms of token0.

However, within the `launch()` function, the price is computed by $\frac{reserve1}{reserve0}$. Assuming that `reserve0` is WETH, while `reserve1` is ULTI. The `_updateTWAP()` will set the price in the reverse order, leading to an incorrect price.

```
File: ULTI.sol
    function launch() external payable onlyOwner {
..SNIP..
        // Initialize TWAP variables
        (uint112 reserve0, uint112 reserve1, uint32 blockTimestamp)
= lpPair.getReserves(); // @audit-info ETH
        blockTimestampLast = blockTimestamp;
        twapPrice = reserve1 / reserve0;
```

Recommendation:

Update the logic of the `_updateTWAP` function to ensure the price ordering is correct and consistent.

ULTI: Fixed in the following commit

Zenith: Verified

[H-4] Incorrect assumption related to token ordering

Severity: High	**Status:** Resolved

Context:

- ULTI.sol#L179
- ULTI.sol#L383-L386

Description:

The `twapPrice` is the price of 1 unit of ETH in terms of ULTI. Thus, the denominator must always be ETH. However, there is no guarantee that `reserve0` will point to ETH.

In Uniswap V2, tokens are sorted by their contract addresses in ascending order, and the token with the lower address is assigned to token0, while the token with the higher address is assigned to token1.

In Uniswap V2, ETH is wrapped to WETH under the hood. The WETH address on Ethereum is `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2`. If the address of ULTI after deployment is lower than the WETH address, the token0 will point to ULTI instead of WETH, resulting in an incorrect price.

Recommendation:

Additional logic is needed to ensure that the correct reserve (`reserve0` or `reserve1`) is set as the denominator when computing the price. Denominator must be WETH/ETH.

ULTI: Noted. Fixed by introducing a boolean flag `IsUltiToken0` as part of the following commit.

Zenith: Verified

4.2 Medium Risk

A total of 9 medium risk findings were identified.

[M-1] Incorrect scaling of streak bonus with adoption multiplier

Severity: Medium	Status: Resolved

Context:

- ULTI.sol#L708

Description: When calculating `streakBonusWithAdoptionMultiplier`, the `adoptionMultiplier` is applied to the `streakBonus`:

```
        // Calculate the streak bonus percentage
        uint256 streakBonusPercentage =
STREAK_BONUS_PERCENTAGE_INCREASE_PER_CYCLE * streakCount;

        // Calculate the streak bonus
        uint256 streakBonus = ultiMinted * streakBonusPercentage / 100;
>>      uint256 streakBonusWithAdoptionMultiplier = streakBonus *
getAdoptionMultiplier(currentCycle) / PRECISION_FACTOR;
```

The adoption multiplier is stored as `percentage * PRECISION_FACTOR`:

```
    // Precomputed values for efficiency
    // Adoption multipliers: Exponential decay values for days 1 to 33,
from 33 to 3 scaled by 10^6 for precision
    uint256[ULTI_NUMBER] public adoptionMultipliers = [
        33000000, 30617548, 28407099, 26356235, 24453433, 22688006,
21050034, 19530316, 18120316, 16812110,
        15598352, 14472221, 13427392, 12457995, 11558584, 10724106,
9949874, 9231538, 8565062, 7946703,
        7372987, 6840691, 6346824, 5888612, 5463480, 5069042, 4703080,
4363538, 4048511, 3756226,
        3485044, 3233439, 3000000
    ];
```

However, the final calculation does not account for this scaling, leading to an inflated streak bonus value. The multiplier needs to be properly scaled down by dividing by 100 to get the actual bonus value with applied percentage.

Recommendation:

```
+ uint256 streakBonusWithAdoptionMultiplier = streakBonus *
getAdoptionMultiplier(currentCycle) / (PRECISION_FACTOR * 100);
```

ULTI: Fixed with the following commit

Zenith: Verified

[M-2] A significant streak bonus can be achieved with minimal investment

Severity: Medium	Status: Resolved

Context:

- ULTI.sol#L693-L701

Description: The ULTI protocol rewards users with a streak bonus that increases their minted ULTI based on consecutive deposits across cycles. If a user mints more ULTI in the current cycle than the previous one, their streak count is incremented:

```
    function _calculateStreakBonus(address user, uint256 ultiMinted,
uint256 currentCycle) internal view returns (uint256) {
        if (currentCycle < 3) return 0;

        uint256 ultiMintedPreviousCycle = ultiMintedForCycle[currentCycle
- 1][user];
        if(ultiMintedPreviousCycle == 0) return 0;
        // Cap the streak count at 10 cycles
        uint256 streakCount = 0;
        for (uint256 i = 1; i <= STREAK_BONUS_CYCLE_CAP && currentCycle >
i; i++) {
            if (currentCycle <= i) break;
            uint256 ultiMintedIMinus1CycleAgo =
ultiMintedForCycle[currentCycle - i - 1][user];
            if (ultiMintedPreviousCycle >= ultiMintedIMinus1CycleAgo) {
>>              streakCount++;
            } else {
                break;
            }
        }
```

However, there is a vulnerability in this algorithm. Since there is no minimum limit for the minted ULTI amount, users can deposit tiny amounts of ETH (e.g., 100, 101, 102 wei) in each cycle to artificially boost their streak bonus without making significant contributions.

Recommendation: To prevent abuse, additional rules should be implemented for streak bonus calculation. For example minted amount for a cycle should be no lesser than `A * X prev`, where `A` - is fixed coefficient and `Xprev` - is ULTI minted in the previous cycle.

Additionally, enforcing a minimum deposit amount would further discourage users from exploiting the streak mechanism with insignificant deposits.

ULTI: Fixed with the following commit

Zenith: Verified. An additional check is included to ensure that the current cycle's minted amount is consistent with previous cycles.

[M-3] Insufficient slippage check in `_swapEthForUlti`

Severity: Medium Status: Resolved

Context:

- ULTI.sol#L562-L563

Description:

```
    function _swapEthForUlti(uint256 ethAmount) internal returns
(uint256[] memory) {
        address[] memory path = new address[](2);
        path[0] = wethAddress;
        path[1] = address(this);

        // Calculate the minimum amount of ULTI tokens to receive
(accounting for slippage)
        uint256[] memory amountsOut =
uniswapRouter.getAmountsOut(ethAmount, path);
>>      uint256 amountOutMin = (amountsOut[1] * (10000 -
MAX_SLIPPAGE_BPS)) / 10000; // 0.5% max slippage

        // Execute the swap
        return uniswapRouter.swapExactETHForTokens{value: ethAmount}(
            amountOutMin,
            path,
            msg.sender,
            block.timestamp + 15 minutes
        );
    }
```

`amountOutMin` is calculated based on the current price that is fetched at the moment of the transaction execution, which won't protect the contract from slippage issues. Consider the following scenario:

- the ULTI/ETH price is 330,000
- a user calls the `pump` function, but an ETH-to-ULTI swap is frontrun, driving the price down to 320,000;
- inside `_swapEthForUlti`, the minimum amount is calculated as 320,000 * 0.995 = 318,400;
- the problem is that the received amount will always equal 320,000, meaning slippage check in Uniswap router will always pass.

Recommendation: To protect the protocol from slippage during ETH-to-ULTI swaps it is recommended to specify `minUltiAmount` as an argument in the `pump` function:

```
+   function _swapEthForUlti(uint256 ethAmount, uint256 minUltiAmount)
internal returns (uint256[] memory) {
        ---SNIP---
        return uniswapRouter.swapExactETHForTokens{value: ethAmount}(
+           minUltiAmount,
            path,
            msg.sender,
            block.timestamp + 15 minutes
        );
    }
```

ULTI: Fixed with PR-2

Zenith: Verified

[M-4] The streak bonus is calculated incorrectly

Severity: Medium | Status: Resolved

Context:

- ULTI.sol#L695-L696

Description: The ULTI protocol rewards users who deposit every cycle with a streak bonus, which multiplies their minted ULTI when depositing ETH. The streak bonus applies when the amount of ULTI minted in the current cycle exceeds that of the previous cycle. However, there is a flaw in the streak-counting algorithm:

```
    function _calculateStreakBonus(address user, uint256 ultiMinted,
uint256 currentCycle) internal view returns (uint256) {
        if (currentCycle < 3) return 0;

>>      uint256 ultiMintedPreviousCycle = ultiMintedForCycle[currentCycle
- 1][user];
        if(ultiMintedPreviousCycle == 0) return 0;
        // Cap the streak count at 10 cycles
        uint256 streakCount = 0;
        for (uint256 i = 1; i <= STREAK_BONUS_CYCLE_CAP && currentCycle >
i; i++) {
            if (currentCycle <= i) break;
            uint256 ultiMintedIMinus1CycleAgo =
ultiMintedForCycle[currentCycle - i - 1][user];
>>          if (ultiMintedPreviousCycle >= ultiMintedIMinus1CycleAgo) {
                streakCount++;
            } else {
                break;
            }
        }
```

The issue arises because the value of `ultiMintedPreviousCycle` is not updated during the loop, leading to the same amount being compared against the ULTI minted in all previous cycles.

For example, if a user mints 1,000 tokens in cycle 10 and then deposits in cycle 11, those 1,000 tokens will be compared against all previous cycles, which are zero. This would result in a `streakCount = 10`, even though the user wasn't committing for all 10 cycles.

Recommendation:

```diff
        for (uint256 i = 1; i <= STREAK_BONUS_CYCLE_CAP && currentCycle >
i; i++) {
-           if (currentCycle <= i) break;
            uint256 ultiMintedIMinus1CycleAgo =
ultiMintedForCycle[currentCycle - i - 1][user];
+           if (ultiMintedPreviousCycle >= ultiMintedIMinus1CycleAgo &&
ultiMintedPreviousCycle !=0) {
                streakCount++;
            } else {
                break;
            }
+           ultiMintedPreviousCycle = ultiMintedForCycle[currentCycle - i
- 1][user];
        }
```

ULTI: Fixed with the following commit

Zenith: Verified

[M-5] `deposit` function can be unavailable during volatile markets

Severity: Medium	Status: Resolved

Context:

- ULTI.sol#L542-L543
- UniswapV2Router02.sol#L51
- UniswapV2Router02.sol#L56

Description: When a user calls the `deposit` function, a portion of ETH and ULTI is used to supply liquidity to the ETH/ULTI pair on Uniswap:

```
    function _addLiquidity(uint256 ultiForLP, uint256 ethForLP) internal
returns (address) {
        require(ethForLP > 0 && ultiForLP > 0, "Not enough ETH or ULTI in
the contract to add liquidity");
        ---SNIP---
        // Add liquidity to Uniswap
        uniswapRouter.addLiquidityETH{value: ethForLP}(
            address(this),
            ultiForLP,
>>          (ultiForLP * (10000 - MAX_SLIPPAGE_BPS)) / 10000, // 0.5% max
slippage for ULTI
>>          (ethForLP * (10000 - MAX_SLIPPAGE_BPS)) / 10000, // 0.5% max
slippage for ETH
            address(this),
            block.timestamp + 15 minutes
        );
```

The contract sets slippage limits based on the `ultiForLP` value calculated using TWAP. However, Uniswap's router uses the current spot price for slippage checks:

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) ==
address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) =
UniswapV2Library.getReserves(factory, tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            // @audit spot price is used
>>          uint amountBOptimal = UniswapV2Library.quote(amountADesired,
reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
>>              require(amountBOptimal >= amountBMin, 'UniswapV2Router:
INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                // @audit spot price is used
>>              uint amountAOptimal =
UniswapV2Library.quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
>>              require(amountAOptimal >= amountAMin, 'UniswapV2Router:
INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

This can cause issues if there's a difference between the TWAP and the spot price, resulting in the transaction being reverted due to slippage check failure. Consider a following scenario:

- ULTI is deployed at t0 with reserves of 10,890,000 ULTI and 33 ETH, setting the ETH/ULTI price at 330,000.
- At t0 + 500, Alice swaps 1 ETH for ULTI via the Uniswap router. Now, reserves are 10,269,474 ULTI and 35 ETH, and the spot price drops to 293,413.
- At t0 + 600, Bob calls deposit with 1 ETH. Since the TWAP (323,902) lags behind the spot price (293,413), and the `ultiForLP` is calculated using TWAP, there could be a scenario where `amountOptimal < (ultiForLP * (10000 - MAX_SLIPPAGE_BPS)) / 10000`, causing the transaction to revert.

Recommendation: It is recommended to set slippage parameters in `_addLiquidity` to zero:

```
    function _addLiquidity(uint256 ultiForLP, uint256 ethForLP) internal
returns (address) {
        ---SNIP---
        // Add liquidity to Uniswap
        uniswapRouter.addLiquidityETH{value: ethForLP}(
            address(this),
            ultiForLP,
+           0,
+           0,
            address(this),
            block.timestamp + 15 minutes
        );
```

Additionally, allow depositors to specify a minimum acceptable amount of ULTI to protect against slippage:

```
+   function deposit(address referrer, uint256 minUltiMinted) external
payable nonReentrant unstoppable returns (uint256, uint256) {
        _updateTWAP();

        require(msg.value > 0, "Some Ether is required");
        require(referrer != msg.sender, "Cannot refer yourself");
        require(referrers[referrer] != msg.sender, "Cannot be referred by
a user that you already referred");

        uint256 cycle = getCurrentCycle();

        // Calculate ULTI tokens to mint
        uint256 ethDeposited = msg.value;
        uint256 ultiMinted = ethDeposited * twapPrice;
+       require(ultiMinted >= minUltiMinted, "Not enough ULTI");
```

ULTI: Fixed with PR-1

Zenith: Verified.

[M-6] Seeding of ETH liquidity can be blocked

Severity: Medium	Status: Resolved

Context:

- ULTI.sol#L165-L184

Description: Since the ULTI deployment and `launch` occur in separate transactions, there's a small chance that someone could create a ULTI LP Pair in Uniswap V2 Factory before the liquidity is seeded. An attacker could provide 1 wei of WETH and call the `sync` function on the pair:

```
    // force reserves to match balances
    function sync() external lock {
        _update(IERC20(token0).balanceOf(address(this)),
IERC20(token1).balanceOf(address(this)), reserve0, reserve1);
    }
```

This would cause reserve imbalance, where `reserve0 = 1` and `reserve0 = 0` (assume that ULTI is `token0`). Thus the subsequent `addLiquiidty` call in the `launch` transaction will fail with INSUFFICIENT_LIQUIDITY error:

```
    // given some amount of an asset and pair reserves, returns an
equivalent amount of the other asset
    function quote(uint amountA, uint reserveA, uint reserveB) internal
pure returns (uint amountB) {
        require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
>>      require(reserveA > 0 && reserveB > 0, 'UniswapV2Library:
INSUFFICIENT_LIQUIDITY');
        amountB = amountA.mul(reserveB) / reserveA;
    }
```

Because `founderGiveaway` funds are transferred during deployment, they may become trapped if the `launch` fails.

Recommendation: It is recommended to transfer `founderGiveaway` in `launch` transaction or implement emergency withdraw function to retreive ETH if the `launch` fails.

ULTI: Fixed with the following commit

Zenith: Verified

[M-7] `totalDirectlyMintedPerUser` can be bypassed to mint excessive referral bonuses

Severity: Medium   Status: Resolved

Context:

- ULTI.sol#L254

Description:

The intention of `totalDirectlyMintedPerUser` is to prevent excessive referral bonuses from being minted.

```
File: ULTI.sol
253:            // Cap referral bonus to prevent excessive rewards
254:            uint256 cap = totalDirectlyMintedPerUser[referrer] -
cumulativeReferralBonuses[referrer][depositor];
255:            ultiForReferrer = ultiForReferrer > cap ? cap :
ultiForReferrer;
256:
257:            if (ultiForReferrer > 0) {
258:                _mint(referrer, ultiForReferrer);
```

Assume that Alice directly minted 1000 ULTI (`totalDirectlyMintedPerUser[Alice] = 1000`). Ideally, we should limit Alice's referral bonus to 1000.

However, since the cap is based on the referrer/depositor pair (`cumulativeReferralBonuses[referrer][depositor]`), one could create 100 new accounts and set Alice as a referral bonus, and she could receive up to `100 * 1000` referral bonuses. It might be better to reduce Alice's capacity/bonus allowance whenever a referral bonus is minted to her.

Recommendation: Consider reducing the user's capacity/bonus allowance whenever a referral bonus is minted

ULTI:

Agree with this observation. I came up with a modification to keep the logic and incentives simple while preventing significant abuses.

```
// Cap referral bonus to so that the referrer has skin in the game
// The cap is 10X the total ULTI directly minted by the referrer minus
referral bonuses not yet claimed
// Once the referrer has claimed their bonuses, the cap is reset
uint256 remainingBonusAllowance = 10 *
totalDirectlyMintedPerUser[referrer] - referralBonuses[referrer];
ultiForReferrer = ultiForReferrer > remainingBonusAllowance ?
remainingBonusAllowance : ultiForReferrer;

if (ultiForReferrer > 0) {
    referralBonuses[referrer] += ultiForReferrer;
    _updateContributors(cycle, referrer, 0, msg.value, ultiForReferrer);

    // Bonus for the referred user
    ultiForReferred = (ultiForReferrer *
REFERRAL_BONUS_FOR_REFERRED_PERCENTAGE) / 100;
    if (ultiForReferred > 0) {
        referralBonuses[depositor] += ultiForReferred;
    }
}
```

Addressed in the following commit

Zenith: Verified. Bonus allowance is now based on a per-user basis instead of referrer/depositor pair.

[M-8] Incorrect deadline parameter in Uniswap function calls

Severity: Medium	Status: Resolved

Context:

- ULTI.sol#L545
- ULTI.sol#L570

Description: When the protocol interacts with Uniswap V2 via adding liquidity or swapping tokens, it uses the `deadline` parameter. This parameter is checked by the Uniswap router to ensure the transaction is executed within a specific time window. If the time at execution exceeds the deadline, the transaction is reverted:

```
    function swapETHForExactTokens(uint amountOut, address[] calldata
path, address to, uint deadline)
        external
        virtual
        override
        payable
>>      ensure(deadline)
```

This mechanism protects users and the protocol from latency issues caused by delayed transactions. However, the `ULTI.sol` contract specifies the `deadline` parameter at the moment of execution, therefore current `block.timestamp` is used:

```
    function _swapEthForUlti(uint256 ethAmount) internal returns
(uint256[] memory) {
        ---SNIP---

        // Execute the swap
        return uniswapRouter.swapExactETHForTokens{value: ethAmount}(
            amountOutMin,
            path,
            msg.sender,
>>          block.timestamp + 15 minutes
        );
```

This causes the deadline check to always pass, leaving the contract vulnerable to slippage or latency risks.

Recommendation:

It is recommended to pass the `deadline` as an argument in the functions interacting with the Uniswap router, such as `deposit` and `pump`. In the `launch` function, however, setting `deadline = block.timestamp` is acceptable since there is no slippage risk during initial liquidity seeding.

ULTI: Fixed with the following commit

Zenith: Verified

[M-9] Incorrect update to `cumulativeReferralBonuses`

Severity: Medium Status: Resolved

Context:

- ULTI.sol#L271

Description:

The `cumulativeReferralBonuses` record all the referral bonuses minted to a specific referrer/depositor pair. Thus, the correct code should be as follows:

```
- cumulativeReferralBonuses[referrer][msg.sender] += ultiMinted;
+ cumulativeReferralBonuses[referrer][msg.sender] += ultiForReferred;
```

In addition, it might lead to underflow revert, as shown below.

Assuming that Bob is the referrer, and his `totalDirectlyMintedPerUser[Bob]` is 100. Alice is the depositor + referred user. At this point, `cumulativeReferralBonuses[referrer][depositor] = cumulativeReferralBonuses[Bob][Alice] = 0`.

At T0, Alice minted 1,000,000 ULTI, and `ultiForReferrer` will be $1{,}000{,}000 \times 33 \% = 330{,}000$ (First Cycle)

The cap will be 100. As a result, only 100 ULTI is minted to the referrer (Bob)

```
cap = totalDirectlyMintedPerUser[referrer] -
cumulativeReferralBonuses[referrer][depositor]
cap = totalDirectlyMintedPerUser[Bob] - cumulativeReferralBonuses[Bob]
[Alice]
cap = 100 - 0 = 100

ultiForReferrer = ultiForReferrer > cap ? cap : ultiForReferrer
ultiForReferrer = 330000 > 100 ? cap : ultiForReferrer
ultiForReferrer = True ? 100 : 330000
ultiForReferrer = 100
```

At the end of the function, the following code is executed, which updates `cumulativeReferralBonuses[Bob][Alice]` to `1000000`

```
cumulativeReferralBonuses[referrer][msg.sender] += ultiMinted;
```

```
cumulativeReferralBonuses[Bob][Alice] += 1000000
```

At T1, Alice minted another 500,000 ULTI, and set Bob as referrer again. In this case, the transaction will revert due to an underflow in the following code. As a result, it led to a loss of referrer bonus for both Alice and Bob as the transaction cannot proceed as intended.

```
cap = totalDirectlyMintedPerUser[referrer] -
cumulativeReferralBonuses[referrer][depositor]
cap = totalDirectlyMintedPerUser[Bob] - cumulativeReferralBonuses[Bob]
[Alice]
cap = 100 - 1000000 (Underflow - Revert)
```

Recommendation:

Consider making the following changes:

```
- cumulativeReferralBonuses[referrer][msg.sender] += ultiMinted;
+ cumulativeReferralBonuses[referrer][msg.sender] += ultiForReferred;
```

ULTI: I addressed both points in the following commit

1. `ultiForReferred` is now used instead of `ultiMinted` in the accumulator (also moved the line up to be executed only if `ultiForReferrer > 0`)
2. a conditional statement was added to prevent the underflow

Zenith: Verified

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] Referred rewards are excluded from `_updateContributors`

Severity: Low	Status: Resolved

Context:

- ULTI.sol#L264
- ULTI.sol#L227

Description: When a user deposits ETH into ULTI, the minted ULTI amount is composed of three parts:

```
U = Ueth + Ustreak + Uref
Ueth - ULTI minted based on ETH/ULTI price;
Ustreak - streak bonus
Uref - bonus for the referred user
```

Currently, the referred bonus (Uref) is not included when the `_updateContributors` function is called, which leads to inaccurate contributor updates.

Recommendation: Consider including ULTI minted as referred rewards in the contributors data update.

ULTI: Fixed with the following commit

Zenith: Verified.

[L-2] Invalid `MAX_SLIPPAGE_BPS` value

Severity: Low	Status: Resolved

Context: ULTI.sol#L53

Description: The `ULTI.sol` contract defines `MAX_SLIPPAGE_BPS` to calculate the acceptable slippage for ETH to ULTI swaps:

```
uint256 public constant MAX_SLIPPAGE_BPS = 500; // 500 in basis points
(0.5%)
```

However, `MAX_SLIPPAGE_BPS = 500` actually represents 5%, not 0.5%, as indicated in the comment. The contract uses 10,000 as the percentage divisor, so the slippage is incorrectly calculated.

Recommendation: To correctly represent 0.5% slippage, `MAX_SLIPPAGE_BPS` should be set to 50.

ULTI: Fixed with the following commit

Zenith: Verified

[L-3] Streak bonus should been excluded when updating the `totalDirectlyMintedPerUser`

Severity: Low | Status: Resolved

Context:

- ULTI.sol#L211

Description:

It would be more secure if `totalDirectlyMintedPerUser` excludes the streak bonus. Technically, the `totalDirectlyMintedPerUser` should only record the ULTI directly minted by the users via the ETH they paid for (`ethDeposited * twapPrice`). Streak Bonus ULTI is technically considered extra ULTI, which the user receives as a bonus but isn't directly paid for.

```
File: ULTI.sol
205:          // Apply streak bonus
206:          uint256 streakBonus = _calculateStreakBonus(msg.sender,
ultiMinted, cycle);
207:          ultiMinted += streakBonus;
208:
209:          // Mint ULTI tokens to the depositor
210:          _mint(msg.sender, ultiMinted);
211:          totalDirectlyMintedPerUser[msg.sender] += ultiMinted;
```

Recommendation: Consider excluding streak bonus when updating the `totalDirectlyMintedPerUser`.

```
// Apply streak bonus
uint256 streakBonus = _calculateStreakBonus(msg.sender, ultiMinted,
cycle);
ultiMinted += streakBonus;

// Mint ULTI tokens to the depositor
_mint(msg.sender, ultiMinted);
- totalDirectlyMintedPerUser[msg.sender] += ultiMinted;
+ totalDirectlyMintedPerUser[msg.sender] += (ultiMinted - streakBonus);
```

ULTI: Addressed in the following commit

Zenith: Fixed. Streak bonus is no longer included in the `totalDirectlyMintedPerUser`.

4.4 Informational

A total of 4 informational findings were identified.

[I-1] Error in comments for `adoptionMultipliers`

Severity: Informational	Status: Resolved

Context:

- ULTI.sol#L68

Description:

```
    // Precomputed values for efficiency
    // Adoption multipliers: Exponential decay values for days 1 to 33,
from 33 to 3 scaled by 10^6 for precision
    uint256[ULTI_NUMBER] public adoptionMultipliers = [
```

The comments in the code state that adoption multipliers are configured for "days 1 to 33." However, these values are actually applied to cycles, not days.

Recommendation:

```
+   // Adoption multipliers: Exponential decay values for cycles 1 to 33,
from 33 to 3 scaled by 10^6 for precision
```

ULTI: Fixed with the following commit

Zenith: Verified

[I-2] Pump cooldown does not align with the specification

Severity: Informational	**Status:** Acknowledged

Context:

- ULTI.sol#L281

Description:

Per the whitepaper:

Frequency: Pumps can be initiated every ~1 hour, ensuring that the process supports sustainable growth with tight slippage tolerance, preventing market instability, and arbitrage opportunities like MEV.

It mentioned that the frequency is 1 hour. However, in the actual implementation, the pump can be triggered once every 5 minutes 30 seconds.

```
File: ULTI.sol
    // Pump-related constants
    uint256 public constant PUMP_INTERVAL = ULTI_NUMBER * 10 seconds;
// 330 seconds (5 minutes 30 seconds)
```

Recommendation:

ULTI: I had decreased this value down for testing purposes. 3300 seconds (55 minutes) should be the final value.

Zenith: Acknowledged

[I-3] Dust amount of refunded ETH will remain in the `ULTI` contract

Severity: Informational	Status: Resolved

Context:

- ULTI.sol#L539

Description:

When the `UniswapV2Router02.addLiquidityETH` is executed, there might be dust amount of ETH being refunded. This will be refunded to the `ULTI` contract instead of the caller.

https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router02.sol#L99

```
    function addLiquidityETH(
        address token,
..SNIP..
        // refund dust eth, if any
        if (msg.value > amountETH)
TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
    }
```

Recommendation: Consider either transferring the dust ETH back to the callers OR adding a comment to document this behavior.

ULTI: The contract will keep the dust, I'll comment on this 3P call: "Executing addLiquidityETH may not utilize 100% of the ETH provided, and the remaining ETH will be refunded to the contract, which is fine" The contract will keep the dust. Comment added in the following commit

Zenith: Verified. A comment regarding this has been added to the codebase.

[I-4] Potential impact of TWAP during first 10 minutes

Severity: Informational	Status: Acknowledged

Context:

- ULTI.sol#L725

Description:

It might be better to revert instead of returning and have the deposits resume after the TWAP takes effect. In the current implementation, during launch, the price in `ULTI` contract and UniswapV2 pool is set to 330000 ULTI per ETH (According to `LP_INITIAL_RATIO`).

During the first 10 minutes, the `_updateTWAP()` function will simply return. Thus, the price in `ULTI` contract will always remain static at 330000 ULTI per ETH, while the price in UniswapV2 pool is dynamic (it can go up or down depending on trading direction). In a scenario where the ULTI's price in the pool is higher than the `ULTI` contract, it presents an arbitrage opportunity, which does not seem ideal.

Recommendation: Consider accepting deposits after the `MIN_TWAP_TIME` (10 minutes) has passed so that the TWAP can take effect.

ULTI: This is a good remark but after thinking more about it, the first 10 minutes after launch will likely see some volatility towards the upside making the deposit function as it is today the most appealing for options for knowledgeable users. Generic snipping bots that track new Uniswap pairs may push the price a bit offering arbitrage opportunities to users who read the whitepaper and followed public announcements and warnings. I think is reasonable to accept the early discrepancies between TWAP initially set at 330000 ULTI per ETH and the early price movement around this point. Also note that the initial seeding of the LP + very first large deposits should help soften early volatility.

C4: Acknowledged.