Ulti Security Audit

Report Version 1.0

November 19, 2024

Conducted by **Hunter Security**

Table of Contents

1 About Hunter Security

Hunter Security is an industry-leading smart contract security auditing firm. Having conducted over 100 security audits protecting over $1B of TVL, our team always strives to deliver top-notch security services to the best DeFi protocols. For security audit inquiries, you can reach out on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Ulti
Repository	https://github.com/ulti-org/ulti-protocol-contract
Commit hash	ce17e0ac3c1478effb1622b263e2e82b4aabf4b8
Resolution	57d6a1478dc63bb9e53dbbd7cd7c5818667a45f9
Methods	Manual review & testing

Scope

contracts/ULTI.sol

Issues Found

High risk	4
Medium risk	2
Low risk	6

5 Findings

5.1 High

5.1.1 Current cycle allocation updated instead of previous one

Severity: High

Context: ULTI.sol

Description: At the end of the first pump for a cycle, cycle maintenance is performed.

The problem is that the maintenance is performed for the current cycle while it should be done for the previous one that has just already passed.

Recommendation: Consider updating the previous cycle instead of the current one.

Resolution: Resolved.

5.1.2 Permanent Denial-of-Service due to not strictly limited pumper array

Severity: High

Context: ULTI.sol

Description: The pumpers array comprises all contributors who have invoked the pump function during the cycle.

The problem is that its length is not strictly limited. Despite having a maximum of 33 top contributors at a time, they can change after each pump interval. Therefore, the maximum possible pumpers count that can be reached is *33 days / 55 minutes = 864*. At this length, iterating through the array would most likely cause a revert due to Out-of-Gas exception causing the protocol permanent Denial-of-Service.

Recommendation: Consider enforcing a strict limit on the pumpers count similarly to the top contributors mechanism.

Resolution: Resolved.

5.1.3 Updating the TWAP in a flawed manner

Severity: High

Context: ULTI.sol

Description: When retrieving the TWAP, Uniswap's OracleLibrary.getQuoteAtTick is used to calculate the price after having the *timeWeightedAverageTick*.

The problem is in the arguments being passed to the function. It expects having the base token first (ULTI) and quote token second (WETH). However, the order is currently determined by the *isUltiToken0* flag. Therefore, the price will be incorrect if the *wethAddress* is > *address(this)*.

Recommendation: Consider always passing ULTI as the base token and removing the following inversion.

Resolution: Resolved.

5.1.4 Incorrect price conversion upon swapping

Severity: High

Context: ULTI.sol

Description: When swapping ETH for ULTI, the returned amount is calculated using the TWAP in order to apply slippage tolerance and check later.

The problem is that the calculation is using the TWAP as a multiplier rather than a divisor.

Recommendation: Consider implementing the following fix:

```
wethToSwap * 1e18 / _getTWAP()
```

Resolution: Resolved.

5.2 Medium

5.2.1 Not handling overflow protection when retrieving Uniswap price

Severity: Medium

Context: ULTI.sol

Description: When calculating the spot price, the *sqrtPriceX96* is obtained from the pool and then converted in the following way:

```
function _getSpotPrice() internal view returns (uint256 spotPrice) {
    (uint160 sqrtPriceX96,,,,,,) = liquidityPool.slot0();
    uint256 priceX192 = uint256(sqrtPriceX96) * uint256(sqrtPriceX96);

    if (address(this) < wethAddress) {
        spotPrice = priceX192 * 1e18 >> 192;
    } else {
        spotPrice = (1 << 192) * 1e18 / priceX192;
    }
}
```

This implementation is vulnerable to Denial-of-Service as the function would throw an overflow exception when the *priceX192* is greater than _(2**256 - 1) / 1e18_ which happens at about ~18x price difference.

Recommendation: Consider reusing the calculations made in UniswapV3's OracleLibrary.getQuoteAtTick to prevent potential overflow.

Resolution: Resolved.

5.2.2 Early bird period price may cause adding liquidity to fail

Severity: Medium

Context: ULTI.sol

Description: The early bird period allows users to purchase ULTI tokens at the predefined ratio that is also the pool's initial price.

The problem is that the current price in the pool (or the TWAP) may be significantly different from the early bird price as the duration of that period is 1 day. Therefore, adding liquidity to the pool as this ratio applying a tight slippage would most likely cause a revert and temporary Denial-of-Service during the first day of the protocol.

Recommendation: Consider removing the early bird period or preventing liquidity addition failure via try-catch.

Resolution: Resolved.

5.3 Low

5.3.1 Referrers may claim their top contributor and referrer bonuses early

Severity: Low

Context: ULTI.sol

Description: When a user has their first bonuses allocated, a timestamp at which a cooldown period starts.

The problem is this timestamp is not set when a user that is not a depositor yet is a referrer to another user.

Recommendation: Consider updating the referrer's claim timestamp in the deposit flow if not set.

Resolution: Resolved.

5.3.2 The cumulative deposits validation is insufficient

Severity: Low

Context: ULTI.sol

Description: When updating the streak count of a user, a cumulative deposits check is in place ensuring that total deposits for this cycle are greater than the ones for the previous, but not more than 10x. This is done to incentivize users to gradually increase their participation in the protocol.

The problem is that a user may still deposit some dust amount during the first several cycles and that way reach to the *STREAK_BONUS_COUNT_START* without actually providing valuable deposits over the necessary number of cycle.

Recommendation: Consider requiring a minimum deposit amount to prevent dust entries.

Resolution: Resolved.

5.3.3 Cycle maintenance should be performed at the beginning of a new cycle

Severity: Low

Context: ULTI.sol

Description: Cycle maintenance allocates the top contributor bonuses for the previous cycle and collect fees from the Uniswap pool accumulated since the last maintenance.

The problem is that it is performed at the end of a pump instead of the beginning hence the collected fees are not included in the balance of the contract for the current cycle, but for the next one.

Recommendation: Consider performing cycle maintenance at the start of a pump.

Resolution: Resolved.

5.3.4 Fees for the first cycle could be collected

Severity: Low

Context: ULTI.sol

Description: When performing a cycle maintenance, the fees accumulated in the pool are being collected so that they can be included in the balance for the next cycle distribution.

The problem is that collecting the fees for the first cycle is skipped as no maintenance is performed.

Recommendation: Consider performing cycle maintenance for the first cycle.

Resolution: Resolved.

5.3.5 Skin-in-the-game cap applied to total bonuses rather than just referrals

Severity: Low

Context: ULTI.sol

Description: The skin-in-the-game cap is a mechanism which makes sure that the referrer bonuses are capped at 10x the total deposits.

The problem is that currently the cap is not compared to the total referral bonuses ever received by the given referrer, but to the amount of bonuses they have pending to claim.

Recommendation: Consider tracking and using the total referrer bonuses only instead of all pending bonuses to claim.

Resolution: Resolved.

5.3.6 TWAP cached in storage may mislead external parties

Severity: Low

Context: ULTI.sol

Description: The TWAP is currently obtained through the UTLI:WETH Uniswap pool, then cached in storage and used in different functionalities.

The problem is that when returned via the *getGlobalData()* getter, it might mislead the caller as it could be not up-to-date.

Recommendation: Consider returning the *twap* from *_updateTwap* and using it as a local variable instead of caching in storage.

Resolution: Resolved.

5.4 Informational

5.4.1 Typographical mistakes, non-critical issues and code-style suggestions

Severity: Informational

Context: ULTI.sol

Description: The contract contains one or more typographical mistakes, non-critical issues and code-style suggestions. In an effort to keep the report size reasonable, we enumerate these below:

1. The *getCurrentDayInCycle()* function could be simplified to just *(block.timestamp % ULTI_CYCLE_DURATION) / 1 days + 1* and does not need to return an *uint8*.
2. Redundant DoS check in *_createLiquidity* as the factory would revert if the pool has already been created anyway.
3. Could use *_transferOwnership(address(0))* instead of *renounceOwnership*.
4. Consider using the latest Solidity compiler version.
5. The *DepositLiquiditySlippageTooHigh* custom error is never used.
6. *STREAK_BONUS_PERCENTAGE_INCREASE_PER_CYCLE* is never used.
7. *founderGiveaway* is never used - could simply emit *msg.value* in *launch()*.
8. 9._ = 0;_ can be removed everywhere as the default *uint* value is 0.
9. The *fallback()* function could be removed.
10. *_updatePumpCount* can simply add the new pumper and do ++ in both cases.
11. Extra space in a comment line: *the streak bonus*.
12. *bonys* should be *bonus*.
13. *scaled by 1e16* should say *scaled by 1e6*.
14. Checking *&& referrer != msg.sender* in *_updateReferrals* is redundant as it has already been validated in *deposit*.
15. Incorrect comment: *Each value is scaled by 10^6*, should be 10^8.
16. Unnecessary to declare *ethInLP* and *deadline* local variables in *launch*.
17. The first and second action in the NatSpec comment for *launch* should be the other way around.
18. *// Third-pary interfaces -> // Third-party interfaces*.
19. Consider using a constant for the minimum ETH required (33e18) to launch instead of computing it in the function (just to adhere to the convention other constants apply).
20. Incorrect comment: */// - To apply multiplier: actualValue = (originalValue * _getAdoptionMultiplier(cycleNumber)) / 1_000_000*, should use 100_000_000 (1e8).
21. Unclear comment: ** @dev Requires the contract to have sufficient WETH balance to process withdrawals*.
22. *IERC20* import never used.
23. The comments numbering in *_updateStreakCount* is incorrect.
24. Unnecessary check in *deposit - && block.timestamp >= nextDepositOrClaimTimestamps[msg.sender]*.
25. Unnecessary to declare a local variable *ethDeposited* instead of using directly *msg.value* in *deposit*.
26. The *unstoppable* modifier is redundant - the functions will revert anyway if the pool address has not been set.
27. Use *isUltiToken0* instead of *address(this) < wethAddress*.

Resolution: Resolved.