# Ulti Security Review

## Pashov Audit Group

Conducted by: T1MOH, samuraii77, btk

November 25th - December 1st

# 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* - 926e9bb4ba8a3d2ebb60c2f88ba645b31caad05c

*fixes review commit hash* - 4ec79b2fd3862b3ad11c21601774428dcf1a3d02

## Scope

The following smart contracts were in scope of the audit:

- `ULTI`
- `FullMath`
- `LiquidityAmounts`
- `Oracle`
- `PoolAddress`
- `TickMath`

# 7. Executive Summary

Over the course of the security review, T1MOH, samuraii77, btk engaged with Ulti to review Ulti. In this period of time a total of **3** issues were uncovered.

## Protocol Summary

| | |
|---|---|
| **Protocol Name** | Ulti |
| **Repository** | https://github.com/ulti-org/ulti-protocol-contract |
| **Date** | November 25th - December 1st |
| **Protocol Type** | Farming protocol |

## Findings Count

| **Severity** | **Amount** |
|---|---|
| Medium | 1 |
| Low | 2 |
| **Total Findings** | **3** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [M-01] | Malicious actors can grief the LP contribution | Medium | Resolved |
| [L-01] | pump() behaves incorrectly in case nobody deposited in the previous cycle | Low | Resolved |
| [L-02] | _getLiquidityAmounts() returns a wrong amount of tokens in position | Low | Resolved |

# 8. Findings

## 8.1. Medium Findings

## [M-01] Malicious actors can grief the LP contribution

### Severity

**Impact:** High

**Likelihood:** Low

### Description

Depositing ETH into the ULTI protocol is the primary mechanism for acquiring ULTI tokens and engaging with the ecosystem. The `deposit()` function includes a deadline parameter, which is passed through multiple internal calls (_allocateDeposit -> _tryIncreaseLiquidity -> increaseLiquidity):

```
function deposit(
        uint256 inputTokenAmount,
        address referrer,
        uint256 minUltiToAllocate,
        uint256 deadline,
        bool autoClaim
    ) external nonReentrant unstoppable {
```

However, there is no validation to ensure that the deadline parameter is not already expired. This oversight allows malicious users to exploit the system by intentionally providing an outdated deadline. When this occurs, the try nfpm.increaseLiquidity function fails silently:

```
try nonfungiblePositionManager.increaseLiquidity(increaseParams) {}
        catch {
            // Skip, failing to add liquidity should never block deposits
            // The input token dedicated to liquidity will instead be used to
            // pump
        }
```

Resulting in LP contributions (inputTokenForLP and ultiForLP) remaining in the contract for future pumps.

# Recommendations

Revert when `block.timestamp > deadline`.

# 8.2. Low Findings

## [L-01] `pump()` behaves incorrectly in case nobody deposited in the previous cycle

In case nobody deposited in the previous cycle, function `_performCycleMaintenance()` will always execute in the current cycle's pumps

```
function pump(uint256 maxInputTokenPerUlti, uint256 deadline)
        external
        nonReentrant
        unstoppable
        returns (uint256 inputTokenToSwap, uint256 ultiToBurn)
    {
        ...

        // 2. Perform cycle maintenance if needed
        uint32 cycle = getCurrentCycle();
        if (cycle > 1 && !isTopContributorsBonusAllocated[cycle - 1]) {
@>          _performCycleMaintenance(cycle - 1);
        }
        ...
```

That's because `_allocateTopContributorsBonuses()` always returns early and doesn't mark the cycle as allocated:

```
function _performCycleMaintenance(uint32 cycle) private {
        if (cycle < 1) return;

        _allocateTopContributorsBonuses(cycle);

        _collectAndProcessLiquidityFees();

        _burnExcessUlti();
    }

    function _allocateTopContributorsBonuses(uint32 cycle) private {
        // 1. Get the total bonus amount allocated for this cycle
        uint256 topContributorsBonusAmount = topContributorsBonuses[cycle];

        // Skip if no bonuses amount were allocated for this cycle
        if (topContributorsBonusAmount == 0) {
            return;
        }
        ...
```

As a result: 1)pumps in the current cycle always require extra gas to claim fee; 2) Input token balance increases and therefore pump swaps more tokens than expected.

Recommendation: set allocated before early return

```
function _allocateTopContributorsBonuses(uint32 cycle) private {
        // 1. Get the total bonus amount allocated for this cycle
        uint256 topContributorsBonusAmount = topContributorsBonuses[cycle];

        // Skip if no bonuses amount were allocated for this cycle
        if (topContributorsBonusAmount == 0) {
+           isTopContributorsBonusAllocated[cycle] = true;
            return;
        }
        ...
```

# [L-02] `_getLiquidityAmounts()` returns a wrong amount of tokens in position

Upon `_getLiquidityAmounts()` being called, we return the input token amount and the `ULTI` token amount in the position:

> @return inputTokenAmountInPosition The amount of input token in the current liquidity position @return ultiAmountInPosition The amount of ULTI in the current liquidity position

However, the code to do that is incorrect and will return a wrong value if someone else has provided liquidity to the pool. This is how we compute the values discussed above:

```
uint128 liquidity = liquidityPool.liquidity();
...
(uint256 amount0, uint256 amount1) = LiquidityAmounts.getAmountsForLiquidity
  (sqrtPriceX96, sqrtRatioAX96, sqrtRatioBX96, liquidity);
...
```

We calculate `amount0` and `amount1` based on the liquidity of the pool. This is wrong as our position might not be the only one in the pool. If someone else has provided liquidity to the pool, the return values will also include his amounts which would be incorrect.

To properly calculate the amounts, fetch the liquidity from our position and use that liquidity for the `getAmoiuntsForLiquidity()` function input instead of the total liquidity for the pool.