Audit of Trinity

A report of findings by

Genji Sakamoto

December 9th, 2020

Table of Contents

Executive Summary

This audit report has been written to discover issues and vulnerabilities in the Trinity smart contract.

This process included a line by line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

Audited smart contract

- Trinity.sol

Audit Method

- Static analysis based on source.
- Dynamic analysis by testing deployed ones on Ropsten.

Audit Focus

- Contract logic.
- Vulnerabilities for common and uncommon attacks
- Gas optimization
- Validation for variable limiters
- Transparency for all users.

Conclusion

While auditing the Trinity smart contract, I found that it has many exciting and attractive features that could engage in marketing and ensure the liquidity pool's stable-ongoing growth.

The implementation of the contract is correct according to the tokenomics. But there are some small issues that I recommend fixing before deployment to ensure the high-secured, perfectness, and optimization of the contract.

Type of Issues

Title	Description	Issues	SWC ID
Integer Overflow and Underflow	An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type.	0	SWC-101
Function Incorrectness	Function implementation does not meet the specification, leading to intentional or unintentional vulnerabilities.	0	
Buffer Overflow	An attacker is able to write to arbitrary storage locations of a contract if array of out bound happens.	0	SWC-124
Reentrancy	A malicious contract can call back into the calling contract before the first invocation of the function is finished.	0	SWC-107
Transaction Order Dependence	A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	0	SWC-114
Timestamp Dependence	Timestamp can be influenced by minors to some degree.	1	SWC-116
Insecure Compiler Version	Using a fixed outdated compiler version or floating pragma can be problematic, if there are publicly disclosed bugs and issues that affect the current compiler version used.	0	SWC-102 SWC-103
Insecure Randomness	Block attributes are insecure to generate random numbers, as they can	0	SWC-120

	be influenced by minors to some degree.		
“tx.origin” for authorization	“tx.origin” should not be used for authorization. Use “msg.sender” instead.	1	SWC-115
Delegate call to Untrusted Calling	Calling into untrusted contracts is very dangerous, the target and arguments provided must be sanitized.	0	SWC-112
State Variable Default Visibility	Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.	1	SWC-108
Function Default Visibility	Functions are public by default. A malicious user is able to make unauthorized or unintended state changes if a developer forgot to set the visibility.	0	SWC-100
Uninitialized Variables	Uninitialized local storage variables can point to other unexpected storage variables in the contract.	0	SWC-109
Assertion Failure	The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.	0	SWC-110
Deprecated Solidity Features	Several functions and operators in Solidity are deprecated and should not be used as best practice.	0	SWC-111
Unused Variables	Unused variables reduce code quality.	0	

Findings

Trinity.sol

State Visibility

Variable visibility is not set for multiple variables.

A public variable is easily accessible to users of the contract, and while no specific issues were found with these variables identified, it is best practice to utilize the most restrictive visibility as possible unless it specifically needs to be public.

```
contract Balancer {
    using SafeMath for uint256;
    IUniswapV2Router02 public immutable _uniswapV2Router;
    TRINITY _tokenContract;
```

Use msg.sender rather than tx.origin

It is recommended to use msg.sender rather than tx.origin for security.

```
  _rOwned[tx.origin] = _rOwned[tx.origin].add(tRewardForCaller.mul(currentRate));
```

Perfect Condition Check

The contract validates _maxTxAmount and tradingEnabled flag for all users except owner, but it is missing one case.

Suppose the owner makes a transaction when the locked amount for liquidity is over the _minTokensBeforeSwap value. In that case, auto swap needs to be done, and both sender and recipient for the transaction are not owner, so it checks _maxTxAmount and tradingEnabled flag, and may fail in some cases.

It is perfect for checking inSwapAndLiquify flag together for checking if the original transaction maker is the owner or not.

```
if(sender != owner() && recipient != owner()) {
    require(amount <= _maxTxAmount, "Trinity: Transfer amount exceeds the maxTxAmount.");
    if((_msgSender() == currentPoolAddress || _msgSender() == address(_uniswapV2Router)) && !tradingEnabled)
        require(false, "Trinity: trading is disabled.");
}
```

Transparency

The contract uses the reward wallet that owner inputs when deploy, and for transparency, the owner needs to burn its private key and prove it to users.

It is perfect to use a new empty contract address to keep high transparency.

```
constructor (IUniswapV2Router02 uniswapV2Router, address rewardWallet, uint256 initialRewardLockAmount) public {
    _lastRebalance = now;

    _uniswapV2Router = uniswapV2Router;
    _rewardWallet = rewardWallet;
    _initialRewardLockAmount = initialRewardLockAmount;
```

Gas Optimization

The contract calls removeLiquidityETH without checking if amountToRemove is 0, and it is not good to optimize gas.

If the contract checks if the ETH/TRI LP amount locked in self is 0 in advance, it will become perfect for gas optimization.

```
function _rebalanceLiquidity() private {
   _lastRebalance = now;

   uint256 amountToRemove = IERC20(_uniswapETHPool).balanceOf(address(this)).mul(_liquidityRemoveFee).div(100);

   removeLiquidityETH(amountToRemove);
```

The contract swaps pair token by two swaps (ex: TRI-ETH, ETH-DAI), but it could be done in one step using the swapExactTokensForTokensSupportingFeeOnTransferTokens function of uniswapV2Router. The contract defines swapTokensForTokens function for it, but it is not used. I recommend to use it for gas optimization.

```
function swapTokensForTokens(address pairTokenAddress, uint256 tokenAmount) private {
    // generate the uniswap pair path of token -> weth
    address[] memory path = new address[](3);
    path[0] = address(this);
    path[1] = _uniswapV2Router.WETH();
    path[2] = pairTokenAddress;

    _approve(address(this), address(_uniswapV2Router), tokenAmount);


    // make the swap
    _uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        tokenAmount,
        0, // accept any amount of ETH
        path,
        address(this),
        block.timestamp
    );
}
```

```
function swapAndLiquifyForTokens(address pairTokenAddress, uint256 lockedBalanceForPool) private lockTheSwap {
    // split the contract balance into halves
    uint256 half = lockedBalanceForPool.div(2);
    uint256 otherHalf = lockedBalanceForPool.sub(half);

    uint256 initialBalance = address(this).balance;

    // swap tokens for ETH
    swapTokensForEth(half);

    // how much ETH did we just swap into?
    uint256 newBalance = address(this).balance.sub(initialBalance);

    // capture the contract's current pairToken balance.
    // this is so that we can capture exactly the amount of pairToken that the
    // swap creates, and not make the liquidity event include any pairToken that
    // has been manually sent to the contract
    uint256 initialPairTokenBalance = IERC20(pairTokenAddress).balanceOf(address(this));

    // swap eth for pairToken
    swapEthForTokens(pairTokenAddress, newBalance);


    // how much pairToken did we just swap into?
    uint256 newPairTokenBalance = IERC20(pairTokenAddress).balanceOf(address(this)).sub(initialPairTokenBalance);

    // add liquidity to uniswap
    addLiquidityForTokens(pairTokenAddress, otherHalf, newPairTokenBalance);

    emit SwapAndLiquify(pairTokenAddress, half, newPairTokenBalance, otherHalf);

    _sendRewardInterestToPool();
}
```

Interface Classification

Now the contract defined several functions of IUniswapV2Router01 in IUniswapV2Router02. Of course, IUniswapV2Router02 inherits IUniswapV2Router01, and there is no problem in the contract's running, but uniswap is open source, and I recommend to classify functions along with the original source.

```
interface IUniswapV2Router02 {
    function factory() external pure returns (address);
    function WETH() external pure returns (address);
```

Transparency and Security

- The owner can’t disable trading once made enable.

```
function _enableTrading() external onlyOwner() {
    tradingEnabled = true;
    TradingEnabled();
}
```

- The owner can’t set a max transaction amount limit under 1000 TRI.

```
function _setMaxTxAmount(uint256 maxTxAmount) external onlyOwner() {
    require(maxTxAmount >= 1000e9 , 'Trinity: maxTxAmount should be greater than 1000e9');
    _maxTxAmount = maxTxAmount;
    MaxTxAmountUpdated(maxTxAmount);
}
```

- There is not any mint function that somebody would ever use for an irregular purpose.

- The alchemy caller always should get TRI reward.

```
function _setLiquidityRemoveFee(uint256 liquidityRemoveFee) external onlyOwner() {
    require(liquidityRemoveFee >= 1 && liquidityRemoveFee <= 10 , 'Trinity: liquidityRemoveFee should be in 1 - 10');
    _liquidityRemoveFee = liquidityRemoveFee;
    LiquidityRemoveFeeUpdated(liquidityRemoveFee);
}
```